PyStan Documentation

Release 2.19.1.1

Allen B. Riddell

Oct 27, 2019
## Contents

1 Documentation .......... 3
2 Stan documentation .......... 45
3 Stan Discourse .......... 47
4 Important links .......... 49
5 Related projects .......... 51
6 Similar projects .......... 53
7 PyStan3 / Stan3 .......... 55
8 Indices and tables .......... 57

Python Module Index .......... 59
Index .......... 61
PyStan provides an interface to Stan, a package for Bayesian inference using the No-U-Turn sampler, a variant of Hamiltonian Monte Carlo.

License: Open source, GPL3
1.1 What’s New

1.1.1 v2.19.1.1 (25. Oct 2019)

• Update Stan source to v2.19.1 (release notes)
• Fix order of per-chain values returned by Fit summary (GH646)
• Inverse mass matrix can be provided (GH505)

1.1.2 v2.19.0.0 (31. Mar 2019)

• Update Stan source to v2.19.0 (release notes)
• Stan 2.19 requires a compiler which supports C++14. For example, if you use gcc as your compiler you will need to use gcc version 4.9.3 or higher.
• New sampler parameters for VB mode: log_p__ and log_g__ values are added to sample file (csv) if defined with sample_file.
• Reminder: Python 2 is not supported after 2019-12-31. PyStan will phase out testing of Python 2 after this point.

1.1.3 v2.18.1.0 (22. Dec 2018)

• Update Stan source to v2.18.1 (release notes)
• Performance improvements for extract method.
1.1.4 v2.18.0.0 (16. Aug 2018)

- Update Stan source to v2.18.0 (release notes)
- Fit method to_dataframe organizes draws into a Pandas DataFrame. Only works if Pandas is installed. Thanks to Liam Brannigan and Ari Hartikainen.
- Improved effective sample size calculation. Thanks to Aki Vehtari.
- Implemented pystan.diagnostics. Thanks to @jjramsey.
- Several minor bug fixes.

1.1.5 v2.17.1.0 (16. Jan 2018)

- Update Stan source to v2.17.1 (release notes) (bugfix release)
- Remove deprecation warnings (Thanks to Alexander Rudiuk)
- Drop testing of Python 3.4.

1.1.6 v2.17.0.0 (6. Oct 2017)

- Update Stan source to v2.17.0 (release notes),
- Added pystan.stansummary function. Patch by @ahartikainen.
- Marked pystan.stan as deprecated. It will be removed in version 3.0. Please compile and use a Stan program as two separate steps.
- Reminder: Consider using Python 3.5 or higher. In a future release, Python 3.4 wheels will no longer be built.
- Reminder: Stan v2.16.0 is the final release which will not require a C++11 compatible compiler. Future releases will require a C++11 compatible compiler. (This does not affect most users.)

1.1.7 v2.16.0.0 (22. June 2017)

- Update Stan source to v2.16.0 (release notes),
- Ari Hartikainen (Aalto University) @ahartikainen joins the Stan development team.
- Added pystan.lookup (contributed by Marco Inacio, @randommm)
- NOTE: Stan v2.16.0 is the final release which will not require a C++11 compatible compiler. Future releases will require a C++11 compatible compiler. The vast majority of users have a compatible compiler.

1.1.8 v2.15.0.1 (2. May 2017)

- Python 2.7 compatibility fix (#332). Thanks to @monga for the report.

1.1.9 v2.15.0.0 (21. Apr 2017)

- Update Stan source to v2.15.0 (release notes)
- Allow UTF-8 comments in Stan Program code. Thanks to @ahartikainen
- Expose constrain_pars method, thanks to Lars Mescheder.
1.1.10 V2.14.0.0 (1. Jan 2017)

- Update Stan source to v2.14.0 (release notes), includes important fix to the default sampling algorithm (NUTS). All users are encouraged to upgrade.
- Several documentation and minor bug fixes (thanks @ahartikainen, @jrings, @nesanders)
- New OpenPGP signing key for use with PyPI. Key fingerprint is C3542448245BEC68F43070E4CCB669D9761F0CAC.

1.1.11 V2.12.0.0 (15. Sept 2016)

- Update Stan source to v2.12.0 (release notes)
- #239 Fix bug in array indexing (thanks @stephen-hoover)
- #254 Fix off-by-one error in estimated sample size calculation

1.1.12 V2.11.0.0 (28. July 2016)

- Update Stan source to v2.11.0 (release notes)

1.1.13 V2.10.0.0 (18. July 2016)

- Update Stan source to v2.10.0 (release notes)
- Sampling in Fixed_param mode now works. Thanks to @luac for the fix and @axch for the original report.
- Detailed installation instructions from @chendaniely added to the documentation.

1.1.14 v2.9.0.0 (7. Jan 2016)

- Update Stan source to v2.9.0 (release notes)
- Bugs fixed in _chains.pyx and model.py (thanks to @stephen-hoover, Paul Kernfeld)

1.1.15 v2.8.0.2 (6. Nov 2015)

- Cython 0.22 or higher requirement included on PyPI

1.1.16 v2.8.0.1 (5. Nov 2015)

- Python 3.5 support added
- Cython 0.22 or higher now required
- Compiler optimization (−O2) turned on for model compilation. This should increase sampling speed.
- Significant bug fixes (pickling, pars keyword)

1.1.17 v2.8.0.0 (1. Oct 2015)

- Update Stan source to v2.8.0 (release notes)
1.1.18 v2.7.0.1 (22. August 2015)
   • Minor Cython 0.23.1 compatibility fixes
   • Bug preventing mean_pars from being recorded

1.1.19 v2.7.0.0 (21. July 2015)
   • Update Stan source to v2.7.0 (release notes)

1.1.20 v2.6.3.0 (21. Mar 2015)
   • Update Stan source to v2.6.3 (release notes).

1.1.21 v2.6.0.0 (9. Feb 2015)
   • Update Stan source to v2.6.0 (release notes).

1.1.22 v2.5.0.2 (30. Jan 2015)
   • Fix bug in rdump (for >1 dimensional arrays)

1.1.23 v2.5.0.1 (14. Nov 2014)
   • Support for pickling fit objects (experimental)
   • Fix bug that occurs when printing fit summary

1.1.24 v2.5.0.0 (21. Oct 2014)
   • Update Stan source to v2.5.0
   • Fix several significant bugs in the extract method

1.1.25 v2.4.0.3 (9. Sept 2014)
   • Performance improvements for the printed summary of a fit.

1.1.26 v2.4.0.2 (6. Sept 2014)
   • Performance improvements for the extract method (5-10 times faster)
   • Performance improvements for the printed summary of a fit. Printing a summary of a model with more than a hundred parameters is not recommended. Consider using extract and calculating summary statistics for the parameters of interest.
1.1.27 v2.4.0.1 (31. July 2014)

- Sets LBFGS as default optimizer.
- Adds preliminary support for Python binary wheels on OS X and Windows.
- Fixes bug in edge case in new summary code.

1.1.28 v2.4.0.0 (26. July 2014)

- Stan 2.4 (LBFGS optimizer added, Nesterov removed)
- Improve display of fit summaries

1.1.29 v2.3.0.0 (26. June 2014)

- Stan 2.3 (includes user-defined functions, among other improvements).
- Optimizing returns a vector (array) by default instead of a dictionary.

1.1.30 v2.2.0.1 (30. April 2014)

- Add support for reading Stan’s R dump files.
- Add support for specifying parameters of interest in `stan`
- Add Windows installation instructions. Thanks to @patricksnape.
- Lighten source distribution.

1.1.31 v2.2.0.0 (16. February 2014)

- Updates Stan to v2.2.0.

1.1.32 v2.1.0.1 (27. January 2014)

- Implement model name obfuscation. Thanks to @karnold
- Improve documentation of StanFit objects

1.1.33 v2.1.0.0 (26. December 2013)

- Updates Stan code to v2.1.0.

1.1.34 v2.0.1.3 (18. December 2013)

- Sampling is parallel by default.
- `grad_log_prob` method of fit objects is available.
1.1.35  v2.0.1.2 (1. December 2013)

- Improves setuptools support.
- Allows sampling chains in parallel using multiprocessing. See the n_jobs parameter for stan() and the sampling method.
- Allows users to specify initial values for chains.

1.1.36  v2.0.1.1 (18. November 2013)

- Clean up random_seed handling (Stephan Hoyer).
- Add fit methods get_seed, get_inits, and get_stancode.

1.1.37  v2.0.1.0 (24. October 2013)

- Updated to Stan 2.0.1.
- Specifying sample_file now works as expected.

1.1.38  v2.0.0.1 (23. October 2013)

- Stan array parameters are now handled correctly.
- Ancillary methods added to fit instances.
- Fixed bug that caused parameters in control dict to be ignored.

1.1.39  v2.0.0.0 (21. October 2013)

- Stan source updated to to 2.0.0.
- PyStan version now mirrors Stan version.
- Rudimentary plot and traceplot methods have been added to fit instances.
- Warmup and sampling progress now visible.

1.1.40  v.0.2.2 (28. September 2013)

- log_prob method available from StanFit instances.
- Estimated sample size and Rhat included in summary.

1.1.41  v.0.2.1 (17. September 2013)

- StanModel instances can now be pickled.
- Adds basic support for saving output to sample_file.
1.1.42 v.0.2.0 (25. August 2013)

- optimizing method working for scalar, vector, and matrix parameters
- stanfit objects now have summary and __str__ methods à la RStan
- stan source updated to commit cc82d51d492d26f754fd56efe22a99191c80217b (July 26, 2013)
- IPython-relevant bug fixes

1.1.43 v.0.1.1 (19. July 2013)

- Support for Python 2.7 and Python 3.3
- stan and stanc working with common arguments

1.2 Getting started

PyStan is the Python interface for Stan.

1.2.1 Prerequisites

PyStan has the following dependencies:

- Python: 2.7, >=3.3
- Cython: >=0.22
- NumPy: >=1.7

PyStan also requires that a C++ compiler be available to Python during installation and at runtime. On Debian-based systems this is accomplished by issuing the command `apt-get install build-essential`.

1.2.2 Installation

**Note:** Installing PyStan involves compiling Stan. This may take a considerable amount of time.

**Unix-based systems including Mac OS X**

PyStan and the required packages may be installed from the Python Package Index using `pip`.

```
pip install pystan
```

Mac OS X users encountering installation problems may wish to consult the PyStan Wiki for possible solutions.

**Windows**

PyStan on Windows requires Python 2.7/3.x and a working C++ compiler. If you have already installed Python and the MingW-w64 C++ compiler, running `pip install pystan` will install PyStan.

If you need to install a C++ compiler, you will find detailed installation instructions in `PyStan on Windows`. 

1.2. Getting started
1.2.3 Using PyStan

The module’s name is `pystan` so we load the module as follows:

```python
import pystan
```

**Example 1: Eight Schools**

The “eight schools” example appears in Section 5.5 of Gelman et al. (2003), which studied coaching effects from eight schools.

```stan
schools_code = ""
data {
  int<lower=0> J; // number of schools
  vector[J] y; // estimated treatment effects
  vector<lower=0>[J] sigma; // s.e. of effect estimates
}
parameters {
  real mu;
  real<lower=0> tau;
  vector[J] eta;
}
transformed parameters {
  vector[J] theta;
  theta = mu + tau * eta;
}
model {
  eta ~ normal(0, 1);
  y ~ normal(theta, sigma);
}
""
```

```python
schools_dat = {'J': 8,
               'y': [28, 8, -3, 7, -1, 1, 18, 12],
               'sigma': [15, 10, 16, 11, 9, 11, 10, 18]}

sm = pystan.StanModel(model_code=schools_code)
fit = sm.sampling(data=schools_dat, iter=1000, chains=4)
```

In this model, we let `theta` be transformed parameters of `mu` and `eta` instead of directly declaring `theta` as parameters. By parameterizing this way, the sampler will run more efficiently.

In PyStan, we can also specify the Stan model using a file. For example, we can download the file `8schools.stan` into our working directory and use the following call to `stan` instead:

```python
sm = pystan.StanModel(file='8schools.stan')
fit = sm.sampling(data=schools_dat, iter=1000, chains=4)
```

Once a model is compiled, we can use the StanModel object multiple times. This saves us time compiling the C++ code for the model. For example, if we want to sample more iterations, we proceed as follows:

```python
fit2 = sm.sampling(data=schools_dat, iter=10000, chains=4)
```

The object `fit`, returned from function `stan` stores samples from the posterior distribution. The `fit` object has a number of methods, including `plot` and `extract`. We can also print the `fit` object and receive a summary of the posterior samples as well as the log-posterior (which has the name `lp__`).
The method `extract` extracts samples into a dictionary of arrays for parameters of interest, or just an array.

```python
la = fit.extract(permuted=True)  # return a dictionary of arrays
mu = la['mu']
## return an array of three dimensions: iterations, chains, parameters
a = fit.extract(permuted=False)
```

Print the fit:

```python
print(fit)
```

If `matplotlib` and `scipy` are installed, a visual summary may also be displayed using the `plot()` method.

```python
fit.plot()
```

## 1.3 Detailed Installation Instructions

The following is addressed to an audience who is just getting started with Python and would benefit from additional guidance on how to install PyStan.

Installing PyStan requires installing:

- Python
- C++ -compiler
- Python dependencies
- PyStan

### 1.4 Prerequisite knowledge

It is highly recommended to know what `bash` is and the basics of navigating the terminal. You can review or learn it from the Software Carpentry (bash lesson here).

Lessons 1 - 3 are probably the most important.

### 1.4.1 Installing Python

The easiest way to install Python is to use the Anaconda distribution of python. It can be downloaded [here](#).

This is because PyStan (and many python tools) require packages (aka modules) that have C dependencies. These types of dependencies are unable to be installed (at least easily) using `pip`, which is a common way to install python packages. Anaconda ships with it's own package manager (that also plays nicely with `pip`) called `conda`, and comes with many of the data analytics packages and dependencies pre-installed.

Don’t worry about Anaconda ruining your current Python installation, it can be easily uninstalled (described below).

**Anaconda is not a requirement**

Anaconda is not an absolute requirement to get `pystan` to work. As long as you can get the necessary python dependencies installed, pystan will work. If you want to install Anaconda, follow the Windows, Macs, and Linux instructions below.
Linux

After downloading the installer execute the associated shell script. For example, if the file downloaded were named `Anaconda3-4.1.1-Linux-x86_64.sh` you would enter `bash Anaconda3-4.1.1-Linux-x86_64.sh` in the directory where you downloaded the file.

Macs

After downloading the installer, double click the .pkg file and follow the instructions on the screen. Use all of the defaults for installation.

Windows

PyStan on Windows is partially supported. See PyStan on Windows.

The Anaconda installer should be able to be double-clicked and installed. Use all of the defaults for installation except make sure to check Make Anaconda the default Python.

Uninstalling Anaconda

The default location for anaconda can be found in your home directory. Typically this means it in the `~/anaconda` or `~/anaconda3` directory when you open a terminal.

1.4.2 Python dependencies

If you used the Anaconda installer, numpy and cython should already be installed, so additional dependencies should not be needed. However, should you need to install additional dependencies, we can use `conda` to install them as such:

- open a terminal
- type `conda install numpy` to install numpy or replace `numpy` with the package you need to install
- common additional packages include `arviz,matplotlib,pandas,scipy`

1.4.3 Setting up C++ compiler

PyStan 2.19+ needs a C++14 compatible compiler. For GCC 4.9.3+ and GCC 5+ versions are up-to-date. To update your compiler, follow general instructions given for each platform.

- Linux: Install compiler with `apt/yum`
- Macs: Install latest XCode, or use `brew` or `macports`
- Windows: Follow instructions in PyStan on Windows.

To use gcc/clang compiler from the conda follow instructions in https://docs.conda.io/projects/conda-build/en/latest/resources/compiler-tools.html and set-up your CC and CXX environmental variables as given below. Instructions below assume default Anaconda environment. For conda-forge follow installation instructions in anaconda.org.

- open a terminal
- **Linux**: `conda install gcc_linux-64 gxx_linux-64 -c anaconda`
- **Macs**: `conda install clang_osx-64 clangxx_osx-64 -c anaconda`
To check your compiler version

- Open a terminal
- **Linux:** `type gcc --version`
- **OSX:** `type clang --version`

To use specific C++ compiler (Linux: `gcc_linux-64`, `gxx_linux-64`, OSX: `clang_osx-64`, `clangxx_osx-64`), either update your `CC` and `CXX` environmental variables or set-up your path to include folder containing compilers (e.g., `which gcc_linux-64``)

- `type export CC=gcc_linux-64` and `export CXX=g++_linux-64`
- `type export PATH=/path/to/Anaconda/bin:$PATH`

Conda will create a new name for the compiler. You’ll need to search in the `<path to conda>/bin` folder to find the name. *which python* will show this location. For example:

- Open a terminal
- **OSX:** `type ls <path to python env>/bin/ | grep clang`

You should see two compilers like `x86_64-apple-darwin13.4.0-clang` and `x86_64-apple-darwin13.4.0-clang++`.

- `type export CC=x86_64-apple-darwin13.4.0-clang` and `export CXX=x86_64-apple-darwin13.4.0-clang++`

### 1.4.4 Installing PyStan

Since we have the `numpy` and `cython` dependencies we need, we can install the latest version of PyStan using pip. To do so:

- Open a terminal
- `type pip install pystan`

### 1.5 Installing PyStan with Support for Stiff ODE Solvers (CVODES)

Those using Stan functions which require the use of the SUNDIALS library (e.g., `integrate_ode_bdf`) should use the following instructions to install PyStan.

First, make sure that you have installed the following packages:

- Cython
- Numpy

Now install a version of PyStan which compiles and uses the SUNDIALS library:

```
pip install https://github.com/stan-dev/pystan/archive/v2.18.0.0-cvodes.tar.gz
```

(Support for the SUNDIALS library is not included by default because it slows down compilation of every Stan program by several seconds.)

Consult the “Stan Language Manual” (linked to in the ‘Stan Documentation’) for an example of a complete Stan program with a system definition and solver call.
1.6 Optimization in Stan

PyStan provides an interface to Stan’s optimization methods. These methods obtain a point estimate by maximizing the posterior function defined for a model. The following example estimates the mean from samples assumed to be drawn from normal distribution with known standard deviation:

Specifying an improper prior for $\mu$ of $p(\mu) \propto 1$, the posterior obtains a maximum at the sample mean. The following Python code illustrates how to use Stan’s optimizer methods via a call to `optimizing`:

```python
import pystan
import numpy as np

ocode = ""
data {
   int<lower=1> N;
   real y[N];
}
parameters {
   real mu;
}
model {
   y ~ normal(mu, 1);
}
"
sm = pystan.StanModel(model_code=ocode)
y2 = np.random.normal(size=20)
np.mean(y2)
op = sm.optimizing(data=dict(y=y2, N=len(y2)))
op
```

1.7 Avoiding recompilation of Stan models

Compiling models takes time. It is in our interest to avoid recompiling models whenever possible. If the same model is going to be used repeatedly, we would like to compile it just once. The following demonstrates how to reuse a model in different scripts and between interactive Python sessions.

Within sessions you can avoid recompiling a model by reusing the `StanModel` instance:

```python
from pystan import stan

# bernoulli model
model_code = ""
   data {
      int<lower=0> N;
      int<lower=0,upper=1> y[N];
   }
   parameters {
      real<lower=0,upper=1> theta;
   }
   model {
      theta ~ beta(0.5, 0.5); // Jeffreys' prior
      for (n in 1:N)
         y[n] ~ bernoulli(theta);
```
from pystan import StanModel
data = dict(N=10, y=[0, 1, 0, 1, 0, 1, 0, 1, 1, 1])
sm = StanModel(model_code=model_code)
fit = sm.sampling(data=data)
print(fit)

# reuse model with new data
new_data = dict(N=6, y=[0, 0, 0, 0, 0, 1])
fit2 = sm.sampling(data=new_data)
print(fit2)

It is also possible to share models between sessions (or between different Python scripts). We do this by saving compiled models (StanModel instances) in a file and then reloading it when we need it later. (In short, StanModel instances are picklable.)

The following two code blocks illustrate how a model may be compiled in one session and reloaded in a subsequent one using pickle (part of the Python standard library).

```python
import pickle
from pystan import StanModel

# using the same model as before
data = dict(N=10, y=[0, 1, 0, 1, 0, 1, 0, 1, 1, 1])
sm = StanModel(model_code=model_code)
fit = sm.sampling(data=data)
print(fit)

# save it to the file 'model.pkl' for later use
with open('model.pkl', 'wb') as f:
    pickle.dump(sm, f)

The following block of code might appear in a different script (in the same directory).

```python
import pickle
sm = pickle.load(open('model.pkl', 'rb'))
new_data = dict(N=6, y=[0, 0, 0, 0, 0, 1])
fit2 = sm.sampling(data=new_data)
print(fit2)
```

1.7.1 Automatically reusing models

For those who miss using variables across sessions in R, it is not difficult to write a function that automatically saves a copy of every model that gets compiled and opportunistically loads a copy of a model if one is available.
def StanModel_cache(model_code, model_name=None, **kwargs):
    """Use just as you would 'stan'"""
    code_hash = md5(model_code.encode('ascii')).hexdigest()
    if model_name is None:
        cache_fn = 'cached-model-{}.pkl'.format(code_hash)
    else:
        cache_fn = 'cached-{}-{}.pkl'.format(model_name, code_hash)
    try:
        sm = pickle.load(open(cache_fn, 'rb'))
    except:
        sm = pystan.StanModel(model_code=model_code)
        with open(cache_fn, 'wb') as f:
            pickle.dump(sm, f)
    else:
        print("Using cached StanModel")
    return sm

# with same model_code as before
data = dict(N=10, y=[0, 1, 0, 0, 0, 0, 0, 0, 0, 1])
sm = StanModel_cache(model_code=model_code)
fit = sm.sampling(data=data)
print(fit)

new_data = dict(N=6, y=[0, 0, 0, 0, 0, 1])
# the cached copy of the model will be used
sm = StanModel_cache(model_code=model_code)
fit2 = sm.sampling(data=new_data)
print(fit2)

1.8 HMC: Euclidian Metric

Euclidian Metric (also known as mass matrix) is one of the tuned parameters for hmc algorithm. See hmc-algorithm-parameters.

Two examples are shown where it is possible to use pre-tuned metric-matrix with other pre-tuned parameters.

1.8.1 Example 1: (pseudo-)continue sampling

The first example shows an example how to (pseudo-)continue chains to get more draws to old fit. The method is described as pseudo due to fact that continued sampling does not equal sampling done in one step. With user provided seed, this procedure is reproducible. This procedure needs some manual processing and ArviZ is recommended for further analysis. To make this reproducible, user can manually set seed for each fit.

```python
from pystan import StanModel
from pystan.constants import MAX_UINT

# bernoulli model
model_code = ""
    data {
        int<lower=0> N;
        int<lower=0,upper=1> y[N];
    }
    parameters {
```
real<lower=0,upper=1> theta;
}
model {
    theta ~ beta(0.5, 0.5); // Jeffreys' prior
    for (n in 1:N)
        y[n] ~ bernoulli(theta);
}
//

data = dict(N=10, y=[0, 1, 0, 1, 0, 1, 0, 1, 1, 1])
sm = StanModel(model_code=model_code)
# initial seed can also be chosen by user
# MAX_UINT = 2147483647
seed = np.random.randint(0, MAX_UINT, size=1)
fit = sm.sampling(data=data, seed=seed)
# reuse tuned parameters
stepsize = fit.get_stepsize()
# by default .get_inv_metric returns a list
inv_metric = fit.get_inv_metric(as_dict=True)
init = fit.get_last_position()

# increment seed by 1
seed2 = seed + 1
control = {
    "stepsize" : stepsize,
    "inv_metric" : inv_metric,
    "adapt_engaged" : False
}
fit2 = sm.sampling(data=data,
    warmup=0,
    iter=1000,
    control=control,
    init=init,
    seed=seed2)

User needs to combine the draws manually. This can be done with `numpy.concatenate` or with `arviz.concat` with `dim="draw"` option. Combined draws can be further analyzed with ArviZ.

**Case 1: numpy.concatenate**

```python
# combine arrays with numpy
extract1 = fit1.extract()
extract2 = fit2.extract()
combined = {}
for key in extract1.keys():
    arr1 = extract1[key]
    arr2 = extract2[key]
    combined[key] = np.concatenate((arr1, arr2), axis=0)
import arviz as az
# transform shape order from (draw, chain, *shape) to (chain, draw, *shape)
for key in combined.items():
    combined[key] = np.swapaxes(combined[key], 0, 1)
```

(continues on next page)
inference_data = az.from_dict(posterior=combined)

# summary
summary = az.summary(inference_data, index_origin=1)
print(summary) # pandas.DataFrame

# traceplot
az.plot_trace(inference_data)

Case 2: arviz.concat

# Needs ArviZ version 0.4.2+
import arviz as az
# create InferenceData objects
inference_data1 = az.from_pystan(fit1)
inference_data2 = az.from_pystan(fit2)
inference_data = az.concat(inference_data1, inference_data2, dim="draw")

# summary
summary = az.summary(inference_data, index_origin=1)
print(summary) # pandas.DataFrame

# traceplot
az.plot_trace(inference_data)

1.8.2 Example 2: Pretune hmc parameters

The seconds example goes through the process how to use pre-tuned variables. With large and slow models, it is sometimes convenient to reuse the prelearned tuning parameters. This enables user to skip warmup period for new chains. Example code will first run warmup for one chain and then reuse the inverse metric matrix, stepsize and last sample location for further chains.

```python
from pystan import StanModel

# bernoulli model
model_code = ""
    data {
        int<lower=0> N;
        int<lower=0,upper=1> y[N];
    }
    parameters {
        real<lower=0,upper=1> theta;
    }
    model {
        theta ~ beta(0.5, 0.5); // Jeffreys' prior
        for (n in 1:N)
            y[n] ~ bernoulli(theta);
    }
""

data = dict(N=10, y=[0, 1, 0, 1, 0, 1, 0, 1, 1, 1])
sm = StanModel(model_code=model_code)
```

(continues on next page)
The shape of the inserted inverse metric changes depending on the used algorithm. The default algorithm with NUTS is `diag_e`, which means that only the diagonal of the inverse metric matrix is used. This also means that the shape of the inverse metric is `(n_flatnames,)`. With other algorithm `dense_e` the full matrix is defined and the inverse metric matrix shape is `(n_flatnames, n_flatnames)`. The inverse metric matrix needs to be strictly positive definite.

The inverse metric matrix is given for the `.sampling` method inside the `control` dictionary. The `inv_metric` can be either iterable (list, tuple, ndarray), dictionary of iterable with chain order as the key or `inv_metric` can be a string (path to Rdump or JSON file with a parameter “inv_metric”).

### 1.9 Differences between PyStan and RStan

While PyStan attempts to maintain API compatibility with RStan, there are certain unavoidable differences between Python and R.

#### 1.9.1 Methods and attributes

Methods are invoked in different ways: `fit.summary()` and `fit.extract()` (Python) vs. `summary(fit)` and `extract(fit)` (R).

Attributes are accessed in a different manner as well: `fit.sim` (Python) vs. `fit@sim` (R).

#### 1.9.2 Dictionaries instead of Lists

Where RStan uses lists, PyStan uses (ordered) dictionaries.

Python:
1.9.3 Reusing models and saving objects

PyStan uses pickle to save objects for future use.

Python:

```python
import pickle
import pystan

# bernoulli model
model_code = ""
    data {
        int<lower=0> N;
        int<lower=0,upper=1> y[N];
    }
    parameters {
        real<lower=0,upper=1> theta;
    }
    model {
        for (n in 1:N)
            y[n] ~ bernoulli(theta);
    }
""

data = dict(N=10, y=[0, 1, 0, 0, 0, 0, 0, 0, 0, 1])
model = pystan.StanModel(model_code=model_code)
fit = model.sampling(data=data)

with open('model.pkl', 'wb') as f:
    pickle.dump(model, f, protocol=pickle.HIGHEST_PROTOCOL)

# load it at some future point
with open('model.pkl', 'rb') as f:
    model = pickle.load(f)

# run with different data
fit = model.sampling(data=dict(N=5, y=[1, 1, 0, 1, 0]))
```

R:

```r
library(rstan)

model = stan_model(model_code=model_code)
save(model, file='model.rdata')
```

See also *Avoiding recompilation of Stan models.*

If you are saving a large amount of data with pickle.dump, be sure to use the highest protocol version available. Earlier versions are limited in the amount of data they can save in a single file.
### 1.10 API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stan([file, model_name, model_code, fit, ...])</code></td>
<td>Fit a model using Stan.</td>
</tr>
<tr>
<td><code>stanc([file, charset, model_code, ...])</code></td>
<td>Translate Stan model specification into C++ code.</td>
</tr>
<tr>
<td><code>StanModel([file, charset, model_name, ...])</code></td>
<td>Model described in Stan’s modeling language compiled from C++ code.</td>
</tr>
</tbody>
</table>

`StanFit4model` instances are also documented on this page.

```python
pystan.stan(file=None, model_name='anon_model', model_code=None, fit=None, data=None, pars=None, chains=4, iter=2000, warmup=None, thin=1, init='random', seed=None, algorithm=None, control=None, sample_file=None, diagnostic_file=None, verbose=False, boost_lib=None, eigen_lib=None, include_paths=None, n_jobs=-1, allow_undefined=False, **kwargs)
```

Fit a model using Stan.

The `pystan.stan` function was deprecated in version 2.17 and will be removed in version 3.0. Compiling and using a Stan Program (e.g., for drawing samples) should be done in separate steps.

**Parameters**

- **file** (string, file-like object) – Model code must be found via one of the following parameters: file or model_code.
  
  If file is a filename, the string passed as an argument is expected to be a filename containing the Stan model specification.
  
  If file is a file object, the object passed must have a ‘read’ method (file-like object) that is called to fetch the Stan model specification.
- **charset** (string, optional) – If bytes or files are provided, this charset is used to decode. ‘utf-8’ by default.
- **model_code** (string) – A string containing the Stan model specification. Alternatively, the model may be provided with the parameter file.
- **model_name** (string, optional) – A string naming the model. If none is provided ‘anon_model’ is the default. However, if file is a filename, then the filename will be used to provide a name. ‘anon_model’ by default.
- **fit** (StanFit instance) – An instance of StanFit derived from a previous fit. None by default. If fit is not None, the compiled model associated with a previous fit is reused and recompilation is avoided.
- **data** (dict) – A Python dictionary providing the data for the model. Variables for Stan are stored in the dictionary as expected. Variable names are the keys and the values are their associated values. Stan only accepts certain kinds of values; see Notes.
- **pars** (list of string, optional) – A list of strings indicating parameters of interest. By default all parameters specified in the model will be stored.
- **chains** (int, optional) – Positive integer specifying number of chains. 4 by default.
- **iter** (int, 2000 by default) – Positive integer specifying how many iterations for each chain including warmup.
- **warmup** (int, iter//2 by default) – Positive integer specifying number of warmup (aka burnin) iterations. As warmup also specifies the number of iterations used for stepsize adaption, warmup samples should not be used for inference.
• thin(int, optional) – Positive integer specifying the period for saving samples. Default is 1.

• init((0, '0', 'random', function returning dict, list of dict), optional) – Specifies how initial parameter values are chosen: - 0 or ‘0’ initializes all to be zero on the unconstrained support. - ‘random’ generates random initial values. An optional parameter

  \( \text{init}_r \) controls the range of randomly generated initial values for parameters in terms of their unconstrained support;

  – list of size equal to the number of chains (\textit{chains}), where the list contains a dict with initial parameter values;

  – function returning a dict with initial parameter values. The function may take an optional argument \textit{chain_id}.

• seed(int or np.random.RandomState, optional) – The seed, a positive integer for random number generation. Only one seed is needed when multiple chains are used, as the other chain’s seeds are generated from the first chain’s to prevent dependency among random number streams. By default, seed is \text{random.randint}(0, \text{MAX_UINT}).

• algorithm(["NUTS", "HMC", "Fixed_param"], optional) – One of the algorithms that are implemented in Stan such as the No-U-Turn sampler (NUTS, Hoffman and Gelman 2011) and static HMC.

• sample_file(string, optional) – File name specifying where samples for all parameters and other saved quantities will be written. If not provided, no samples will be written. If the folder given is not writable, a temporary directory will be used. When there are multiple chains, an underscore and chain number are appended to the file name. By default do not write samples to file.

• diagnostic_file(string, optional) – File name specifying where diagnostic information should be written. By default no diagnostic information is recorded.

• boost_lib(string, optional) – The path to a version of the Boost C++ library to use instead of the one supplied with PyStan.

• eigen_lib(string, optional) – The path to a version of the Eigen C++ library to use instead of the one in the supplied with PyStan.

• include_paths(list of strings, optional) – Paths for #include files defined in Stan code.

• verbose(boolean, optional) – Indicates whether intermediate output should be piped to the console. This output may be useful for debugging. False by default.

• control(dict, optional) – A dictionary of parameters to control the sampler’s behavior. Default values are used if control is not specified. The following are adaptation parameters for sampling algorithms.

  These are parameters used in Stan with similar names:

  – adapt_engaged : bool
  – adapt_gamma : float, positive, default 0.05
  – adapt_delta : float, between 0 and 1, default 0.8
  – adapt_kappa : float, between default 0.75
- `adapt_t0`: float, positive, default 10
- `adapt_init_buffer`: int, positive, defaults to 75
- `adapt_term_buffer`: int, positive, defaults to 50
- `adapt_window`: int, positive, defaults to 25

In addition, the algorithm HMC (called ‘static HMC’ in Stan) and NUTS share the following parameters:
- `stepsize`: float, positive
- `stepsize_jitter`: float, between 0 and 1
- `metric`: str, {“unit_e”, “diag_e”, “dense_e”}

In addition, depending on which algorithm is used, different parameters can be set as in Stan for sampling. For the algorithm HMC we can set
- `int_time`: float, positive

For algorithm NUTS, we can set
- `max_treedepth`: int, positive

• `n_jobs` (int, optional) – Sample in parallel. If -1 all CPUs are used. If 1, no parallel computing code is used at all, which is useful for debugging.

• `allow_undefined` (boolean, False by default) – If True, the C++ code can be written even if there are undefined functions.

Returns fit

Return type StanFit instance

Other Parameters

• `chain_id` (int, optional) – `chain_id` can be a vector to specify the chain_id for all chains or an integer. For the former case, they should be unique. For the latter, the sequence of integers starting from the given `chain_id` are used for all chains.

• `init_r` (float, optional) – `init_r` is only valid if `init` == “random”. In this case, the initial values are simulated from [-`init_r, init_r`] rather than using the default interval (see the manual of (Cmd)Stan).

• `test_grad` (bool, optional) – If `test_grad` is True, Stan will not do any sampling. Instead, the gradient calculation is tested and printed out and the fitted StanFit4Model object is in test gradient mode. By default, it is False.

• `append_samples` (bool, optional)

• `refresh` (int, optional) – Argument `refresh` can be used to control how to indicate the progress during sampling (i.e. show the progress every code{refresh} iterations). By default, `refresh` is max(iter/10, 1).

• `obfuscate_model_name` (boolean, optional) – `obfuscate_model_name` is only valid if `fit` is None. True by default. If False the model name in the generated C++ code will not be made unique by the insertion of randomly generated characters. Generally it is recommended that this parameter be left as True.
Examples

```python
>>> from pystan import stan
>>> import numpy as np

>>> model_code = '''
... parameters {
...   real y[2];
... }
... model {
...   y[1] ~ normal(0, 1);
...   y[2] ~ double_exponential(0, 2);
... }'''
>>> fit1 = stan(model_code=model_code, iter=10)
>>> print(fit1)

>>> excode = '''
... transformed data {
...   real y[20];
...   y[1] = 0.5796; y[2] = 0.2276; y[3] = -0.2959;
...   y[7] = 0.7111; y[8] = 1.4424; y[9] = 2.5430;
...   y[10] = 0.3746; y[11] = 0.4773; y[12] = 0.1803;
...   y[13] = 0.5215; y[14] = -1.6044; y[15] = -0.6703;
...   y[16] = 0.9459; y[17] = -0.382; y[18] = 0.7619;
...   y[19] = 0.1006; y[20] = -1.7461;
... }
... parameters {
...   real mu;
...   real<lower=0, upper=10> sigma;
...   vector[2] z[3];
...   real<lower=0> alpha;
... }
... model {
...   y ~ normal(mu, sigma);
...   for (i in 1:3)
...     z[i] ~ normal(0, 1);
...   alpha ~ exponential(2);
... }'''
>>> def initfun1():
...   return dict(mu=1, sigma=4, z=np.random.normal(size=(3, 2)), alpha=1)
>>> exfit0 = stan(model_code=excode, init=initfun1)
>>> def initfun2(chain_id=1):
...   return dict(mu=1, sigma=4, z=np.random.normal(size=(3, 2)), alpha=1 + chain_id)
>>> exfit1 = stan(model_code=excode, init=initfun2)
```

`pystan.stanc` (file=None, charset='utf-8', model_code=None, model_name='anon_model', include_paths=None, verbose=False, obfuscate_model_name=True, allow_undefined=False)

Translate Stan model specification into C++ code.

**Parameters**

- `file (string, file, optional)`—If filename, the string passed as an argument is expected to be a filename containing the Stan model specification.

If file, the object passed must have a `read` method (file-like object) that is called to fetch the Stan model specification.
• **charset** *(string, 'utf-8' by default)* – If bytes or files are provided, this charset is used to decode.

• **model_code** *(string, optional)* – A string containing the Stan model specification. Alternatively, the model may be provided with the parameter *file*.

• **model_name** *(string, 'anon_model' by default)* – A string naming the model. If none is provided ‘anon_model’ is the default. However, if *file* is a filename, then the filename will be used to provide a name.

• **include_paths** *(list of strings, optional)* – Paths for #include files defined in Stan code.

• **verbose** *(boolean, False by default)* – Indicates whether intermediate output should be piped to the console. This output may be useful for debugging.

• **obfuscate_model_name** *(boolean, True by default)* – If False the model name in the generated C++ code will not be made unique by the insertion of randomly generated characters. Generally it is recommended that this parameter be left as True.

• **allow_undefined** *(boolean, False by default)* – If True, the C++ code can be written even if there are undefined functions.

Returns **stanc_ret** – A dictionary with the following keys: model_name, model_code, cpp_code, and status. Status indicates the success of the translation from Stan code into C++ code (success = 0, error = -1).

Return type  **dict**

**Notes**

C++ reserved words and Stan reserved words may not be used for variable names; see the Stan User’s Guide for a complete list.

The **include** method follows a C/C++ syntax #include foo/my_gp_funs.stan. The method needs to be at the start of the row, no whitespace is allowed. After the included file no whitespace or comments are allowed. **pystan.experimental’(PyStan 2.18) has a ‘fix_include-function to clean the #include statements from the model_code. Example: from pystan.experimental import fix_include model_code = fix_include(model_code)**

**See also:**

*StanModel()*  Class representing a compiled Stan model

*stan()*  Fit a model using Stan

**References**


**Examples**

```
>>> stanmodelcode = '''
... data {
...   int<lower=0> N;
...   real y[N];
... }
```
parameters {
  real mu;
}

model {
  mu ~ normal(0, 10);
  y ~ normal(mu, 1);
}

```python
>>> r = stanc(model_code=stanmodelcode, model_name = "normal1")

(sorted(r.keys()))
['cppcode', 'model_code', 'model_cppname', 'model_name', 'status']

>>> r['model_name']
'normal1'
```

```python
class pystan.StanModel(file=None, charset='utf-8', model_name='anon_model',
                       model_code=None, stanc_ret=None, include_paths=None, boost_lib=None,
                       eigen_lib=None, verbose=False, obfuscate_model_name=True, extra_compile_args=None,
                       allow_undefined=False, include_dirs=None, includes=None)
```

Model described in Stan’s modeling language compiled from C++ code.

Instances of StanModel are typically created indirectly by the functions `stan` and `stanc`.

Parameters

- **file**(string ('filename', 'file')) – If filename, the string passed as an argument is expected to be a filename containing the Stan model specification.
  
  If file, the object passed must have a ‘read’ method (file-like object) that is called to fetch the Stan model specification.

- **charset**(string, 'utf-8' by default) – If bytes or files are provided, this charset is used to decode.

- **model_name**(string, 'anon_model' by default) – A string naming the model. If none is provided 'anon_model' is the default. However, if file is a filename, then the filename will be used to provide a name.

- **model_code**(string) – A string containing the Stan model specification. Alternatively, the model may be provided with the parameter file.

- **stanc_ret**(dict) – A dict returned from a previous call to `stanc` which can be used to specify the model instead of using the parameter file or model_code.

- **include_paths**(list of strings) – Paths for #include files defined in Stan program code.

- **boost_lib**(string) – The path to a version of the Boost C++ library to use instead of the one supplied with PyStan.

- **eigen_lib**(string) – The path to a version of the Eigen C++ library to use instead of the one in the supplied with PyStan.

- **verbose**(boolean, False by default) – Indicates whether intermediate output should be piped to the console. This output may be useful for debugging.

- **allow_undefined**(boolean, False by default) – If True, the C++ code can be written even if there are undefined functions.
• **includes** *(list, None by default)* – If not None, the elements of this list will be assumed to be the names of custom C++ header files that should be included.

• **include_dirs** *(list, None by default)* – If not None, the directories in this list are added to the search path of the compiler.

• **kwargs** *(keyword arguments)* – Additional arguments passed to *stanc*.

**model_name**

*Type* string

**model_code**

Stan code for the model.

*Type* string

**model_cpp**

C++ code for the model.

*Type* string

**module**

Python module created by compiling the C++ code for the model.

*Type* builtins.module

**show()**

Print the Stan model specification.

**sampling()**

Draw samples from the model.

**optimizing()**

Obtain a point estimate by maximizing the log-posterior.

**get_cppcode()**

Return the C++ code for the module.

**get_cxxflags()**

Return the ‘CXXFLAGS’ used for compiling the model.

**get_include_paths()**

Return include_paths used for compiled model.

**See also:**

- **stanc** Compile a Stan model specification
- **stan** Fit a model using Stan

**Notes**

More details of Stan, including the full user’s guide and reference manual can be found at <URL: http://mc-stan.org/>.

There are three ways to specify the model’s code for *stan_model*.

1. parameter **model_code**, containing a string to whose value is the Stan model specification,
2. parameter **file**, indicating a file (or a connection) from which to read the Stan model specification, or
3. parameter **stanc_ret**, indicating the re-use of a model generated in a previous call to *stanc*.  

1.10. API
Examples

```python
>>> model_code = 'parameters {real y;} model {y ~ normal(0,1);}'
>>> model_code; m = StanModel(model_code=model_code)
... # doctest: +ELLIPSIS
'parameters ...'
>>> m.model_name
'anon_model'
```

`optimizing` *(data=None, seed=None, init='random', sample_file=None, algorithm=None, verbose=False, as_vector=True, **kwargs)*

Obtain a point estimate by maximizing the joint posterior.

Parameters

- **data** *(dict)* – A Python dictionary providing the data for the model. Variables for Stan are stored in the dictionary as expected. Variable names are the keys and the values are their associated values. Stan only accepts certain kinds of values; see Notes.

- **seed** *(int or np.random.RandomState, optional)* – The seed, a positive integer for random number generation. Only one seed is needed when multiple chains are used, as the other chain’s seeds are generated from the first chain’s to prevent dependency among random number streams. By default, seed is `random.randint(0, MAX_UINT)`.

- **init** *(0, '0', 'random', function returning dict, list of dict), optional)* – Specifies how initial parameter values are chosen: - 0 or ‘0’ initializes all to be zero on the unconstrained support. - ‘random’ generates random initial values. An optional parameter

  - `init_r` controls the range of randomly generated initial values for parameters in terms of their unconstrained support;

  - list of size equal to the number of chains (chains), where the list contains a dict with initial parameter values;

  - function returning a dict with initial parameter values. The function may take an optional argument `chain_id`.

- **sample_file** *(string, optional)* – File name specifying where samples for all parameters and other saved quantities will be written. If not provided, no samples will be written. If the folder given is not writable, a temporary directory will be used. When there are multiple chains, an underscore and chain number are appended to the file name. By default do not write samples to file.

- **algorithm** *("LBFGS", "BFGS", "Newton"), optional)* – Name of optimization algorithm to be used. Default is LBFGS.

- **verbose** *(boolean, optional)* – Indicates whether intermediate output should be piped to the console. This output may be useful for debugging. False by default.

- **as_vector** *(boolean, optional)* – Indicates an OrderedDict will be returned rather than a nested dictionary with keys ‘par’ and ‘value’.
Returns optim – Depending on as_vector, returns either an OrderedDict having parameters as keys and point estimates as values or an OrderedDict with components 'par' and 'value'. optim['par'] is a dictionary of point estimates, indexed by the parameter name. optim['value'] stores the value of the log-posterior (up to an additive constant, the lp__ in Stan) corresponding to the point identified by ‘optim’['par'].

Return type OrderedDict

Other Parameters

• iter (int, optional) – The maximum number of iterations.
• save_iterations (bool, optional)
• refresh (int, optional)
• init_alpha (float, optional) – For BFGS and LBFGS, default is 0.001.
• tol_obj (float, optional) – For BFGS and LBFGS, default is 1e-12.
• tol_rel_obj (int, optional) – For BFGS and LBFGS, default is 1e4.
• tol_grad (float, optional) – For BFGS and LBFGS, default is 1e-8.
• tol_rel_grad (float, optional) – For BFGS and LBFGS, default is 1e7.
• tol_param (float, optional) – For BFGS and LBFGS, default is 1e-8.
• history_size (int, optional) – For LBFGS, default is 5.
• Refer to the manuals for both CmdStan and Stan for more details.

Examples

```python
>>> from pystan import StanModel

>>> m = StanModel(model_code='parameters {real y;}
model {y ~ normal(0,1);}')

>>> f = m.optimizing()
```

sampling(data=None, pars=None, chains=4, iter=2000, warmup=None, thin=1, seed=None, init='random', sample_file=None, diagnostic_file=None, verbose=False, algorithm=None, control=None, n_jobs=-1, **kwargs)

Draw samples from the model.

Parameters

• data (dict) – A Python dictionary providing the data for the model. Variables for Stan are stored in the dictionary as expected. Variable names are the keys and the values are their associated values. Stan only accepts certain kinds of values; see Notes.
• pars (list of string, optional) – A list of strings indicating parameters of interest. By default all parameters specified in the model will be stored.
• chains (int, optional) – Positive integer specifying number of chains. 4 by default.
• iter (int, 2000 by default) – Positive integer specifying how many iterations for each chain including warmup.
• warmup (int, iter//2 by default) – Positive integer specifying number of warmup (aka burn-in) iterations. As warmup also specifies the number of iterations used for step-size adaption, warmup samples should not be used for inference. warmup=0 forced if algorithm="Fixed_param".
• **thin** (*int, 1 by default*) – Positive integer specifying the period for saving samples.

• **seed** (*int or np.random.RandomState, optional*) – The seed, a positive integer for random number generation. Only one seed is needed when multiple chains are used, as the other chain’s seeds are generated from the first chain’s to prevent dependency among random number streams. By default, seed is `random.randint(0, MAX_UINT)`.

• **algorithm** (*"NUTS", "HMC", "Fixed_param", optional*) – One of algorithms that are implemented in Stan such as the No-U-Turn sampler (NUTS, Hoffman and Gelman 2011), static HMC, or `Fixed_param`. Default is NUTS.

• **init** (*0, '0', 'random', function returning dict, list of dict, optional*) – Specifies how initial parameter values are chosen: 0 or ‘0’ initializes all to be zero on the unconstrained support; ‘random’ generates random initial values; list of size equal to the number of chains (*chains*), where the list contains a dict with initial parameter values; function returning a dict with initial parameter values. The function may take an optional argument `chain_id`.

• **sample_file** (*string, optional*) – File name specifying where samples for all parameters and other saved quantities will be written. If not provided, no samples will be written. If the folder given is not writable, a temporary directory will be used. When there are multiple chains, an underscore and chain number are appended to the file name. By default do not write samples to file.

• **verbose** (*boolean, False by default*) – Indicates whether intermediate output should be piped to the console. This output may be useful for debugging.

• **control** (*dict, optional*) – A dictionary of parameters to control the sampler’s behavior. Default values are used if control is not specified. The following are adaptation parameters for sampling algorithms.

These are parameters used in Stan with similar names:

- `adapt_engaged` : bool, default True
- `adapt_gamma` : float, positive, default 0.05
- `adapt_delta` : float, between 0 and 1, default 0.8
- `adapt_kappa` : float, between default 0.75
- `adapt_t0` : float, positive, default 10

In addition, the algorithm HMC (called ‘static HMC’ in Stan) and NUTS share the following parameters:

- `stepsize` : float or list of floats, positive
- `stepsize_jitter` : float, between 0 and 1
- `metric` : str, {"unit_e", "diag_e", "dense_e"}
- `inv_metric` : np.ndarray or str

In addition, depending on which algorithm is used, different parameters can be set as in Stan for sampling. For the algorithm HMC we can set

- `int_time` : float, positive

For algorithm NUTS, we can set

- `max_treedepth` : int, positive
• **n_jobs** *(int, optional)* – Sample in parallel. If -1 all CPUs are used. If 1, no parallel computing code is used at all, which is useful for debugging.

Returns *fit* – Instance containing the fitted results.

Return type StanFit4Model

Other Parameters

• **chain_id** *(int or iterable of int, optional)* – *chain_id* can be a vector to specify the chain_id for all chains or an integer. For the former case, they should be unique. For the latter, the sequence of integers starting from the given *chain_id* are used for all chains.

• **init_r** *(float, optional)* – *init_r* is only valid if *init* == “random”. In this case, the initial values are simulated from [-*init_r*, *init_r*] rather than using the default interval (see the manual of Stan).

• **test_grad** *(bool, optional)* – If *test_grad* is True, Stan will not do any sampling. Instead, the gradient calculation is tested and printed out and the fitted StanFit4Model object is in test gradient mode. By default, it is False.

• **append_samples** *(bool, optional)*

• **refresh** *(int, optional)* – Argument *refresh* can be used to control how to indicate the progress during sampling (i.e. show the progress every code{refresh} iterations). By default, *refresh* is max(iter/10, 1).

• **check_hmc_diagnostics** *(bool, optional)* – After sampling run `pystan.diagnostics.check_hmc_diagnostics` function. Default is True. Checks for n_eff and that skipped if the flat parameter count is higher than 1000, unless user explicitly defines check_hmc_diagnostics=True.

Examples

```python
>>> from pystan import StanModel
>>> m = StanModel(model_code='parameters {real y;} model {y ~ normal(0,1);}')
>>> m.sampling(iter=100)
```

vb *(data=None, pars=None, iter=10000, seed=None, init='random', sample_file=None, diagnostic_file=None, verbose=False, algorithm=None, **kwargs)*

Call Stan’s variational Bayes methods.

Parameters

• **data** *(dict)* – A Python dictionary providing the data for the model. Variables for Stan are stored in the dictionary as expected. Variable names are the keys and the values are their associated values. Stan only accepts certain kinds of values; see Notes.

• **pars** *(list of string, optional)* – A list of strings indicating parameters of interest. By default all parameters specified in the model will be stored.

• **seed** *(int or np.random.RandomState, optional)* – The seed, a positive integer for random number generation. Only one seed is needed when multiple chains are used, as the other chain’s seeds are generated from the first chain’s to prevent dependency among random number streams. By default, seed is random.randint(0, MAX_UINT).

• **sample_file** *(string, optional)* – File name specifying where samples for all parameters and other saved quantities will be written. If not provided, samples will be written to a temporary file and read back in. If the folder given is not writable, a temporary
directory will be used. When there are multiple chains, an underscore and chain number are appended to the file name. By default do not write samples to file.

- **diagnostic_file** *(string, optional)* – File name specifying where diagnostics for the variational fit will be written.

- **iter** *(−)* – Positive integer specifying how many iterations for each chain including warmup.

- **algorithm** *({'meanfield', 'fullrank'})* – One of "meanfield" and "fullrank" indicating which variational inference algorithm is used. meanfield: mean-field approximation; fullrank: full-rank covariance. The default is ‘meanfield’.

- **verbose** *(boolean, False by default)* – Indicates whether intermediate output should be piped to the console. This output may be useful for debugging.

- **optional parameters, refer to the manuals for both CmdStan (Other) –

  - **iter** –

  - **grad_samples** the number of samples for Monte Carlo enumerate of *(−)* – gradients, defaults to 1.

  - **elbo_samples** the number of samples for Monte Carlo estimate of ELBO *(−)* – (objective function), defaults to 100. (ELBO stands for “the evidence lower bound”).

  - **eta** positive stepsize weighting parameters for variational *(−)* – inference but is ignored if adaptation is engaged, which is the case by default.

  - **adapt_engaged** flag indicating whether to automatically adapt the *(−)* – stepsize and defaults to True.

  - **tol_rel_obj** convergence tolerance on the relative norm of the *(−)* – objective, defaults to 0.01.

  - **eval_elbo**, evaluate ELBO every Nth iteration, defaults to 100 *(−)* –

  - **output_samples** number of posterior samples to draw and save, *(−)* – defaults to 1000.

  - **adapt_iter** number of iterations to adapt the stepsize if *(−)* – adapt_engaged is True and ignored otherwise.

Returns **results** – Dictionary containing information related to results.

Return type **dict**

**Examples**

```python
>>> from pystan import StanModel
>>> m = StanModel(model_code='parameters {real y;} model {y ~ normal(0,1);}')
>>> results = m.vb()
>>> # results saved on disk in format inspired by CSV
>>> print(results['args']['sample_file'])
```
1.10.1 StanFit4model

Each StanFit instance is model-specific, so the name of the class will be something like: StanFit4anon_model. The StanFit4model instances expose a number of methods.

class pystan.StanFit4model

    plot (pars=None)
    Visualize samples from posterior distributions
    Parameters
    pars [sequence of str] names of parameters
    This is currently an alias for the traceplot method.

    extract (pars=None, permuted=True, inc_warmup=False, dtypes=None)
    Extract samples in different forms for different parameters.
    Parameters
    pars [sequence of str] names of parameters (including other quantities)
    permuted [bool] If True, returned samples are permuted. All chains are merged and warmup samples are discarded.
    inc_warmup [bool] If True, warmup samples are kept; otherwise they are discarded. If permuted is True, inc_warmup is ignored.
    dtypes [dict] datatype of parameter(s). If nothing is passed, np.float will be used for all parameters.
    Returns
    samples : dict or array If permuted is True, return dictionary with samples for each parameter (or other quantity) named in pars.
    If permuted is False and pars is None, an array is returned. The first dimension of the array is for the iterations; the second for the number of chains; the third for the parameters. Vectors and arrays are expanded to one parameter (a scalar) per cell, with names indicating the third dimension. Parameters are listed in the same order as model_pars and flatnames.
    If permuted is False and pars is not None, return dictionary with samples for each parameter (or other quantity) named in pars. The first dimension of the sample array is for the iterations; the second for the number of chains; the rest for the parameters. Parameters are listed in the same order as pars.

    stansummary (pars=None, probs=(0.025, 0.25, 0.5, 0.75, 0.975), digits_summary=2)
    Summary statistic table. Parameters ———— pars : str or sequence of str, optional
    Parameter names. By default use all parameters
    probs [sequence of float, optional] Quantiles. By default, (0.025, 0.25, 0.5, 0.75, 0.975)
    digits_summary [int, optional] Number of significant digits. By default, 2
    summary [string] Table includes mean, se_mean, sd, probs_0, ..., probs_n, n_eff and Rhat.

    >>> model_code = 'parameters (real y;) model {y ~ normal(0,1);}'
    >>> m = StanModel(model_code=model_code, model_name="example_model")
    >>> fit = m.sampling()
    >>> print(fit.stansummary())
    inference for Stan model: example_model.

(continues on next page)
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

<table>
<thead>
<tr>
<th></th>
<th>mean</th>
<th>se_mean</th>
<th>sd</th>
<th>2.5%</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>97.5%</th>
<th>n_eff</th>
<th>Rhat</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0.01</td>
<td>0.03</td>
<td>1.0</td>
<td>-2.01</td>
<td>-0.68</td>
<td>0.02</td>
<td>0.72</td>
<td>1.97</td>
<td>1330</td>
<td>1.0</td>
</tr>
<tr>
<td>lp__</td>
<td>-0.5</td>
<td>0.02</td>
<td>0.68</td>
<td>-2.44</td>
<td>-0.66</td>
<td>-0.24</td>
<td>-0.05</td>
<td>-5.5e-4</td>
<td>1555</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Samples were drawn using NUTS at Thu Aug 17 00:52:25 2017.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).

**summary** *(pars=None, probs=None)*

Summarize samples (compute mean, SD, quantiles) in all chains. REF: stanfit-class.R summary method

**Parameters**

- **fit**: StanFit4Model object
- **pars**: str or sequence of str, optional
  Parameter names. By default use all parameters
- **probs**: [sequence of float, optional] Quantiles. By default, (0.025, 0.25, 0.5, 0.75, 0.975)

**summaries** [OrderedDict of array] Array indexed by ‘summary’ has dimensions (num_params, num_statistics). Parameters are unraveled in row-major order. Statistics include: mean, se_mean, sd, probs_0. . . , probs_n, n_eff, and Rhat. Array indexed by ‘c_summary’ breaks down the statistics by chain and has dimensions (num_params, num_statistics_c_summary, num_chains). Statistics for c_summary are the same as for summary with the exception that se_mean, n_eff, and Rhat are absent. Row names and column names are also included in the OrderedDict.

**log_prob** *(upar, adjust_transform=True, gradient=False)*

Expose the log_prob of the model to stan_fit so user can call this function.

**Parameters**

- **upar**: The real parameters on the unconstrained space.
- **adjust_transform**: [bool] Whether we add the term due to the transform from constrained space to unconstrained space implicitly done in Stan.

**Note**

In Stan, the parameters need be defined with their supports. For example, for a variance parameter, we must define it on the positive real line. But inside Stan’s sampler, all parameters defined on the constrained space are transformed to unconstrained space, so the log density function need be adjusted (i.e., adding the log of the absolute value of the Jacobian determinant). With the transformation, Stan’s samplers work on the unconstrained space and once a new iteration is drawn, Stan transforms the parameters back to their supports. All the transformation are done inside Stan without interference from the users. However, when using the log density function for a model exposed to Python, we need to be careful. For example, if we are interested in finding the mode of parameters on the constrained space, we then do not need the adjustment. For this reason, there is an argument named adjust_transform for functions log_prob and grad_log_prob.

**grad_log_prob** *(upars, adjust_transform=True)*

Expose the grad_log_prob of the model to stan_fit so user can call this function.

**Parameters**

- **upar**: array] The real parameters on the unconstrained space.
- **adjust_transform**: [bool] Whether we add the term due to the transform from constrained space to unconstrained space implicitly done in Stan.

**get_adaptation_info** ()

Obtain adaptation information for sampler, which now only NUTS2 has.
The results are returned as a list, each element of which is a character string for a chain.

**get_logposterior** *(inc_warmup=True)*
Get the log-posterior (up to an additive constant) for all chains.
Each element of the returned array is the log-posterior for a chain. Optional parameter *inc_warmup* indicates whether to include the warmup period.

**get_sampler_params** *(inc_warmup=True)*
Obtain the parameters used for the sampler such as *steplsize* and *treedepth*. The results are returned as a list, each element of which is an OrderedDict a chain. The dictionary has number of elements corresponding to the number of parameters used in the sampler. Optional parameter *inc_warmup* indicates whether to include the warmup period.

**get_posterior_mean** *
Get the posterior mean for all parameters

Returns

**means** [array of shape (num_parameters, num_chains)] Order of parameters is given by self.model_pars or self.flatnames if parameters of interest include non-scalar parameters. An additional column for mean *lp__* is also included.

**constrain_pars** *(np.ndarray[double, ndim=1, mode="c"] upar not None)*
Transform parameters from unconstrained space to defined support

**unconstrain_pars** *(par)*
Transform parameters from defined support to unconstrained space

**get_seed** *
**get_inits** *
**get_stancode** *
**to_dataframe** *(pars=None, permuted=True, dtypes=None, inc_warmup=False, diagnostics=True)*
Extract samples as a pandas dataframe for different parameters.

**pars** [[str, sequence of str]] parameter (or quantile) name(s). If *permuted* is False, *pars* is ignored.

**permuted** [bool, default False] If True, returned samples are permuted. All chains are merged and warmup samples are discarded.

**dtypes** [dict] datatype of parameter(s). If nothing is passed, np.float will be used for all parameters.

**inc_warmup** [bool] If True, warmup samples are kept; otherwise they are discarded. If *permuted* is True, *inc_warmup* is ignored.

**diagnostics** [bool] If True, include MCMC diagnostics in dataframe. If *permuted* is True, *diagnostics* is ignored.

**df** : pandas dataframe

Unlike default in extract (*permuted=True*) *to_dataframe* method returns non-permuted samples (*permuted=False*) with diagnostics params included.

### 1.11 Conversion utilities for Stan’s R Dump format

<table>
<thead>
<tr>
<th><strong>stan_rdump</strong>(data, filename)</th>
<th>Dump a dictionary with model data into a file using the R dump format that Stan supports.</th>
</tr>
</thead>
</table>
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_rdump</code></td>
<td>Read data formatted using the R dump format</td>
</tr>
<tr>
<td><code>pystan.misc.stan_rdump(data, filename)</code></td>
<td>Dump a dictionary with model data into a file using the R dump format that Stan supports.</td>
</tr>
<tr>
<td></td>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>data (dict)</td>
<td>• data (dict) –</td>
</tr>
<tr>
<td>filename (str)</td>
<td>• filename (str) –</td>
</tr>
<tr>
<td><code>pystan.misc.read_rdump(filename)</code></td>
<td>Read data formatted using the R dump format</td>
</tr>
<tr>
<td></td>
<td><strong>Parameters</strong> filename (str) –</td>
</tr>
<tr>
<td></td>
<td><strong>Returns</strong> data</td>
</tr>
<tr>
<td></td>
<td><strong>Return type</strong> OrderedDict</td>
</tr>
</tbody>
</table>

### 1.12 Logging

PyStan uses logging-module from the Python Standard library to output messages for the user. By default, messages are sent to `sys.stdout`. For more information and usage see logging-module [https://docs.python.org/3.7/library/logging.handlers.html](https://docs.python.org/3.7/library/logging.handlers.html) and logging-cookbook [https://docs.python.org/3/howto/logging-cookbook.html](https://docs.python.org/3/howto/logging-cookbook.html)

- https://docs.python.org/3.7/library/logging.handlers.html
- https://docs.python.org/3/howto/logging-cookbook.html

#### 1.12.1 Advanced users

To add other logger handlers or to redirect all messages from PyStan, the user needs to setup output handlers manually. If the setup is done before importing PyStan, PyStan won’t add automatically `logging.StreamHandler` to logging handlers. Otherwise, PyStan adds `logging.StreamHandler` and other handlers coexist with the default handler.

#### Adding FileHandler

To redirect all messages only to a file.

```python
import logging
logger = logging.getLogger("pystan")

# add root logger (logger Level always Warning)
# not needed if PyStan already imported
logger.addHandler(logging.NullHandler())

logger_path = "pystan.log"
fh = logging.FileHandler(logger_path, encoding="utf-8")
fh.setLevel(logging.INFO)
# optional step
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
logger.addHandler(fh)
```

(continues on next page)
To use both (default, file) logging options import pystan before the setup. In this case PyStan adds root handler, which means that user can skip the root handler step.

```python
import pystan
import logging
logger = logging.getLogger("pystan")
...
```

## 1.13 Threading Support with Pystan 2.18+

Notice! This is an experimental feature and is not tested or supported officially with PyStan 2. Official multithreading support will land with PyStan 3.

By default, *stan-math* is not thread safe. Stan 2.18+ has ability to switch on threading support with compile time arguments.

See https://github.com/stan-dev/math/wiki/Threading-Support

Due to use of *multiprocessing* to parallelize chains, user needs to be aware of the cpu usage. This means that each chain will use *STAN_NUM_THREADS* cpu cores and this can have an affect on performance.

### 1.13.1 Windows

These instructions are invalid on Windows with MingW-W64 compiler and should not be used. Usage will crash the current Python session, which means that no sampling can be done.

see https://github.com/Alexpux/MINGW-packages/issues/2519 and https://sourceforge.net/p/mingw-w64/bugs/445/

### 1.13.2 Example

```python
import pystan
import os
import sys

# set environmental variable STAN_NUM_THREADS
# Use 4 cores per chain
os.environ['STAN_NUM_THREADS'] = "4"

# Example model
# see http://discourse.mc-stan.org/t/cant-make-cmdstan-2-18-in-windows/5088/18
stan_code = ""
functions {
  vector bl_glm(vector mu_sigma, vector beta, 
    real[] x, int[] y) {
    vector[2] mu = mu_sigma[1:2];
    vector[2] sigma = mu_sigma[3:4];
    real lp = normal_lpdf(beta | mu, sigma);
    real ll = bernoulli_logit_lpmf(y | beta[1] + beta[2] * to_vector(x));
  }
}
```
return [lp + ll];
}
}
data {
  int<lower = 0> K;
  int<lower = 0> N;
  vector[N] x;
  int<lower = 0, upper = 1> y[N];
}
transformed data {
  int<lower = 0> J = N / K;
  real x_r[K, J];
  int<lower = 0, upper = 1> x_i[K, J];
  {
    int pos = 1;
    for (k in 1:K) {
      int end = pos + J - 1;
      x_r[k] = to_array_1d(x[pos:end]);
      x_i[k] = y[pos:end];
      pos += J;
    }
  }
}
parameters {
  vector[2] beta[K];
  vector[2] mu;
  vector<lower=0>[2] sigma;
}
model {
  mu ~ normal(0, 2);
  sigma ~ normal(0, 2);
  target += sum(map_rect(bl_glm, append_row(mu, sigma),
                             beta, x_r, x_i));
}
"

stan_data = dict(
  K = 4,
  N = 12,
  x = [1.204, -0.573, -1.35, -1.157,
       -1.29, 0.515, 1.496, 0.918,
       0.517, 1.092, -0.485, -2.157],
  y = [1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1])
extra_compile_args = ['-pthread', '-DSTAN_THREADS']
stan_model = pystan.StanModel(
  model_code=stan_code,
  extra_compile_args=extra_compile_args)

# use the default 4 chains == 4 parallel process
# used cores = min(cpu_cores, 4*STAN_NUM_THREADS)
fit = stan_model.sampling(data=stan_data, n_jobs=4)
print(fit)
1.14 PyStan on Windows

PyStan is partially supported under Windows with the following caveats:

- Python 2.7: Doesn’t support parallel sampling. When drawing samples `n_jobs=1` must be used
- Python 3.5 or higher: Parallel sampling is supported
- MSVC compiler is not supported.

PyStan requires a working C++ compiler. Configuring such a compiler is typically the most challenging step in getting PyStan running.

PyStan is tested against the mingw-w64 compiler which works on both Python versions (2.7, 3.x) and supports x86 and x64.

Due to problems with MSVC template deduction, functions with Eigen library are failing. Until this and other bugs are fixed no support is provided for Windows + MSVC. Currently, no fix is known for this problem, other than to change the compiler to GCC or clang-cl.

### 1.14.1 Installing Python

There several ways of installing PyStan on Windows. The following instructions assume you have installed Python as packaged in the Anaconda Python distribution or Miniconda distribution. The Anaconda distribution is well-maintained and includes packages such as Numpy which PyStan requires. The following instructions assume that you are using Windows 7 (Windows 10 disregards user choice and user privacy).

### 1.14.2 Open Command prompt

All the following commands are written in a command line prompt. You can use one like Anaconda Prompt if you installed Python using Anaconda in the previous step, or Command Prompt which comes with Windows.

Test that the conda package manager is properly installed by typing:

```
conda info
```

To update conda package manager to the latest version:

```
conda update conda
```

### 1.14.3 Create a conda virtual environment (optional)

It is a good practice to keep specific projects on their on conda virtual environments to prevent unnecessary package collisions. Create a new conda environment with:

```
conda create -n stan_env python=3.7
```

where `stan_env` is the name of the environment.

After this activate environment with:

```
conda activate stan_env
```

or if your conda doesn’t include `conda activate use:`

```
conda activate stan_env
```
1.14.4 Installing C++ compiler

There are several ways to install mingw-w64 compiler toolchain, but in these instructions install compiler with conda package manager which comes with the Anaconda package.

To install mingw-w64 compiler type:

```bash
conda install libpython m2w64-toolchain -c msys2
```

This will install

- `libpython` package which is needed to import mingw-w64. <https://anaconda.org/anaconda/libpython>
- `mingw-w64` toolchain. <https://anaconda.org/msys2/m2w64-toolchain>

`libpython` setups automatically `distutils.cfg` file, but if that is failed use the following instructions to setup it manually

In PYTHONPATH\Lib\distutils create a distutils.cfg file with a text editor (e.g. Notepad) and add the following lines:

```
[build]
compiler=mingw32
```

To find the correct distutils path, run the following lines in python:

```python
>>> import distutils
>>> print(distutils.__file__)
```

1.14.5 Install dependencies

It is recommended that on Windows the dependencies are installed with conda and the conda‐forge channel. Required dependencies are numpy and cython:

```bash
conda install numpy cython -c conda‐forge
```

Optional dependencies are matplotlib, scipy and pandas:

```bash
conda install matplotlib scipy pandas -c conda‐forge
```

1.14.6 Installing PyStan

You can install PyStan with either pip (recommended) or conda:

```bash
pip install pystan
conda install pystan -c conda‐forge
```
You can verify that everything was installed successfully by opening up the Python terminal (run `python` from a command prompt) and drawing samples from a very simple model:

```python
>>> import pystan
>>> model_code = 'parameters {real y;} model {y ~ normal(0,1);}'
>>> model = pystan.StanModel(model_code=model_code)
>>> y = model.sampling().extract()['y']
>>> y.mean()  # with luck the result will be near 0
```

## 1.14.7 Steps

With conda:

```bash
conda install numpy cython matplotlib scipy pandas pystan -c conda-forge
```

With pip:

```bash
pip install pystan
```

## 1.15 External C++ (experimental)

This feature is experimental and nothing is guaranteed.

Instructions follow loosely https://dfm.io/posts/stan-c++/ which gives more in-depth example using external C++.

### 1.15.1 Setting up C++

Your C++ code needs to provide needed gradients and partial derivatives to work properly. This can be done by manually implementing each needed component or by using Stan’s autodiff (special syntax). Each function also needs to accept a `std::ostream` as a last argument.

In the following instructions it is assumed that the first code is saved to a file called `external_manual.hpp` and `external_autograd.hpp`. Both files are saved under `cwd` in a directory `external_cpp`.

The following code shows a minimal example with manually implemented gradients for a function $C = A \times B \times B + A$:

```cpp
inline double my_func (double A, double B, std::ostream* pstream) {
  double C = A * B * B + A;
  return C;
}
```

```cpp
inline var my_func (const var& A_var, const var& B_var, std::ostream* pstream) {
  // Compute the value
  double A = A_var.val(),
  B = B_var.val(),
  C = my_func(A, B, pstream);

  // Compute the partial derivatives:
  double dC_dA = B * B + 1.0,
  dC_dB = 2.0 * A * B;
```

(continues on next page)
// Autodiff wrapper:
return var(new precomp_vv_vari(
    C,       // The _value_ of the output
    A_var.vi_,  // The input gradient wrt A
    B_var.vi_,  // The input gradient wrt B
    dC_dA,     // The partial introduced by this function wrt A
    dC_dB      // The partial introduced by this function wrt B
));
}
inline var my_func (double A, const var& B_var, std::ostream* pstream) {
    double B = B_var.val(),
    C = my_func(A, B, pstream),
    dC_dB = 2.0*A*B;
    return var(new precomp_v_vari(C, B_var.vi_, dC_dB));
}
inline var my_func (const var& A_var, double B, std::ostream* pstream) {
    double A = A_var.val(),
    C = my_func(A, B, pstream),
    dC_dA = B*B+1.0;
    return var(new precomp_v_vari(C, A_var.vi_, dC_dA));
}

The minimal example using Stan's autograd for a function $C = A*B*B+A$:

```cpp
template <typename T1, typename T2>
typename boost::math::tools::promote_args<T1, T2>::type
my_other_func (const T1& A, const T2& B, std::ostream* pstream) {
    typedef typename boost::math::tools::promote_args<T1, T2>::type T;
    T C = A*B*B+A;
    return C;
}
```

1.15.2 Setting up Stan

User needs to define functions inside a `functions` block. Function names are defined in the C++ code.

```cpp
functions {
    real my_func(real A, real B);
    real my_other_func(real A, real B);
}
data {
    real B;
}
parameters {
    real A;
    real D;
}
transformed parameters {
    real C = my_func(A, B);
    real E = my_other_func(D, B);
}
```
model {
    C ~ std_normal();
    E ~ std_normal();
}

1.15.3 Setting up Python

In Python user needs information of location for external C++ code. This information is passed to StanModel with two list-objects include_dir and includes. Also allow_undefined keyword is set to True.

```python
import pystan
import os

model_code = ""
    
include_dir = [os.path.join(".", "external_cpp")]
include_files = ["external_manual.hpp", "external_autograd.hpp"]
stan_model = pystan.StanModel(model_code=model_code,
    verbose=True,
    allow_undefined=True,
    includes=include_files,
    include_dirs=[include_dir],
)

fit = stan_model.sampling()
print(fit)
```

Compilation with external C++ can take longer than normally. The external C++ code is injected to the stanc translated C++ code. This injections is done automatically by PyStan before model compilation.

1.16 Unpickling fit without Model instance (experimental)

This feature is experimental and nothing is guaranteed.

In PyStan it is needed to import model instance before unpickling the fit object. This is commonly done by pickling model instance and fit object together in an object that keeps the order (list, tuple, OrderedDict, dict with python 3.6+).

Sometimes mistakes are done, and fit objects are pickled without the model instance, and there is no way to reload the model from other pickled files. Rebuilding the model from Stan-code does not work in this case, because the module name contains random parts that are decided at the runtime. To still get data and samples saved to fit object, PyStan has unpickle_fit function in its experimental submodule. The function will create a fake StanModel instance with the correct module name, which enables one to unpickle fit object without the original model instance. This means, that the model object should not be used to anything which includes the functionality wrapped in the fit object.

The suggested use of this feature is to enable users to extract samples with .extract() method (and access other information stored to fit.sim object). After extraction it is recommended to save samples in another format, such as dict, netCDF4 (arviz.InferenceData) or tabular format (pandas.DataFrame).
1.16.1 Example

Following example will show how to unpickle fit object without model instance and save the resulting samples in other formats.

```python
import pystan
import pystan.experimental

path = "path/to/fit.pickle"
fit = pystan.experimental.unpickle_fit(path)
```

Example on how to save extracted samples to with pickle

```python
import pickle
with open("path/to/fit_dict.pickle", "wb") as f:
    pickle.dump(fit.extract(), protocol=pickle.HIGHEST_PROTOCOL)
```

Example on how to save extracted samples + other information with ArviZ to netCDF4.

```python
import arviz as az
idata = az.from_pystan(fit)
idata.to_netcdf("path/to/fit.nc")
```

Example on how to save extracted samples to csv with pandas

```python
# using pystan.misc.to_dataframe -function works also with the older fit objects
df = pystan.misc.to_dataframe(fit)
df.to_csv("path/to/fit.csv", index=False)
```
• Stan: http://mc-stan.org/
• BUGS Examples
Stan Discourse

- Stan Discourse https://discourse.mc-stan.org/
Important links

- **Source code repo**: https://github.com/stan-dev/pystan
- **HTML documentation**: http://pystan.readthedocs.org
- **Issue tracker**: https://github.com/stan-dev/pystan/issues
CHAPTER 5

Related projects

• ArviZ: Exploratory analysis of Bayesian models with Python by @arviz-devs
• Prophet: Timeseries forecasting by @facebook
• Scikit-learn integration: pystan-sklearn by @rgerkin.
• BAMBI: BAyesian Model-Building Interface by @bambinos
• Jupyter tool: StanMagic by @Arvinds-ds
• Jupyter tool: JupyterStan by @janreyberg
CHAPTER 6

Similar projects

• PyMC: http://pymc-devs.github.io/pymc/
• emcee: http://dan.iel.fm/emcee/current/
The development of PyStan3 with updated API can be found under stan-dev/pystan-next
CHAPTER 8

Indices and tables

- genindex
- modindex
- search
p
pystan, 21
Index

C
constrain_pars() (pystan.StanFit4model method), 35

E
extract() (pystan.StanFit4model method), 33

G
get_adaptation_info() (pystan.StanFit4model method), 34
get_cppcode() (pystan.StanModel method), 27
get_cxxflags() (pystan.StanModel method), 27
get_include_paths() (pystan.StanModel method), 27
get_inits() (pystan.StanFit4model method), 35
get_logposterior() (pystan.StanFit4model method), 35
get_posterior_mean() (pystan.StanFit4model method), 35
get_sampler_params() (pystan.StanFit4model method), 35
get_seed() (pystan.StanFit4model method), 35
get_stancode() (pystan.StanFit4model method), 35
grad_log_prob() (pystan.StanFit4model method), 34

L
log_prob() (pystan.StanFit4model method), 34

M
model_code (pystan.StanModel attribute), 27
model_cpp (pystan.StanModel attribute), 27
model_name (pystan.StanModel attribute), 27
module (pystan.StanModel attribute), 27

O
optimizing() (pystan.StanModel method), 27, 28

P
plot() (pystan.StanFit4model method), 33

pystan (module), 21

R
read_rdump() (in module pystan.misc), 36

S
sampling() (pystan.StanModel method), 27, 29
show() (pystan.StanModel method), 27
stan() (in module pystan), 21
stan_rdump() (in module pystan.misc), 36
stanc() (in module pystan), 24
StanFit4model (class in pystan), 33
StanModel (class in pystan), 26
stansummary() (pystan.StanFit4model method), 33
summary() (pystan.StanFit4model method), 34

to_dataframe() (pystan.StanFit4model method), 35

U
unconstrain_pars() (pystan.StanFit4model method), 35

V
vb() (pystan.StanModel method), 31