API

`stan`([file, model_name, model_code, fit, ...])	Fit a model using Stan.
`stanc`([file, charset, model_code, ...])	Translate Stan model specification into C++ coc
`StanModel`([file, charset, model_name, ...])	Model described in Stan's modeling language cc

StanFit4model instances are also documented on this page.

`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, save_dso=True, verbose=False, boost_lib=None, eigen_lib=None, n_jobs=-1, **kwargs)*

Fit a model using Stan.

Parameters:

file : string {'filename', file-like object}

Model code must 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 burin) 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

> *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*.

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) 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.

save_dso : boolean, optional

Indicates whether the dynamic shared object (DSO) compiled from C++ code will be saved for use in a future Python session. True by default.

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.

Returns: **fit** : 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 intial values are simulated from [-*init_r*, *init_r*] rather than using the default interval (see the manual of Stan).

test_grad: bool, optional :

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)*.

Examples

```
>>> 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[4] <- -0.3742; y[5]  <- 0.3885;   y[6] <- -2.1585;
...     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', verbose=False, obfuscate_model_name=True)*

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.

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.

Returns:

stanc_ret : dict

> 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).

See also

`StanModel`

Class representing a compiled Stan model

`stan`

Fit a model using Stan

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.

References

The Stan Development Team (2013) *Stan Modeling Language User's Guide and Reference Manual*. <http://mc-stan.org/>.

Examples

```
>>> stanmodelcode = '''
... data {
...   int<lower=0> N;
...   real y[N];
... }
...
... parameters {
...   real mu;
... }
...
... model {
...   mu ~ normal(0, 10);
...   y ~ normal(mu, 1);
... }
... '''
>>> r = stanc(model_code=stanmodelcode, model_name = "normal1")
>>> sorted(r.keys())
['cppcode', 'model_code', 'model_cppname', 'model_name', 'status']
>>> r['model_name']
'normal1'
```

class **pystan.StanModel**(*file=None, charset='utf-8', model_name='anon_model', model_code=None, stanc_ret=None, boost_lib=None, eigen_lib=None, save_dso=True, verbose=False, obfuscate_model_name=True*)

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*.

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.

save_dso : boolean, True by default

> Indicates whether the dynamic shared object (DSO) compiled from C++ code will be saved for use in a future Python session.

verbose : boolean, False by default

> Indicates whether intermediate output should be piped to the console. This output may be useful for debugging.

kwargs : keyword arguments

> Additional arguments passed to *stanc*.

See also

> `stanc`

Compile a Stan model specification

> `stan`

Fit a model using Stan

Notes

Instances of StanModel can be saved for use across Python sessions only if *save_dso* is set to True during the construction of StanModel objects.

Even if *save_dso* is True, models cannot be loaded on platforms that differ from the one on which the model was compiled.

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*.

References

The Stan Development Team (2013) *Stan Modeling Language User's Guide and Reference Manual*. <URL: http://mc-stan.org/>.

Examples

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

Attributes

model_name	(string)

model_code	(string) Stan code for the model.
model_cpp	(string) C++ code for the model.
dso	(builtins.module) Python module created by compiling the C++ code for the

Methods

`sampling`([data, pars, chains, iter, warmup, ...])	Draw samples from the model.
`optimizing`([data, seed, init, sample_file, ...])	Obtain a point estimate by maximizing th

show	Print the Stan model specification.
get_cppcode	Return the C++ code for the module.
get_cxxflags	Return the 'CXXFLAGS' used for compiling the 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** : OrderedDict

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’].

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, see Stan manual. Default is 0.001

tol_obj : float, optional

For BFGS and LBFGS, see Stan manual. Default is 1e-12.

tol_grad : float, optional

For BFGS and LBFGS, see Stan manual. Default is 1e-8.

tol_param : float, optional

For BFGS and LBFGS, see Stan manual. Default is 1e-8.

tol_rel_grad : float, optional

For BFGS and LBFGS, see Stan manual. Default is 1e4.

tol_rel_param : float, optional

For BFGS and LBFGS, see Stan manual. Default is 1e7.

history_size : int, optional

For LBFGS, see Stan manual. Default is 5.

Examples

```
>>> 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 burin) iterations. As *warmup* also specifies the number of iterations used for step-size adaption, warmup samples should not be used for inference.

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`.

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.

diagnostic_file : str, optional

File name indicating where diagonstic data for all parameters should be written. If not writable, a temporary directory is used.

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, 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.

Returns:

fit : StanFit4<model_name>

Instance containing the fitted results.

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 intial values are simulated from [-*init_r*, *init_r*] rather than using the default interval (see the manual of Stan).

test_grad: bool, optional :

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).*

Examples

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

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*)

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.

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, 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*.

`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 *stepsize* 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.

`unconstrain_pars`(*par*)

Transform parameters from defined support to unconstrained space

`get_seed()`

`get_inits()`

`get_stancode()`