

---

# Yaps Documentation

*Release 0.1.4*

**Guillaume Baudart, Martin Hirzel, Kiran Kate, Louis Mandel, Avrah**

**Jan 11, 2019**



---

# Contents

---

<b>1</b>	<b>YAPS</b>	<b>3</b>
1.1	Install . . . . .	4
1.2	Tools . . . . .	5
1.3	Documentation . . . . .	5
1.4	License . . . . .	5
1.5	Contributions . . . . .	6
<b>2</b>	<b>Yaps Modeling Language</b>	<b>7</b>
2.1	Comments . . . . .	7
2.2	Data Types and Variable Declarations . . . . .	7
2.3	Expressions . . . . .	8
2.4	Statements . . . . .	8
2.5	Program Blocks . . . . .	9
2.6	Function Definitions . . . . .	9
<b>3</b>	<b>Inference</b>	<b>11</b>
3.1	PyStan Wrapper . . . . .	11
3.2	Direct API use . . . . .	12
<b>4</b>	<b>For Developers</b>	<b>13</b>
4.1	Documentation . . . . .	13
4.2	Distribution . . . . .	13
<b>5</b>	<b>Links</b>	<b>15</b>



Yaps is a new surface language for programming Stan models using python syntax.



Yaps is a new surface language for Stan. It lets users write Stan programs using Python syntax. For example, consider the following Stan program, which models tosses  $x$  of a coin with bias  $\theta$ :

```
data {  
  int<lower=0,upper=1> x[10];  
}  
parameters {  
  real<lower=0,upper=1> theta;  
}  
model {  
  theta ~ uniform(0,1);  
  for (i in 1:10)  
    x[i] ~ bernoulli(theta);  
}
```

It can be rewritten in Python as follows:

```
import yaps  
from yaps.lib import int, real, uniform, bernoulli  
  
@yaps.model  
def coin(x: int(lower=0, upper=1)[10]):  
    theta: real(lower=0, upper=1) <~ uniform(0, 1)  
    for i in range(1,11):  
        x[i] <~ bernoulli(theta)
```

The `@yaps.model` decorator indicates that the function following it is a Stan program. While being syntactically Python, it is semantically reinterpreted as Stan.

The argument of the function corresponds to the `data` block. The type of the data must be declared. Here, you can see that  $x$  is an array of 10 integers between 0 and 1 (`int(lower=0, upper=1)[10]`).

Parameters are declared as variables with their type in the body of the function. Their prior can be defined using the sampling operator `<~` (or `is`).

The body of the function corresponds to the Stan model. Python syntax is used for the imperative constructs of the model, like the `for` loop in the example. The operator `<~` is used to represent sampling and `x.T[a, b]` for truncated distribution.

Note that Stan array are 1-based. The range of the loop is thus `range(1, 11)`, that is 1,2, ... 10.

Other Stan blocks can be introduced using the `with` syntax of Python. For example, the previous program could also be written as follows:

```
@yaps.model
def coin(x: int(lower=0, upper=1)[10]):
    with parameters:
        theta: real(lower=0, upper=1)
    with model:
        theta <~ uniform(0, 1)
        for i in range(1,11):
            x[i] <~ bernoulli(theta)
```

The corresponding Stan program can be displayed using the `print` function:

```
print(coin)
```

Finally, it is possible to launch Bayesian inference on the defined model applied to some data. The communication with the Stan inference engine is based on `PyCmdStan`.

```
flips = np.array([0, 1, 0, 0, 0, 0, 0, 0, 0, 1])
constrained_coin = coin(x=flips)
constrained_coin.sample(data=constrained_coin.data)
```

Note that arrays must be cast into numpy arrays (see `pymcmdstan` documentation).

After the inference the attribute `posterior` of the constrained model is an object with fields for the latent model parameters:

```
theta_mean = constrained_coin.posterior.theta.mean()
print("mean of theta: {:.3f}".format(theta_mean))
```

Yaps provides a lighter syntax to Stan programs. Since Yaps uses Python syntax, users can take advantage of Python tooling for syntax highlighting, indentation, error reporting, ...

## 1.1 Install

Yaps depends on the following python packages:

- astor
- graphviz
- antlr4-python3-runtime
- pycmdstan

To install Yaps and all its dependencies run:

```
pip install yaps
```

To install from source, first clone the repo, then:



```
pip install .
```

By default, communication with the Stan inference engine is based on `PyCmdStan`. To run inference, you first need to install `CmdStan` and set the `CMDSTAN` environment variable to point to your `CmdStan` directory.

```
export CMDSTAN=/path/to/cmdstan
```

## 1.2 Tools

We provide a tool to compile Stan files to Yaps syntax. For instance, if `path/to/coin.stan` contain the Stan model presented at the beginning, then:

```
stan2yaps path/to/coin.stan
```

outputs:

```
# -----
# tests/stan/coin.stan
# -----
@yaps.model
def stan_model(x: int(lower=0, upper=1)[10]):
    theta: real
    theta is uniform(0.0, 1.0)
    for i in range(1, 10 + 1):
        x[(i),] is bernoulli(theta)
    print(x)
```

Compilers from Yaps to Stan and from Stan to Yaps can also be invoked programmatically using the following functions:

```
yaps.from_stan(code_string=None, code_file=None) # Compile a Stan model to Yaps
yaps.to_stan(code_string=None, code_file=None) # Compile a Yaps model to Stan
```

## 1.3 Documentation

The full documentation is available at <https://yaps.readthedocs.io>. You can find more details in the following [article](#):

```
@article{2018-yaps-stan,
  author = {Baudart, Guillaume and Hirzel, Martin and Kate, Kiran and Mandel, Louis_
↪and Shinnar, Avraham},
  title = "{Yaps: Python Frontend to Stan}",
  journal = {arXiv e-prints},
  year = 2018,
  month = Dec,
  url = {https://arxiv.org/abs/1812.04125},
}
```

## 1.4 License

Yaps is distributed under the terms of the Apache 2.0 License, see [LICENSE.txt](#)

## 1.5 Contributions

Yaps is still at an early phase of development and we welcome contributions. Contributors are expected to submit a 'Developer's Certificate of Origin', which can be found in [DCO1.1.txt](#).

---

## Yaps Modeling Language

---

A Yaps model is a Python function prefixed by the `@yaps.model` decorator.

```
import yaps
from yaps.lib import int, real, uniform, bernoulli

@yaps.model
def coin(x: int(lower=0, upper=1)[10]):
    theta: real(lower=0, upper=1) <~ uniform(0, 1)
    for i in range(10):
        x[i] <~ bernoulli(theta)
```

Types definitions, e.g., `int` and `real`, and Stan functions are defined in `yaps.lib`.

Below are examples of Yaps code with the corresponding Stan code.

### 2.1 Comments

```
# This is a comment
x <~ Normal(0,1) # This is a comment
```

### 2.2 Data Types and Variable Declarations

# Yaps	# Stan
#####	#####
x: int	# int x;
x: real	# real x;
x: real[10]	# real x[10];
m: matrix[6,7] [3,3]	# matrix[3,3] m[6,7];

(continues on next page)

(continued from previous page)

```

N: int(lower=1)           # int<lower=1> N;
log_p: real(upper=0)     # real<upper=0> log_p;
rho: vector(lower=-1,upper=1)[3] # vector<lower=-1,upper=1>[3] rho;

mu: vector[7][3]        # vector[7] mu[3];
mu: matrix[7,2] [15,12] # matrix[7,2] mu[15,12];

x = w[5]                # x = w[5];
c = a[1,3]              # c = a[1,3];
a: matrix[3,2] = 0.5 * (b + c) # matrix[3,2] a = 0.5 * (b + c);

```

## 2.3 Expressions

```

m1: matrix[3,2] = [[1,2],[3,4],[5,6]] # matrix[3,2] m1 = [[1,2],[3,4],[5,6]];
vX: vector[2] = [1,10].transpose     # vector[2] vX = [1,10]';
a: int[3] = {1,10,1000}              # int a[3] = {1,10,1000};
b: int[2,3] = {{1,2,3},{4,5,6}}      # int b[2,3] = {{1,2,3},{4,5,6}};

3.0+0.14
-15
2*3+1
(x-y)/2.0
(n*(n+1))/2
x/n
m%n

3**2                                # 3^2
c = a.pmult(b)                      # c = a .* b
c = a.pdiv(b)                       # c = a ./ b
b if a else c                        # a?b:c

x[4]
x[4,:]                               # x[4,] or x[4,:]

```

## 2.4 Statements

```

target += -0.5 * y * y              # target += -0.5 * y * y;
y <~ normal(mu, sigma)              # y ~ normal(mu, sigma);
y is normal(mu, sigma)              # y ~ normal(mu, sigma);
y <~ normal(0,1).T[-0.5, 2.1]      # y ~ normal(0, 1) T[-0.5, 2.1];

for n in range(1,N+1): ...          # for (n in 1:N) {...}
while cond: ...                     # while (cond) {...}
if cond: ...                         # if (cond) {...}
else: ...                            # else {...}

break                                # break;
continue                             # continue;
pass                                 # //nothing

```

(continues on next page)

(continued from previous page)

```
with block:                                # {
    ...                                    #   ...
                                           # }
```

Warning: `range(n)` in python denotes integers from 0 to n-1. In Stan indexes starts from 1 (for `i in 1:n`). The correct translation for `for i in 1:n` is thus `for i in range(1, n+1)`.

## 2.5 Program Blocks

- The keyword arguments of the Yaps model function are Stan data.
- Yaps top-level declarations are parsed as Stan parameters.
- Yaps top-level statements define the Stan model.

```
def model(x: real):                          # data {int x;}
    mu: real                                  # parameters {real mu;}
    x <~ normal(mu, 1)                        # model { x ~ normal(mu, 1)}
```

Yaps also supports a fully annotated syntax where blocks are introduced via python `with` statements

```
with functions: ...                          # function {...}
with transformed_data                          # transformed data {...}
with parameters: ...                          # parameters {...}
with transformed_parameters: ...              # transformed parameters {...}
with model: ...                                # model {...}
with generated quantities: ...                 # generated quantities {...}
```

## 2.6 Function Definitions

User-defined functions must be defined inside the model in the `functions` block. Their syntax follows Python syntax with type annotations

```
with functions:                               # funtions {
    def successor(x: int) -> int:             #   int successor(int x) {
        return x + 1                          #       return x + 1;
                                               #   }
                                               # }
```



By default, communication with the Stan inference engine is based on `PyCmdStan`. A constrained model can be defined by passing concrete values for the data. This constrained model is linked to a `PyCmdStan` model. It is thus possible to invoke the `pymcmdstan` methods `sample`, `run`, `optimize`, or `variational` to launch the inference. After the inference, the result is stored in the `posterior` attribute of the constrained model as an object with one field for each learned parameter.

For example:

```
@yaps.model
def coin(x: int(lower=0, upper=1)[10]):
    theta: real(lower=0, upper=1) is uniform(0, 1)
    for i in range(2, 11):
        x[i] is bernoulli(theta)

flips = np.array([0, 1, 0, 0, 0, 0, 0, 0, 0, 1])

constrained_coin = coin(x=flips)
constrained_coin.sample(data=constrained_coin.data)
theta_mean = constrained_coin.posterior.theta.mean()
print("mean of theta: {:.3f}".format(theta_mean))
```

Errors detected by the Stan compiler and runtime are mapped to the original yaps code.

Note that this interface takes full advantage of the features offered by `PyCmdStan`. In particular, models are cached and only recompiled when a change is detected even if the rest of the python script has changed.

## 3.1 PyStan Wrapper

Yaps also offer a limited wrapper for the `PyStan` interface. For instance, the inference part of the previous example can be rewritten:

```
fit = yaps.apply(pystan.stan, constrained_coin)
theta_mean = fit.extract()['theta'].mean()
print("mean of theta: {:.3f}".format(theta_mean))
```

The wrapper is used to map the errors back to the original yaps code.

## 3.2 Direct API use

Finally it is possible to use yaps only as a compiler and rely on the existing API for PyCmdStan or PyStan. For every decorated yaps model `model`, the string `str(model)` contains the compiled Stan code.

Using PyCmdStan the previous example becomes:

```
coin_dat = {'x': np.array([1,0,1,0,1,0,0,0,0,1])}
coin_model = pycmdstan.Model(code = str(coin))
fit = coin_model.sample(data = coin_dat)
theta_mean = fit.csv['theta'].mean()
print("mean of theta: {:.3f}".format(theta_mean))
```

And using PyStan

```
coin_dat = {'x': np.array([1,0,1,0,1,0,0,0,0,1])}
fit = pystan.stan(model_code=str(coin), data=coin_dat)
theta_mean = fit.extract(permutated=True)['theta'].mean()
print("mean of theta: {:.3f}".format(theta_mean))
```



# CHAPTER 4

---

## For Developers

---

To build the parser, you need to install `antlr4` before installing the package. To test your model with the Stan inference engine, you need to install `cmdstan`. Then install the dependencies.

```
pip install nose astor graphviz antlr4-python3-runtime pycmdstan
make
export CMDSTAN='path/to/cmdstan-dir'
make test
```

To test the round trip on only one file, after the install:

```
yaps-roundtrip path/to/file.stan
```

## 4.1 Documentation

The documentation is hosted by [ReadTheDocs](#). To keep the README in sync with the doc:

```
make doc
```

## 4.2 Distribution

To create a new distribution you need the following packages:

```
pip install setuptools wheel twine
```

Then to build the new distribution and upload it:

```
make distrib
make upload
```

Note: you need valid PyPI credentials to upload the package.



## CHAPTER 5

---

### Links

---

- Stan: <http://mc-stan.org/>
- PyStan: <https://pystan.readthedocs.io>
- PyCmdStan: <https://pycmdstan.readthedocs.io>