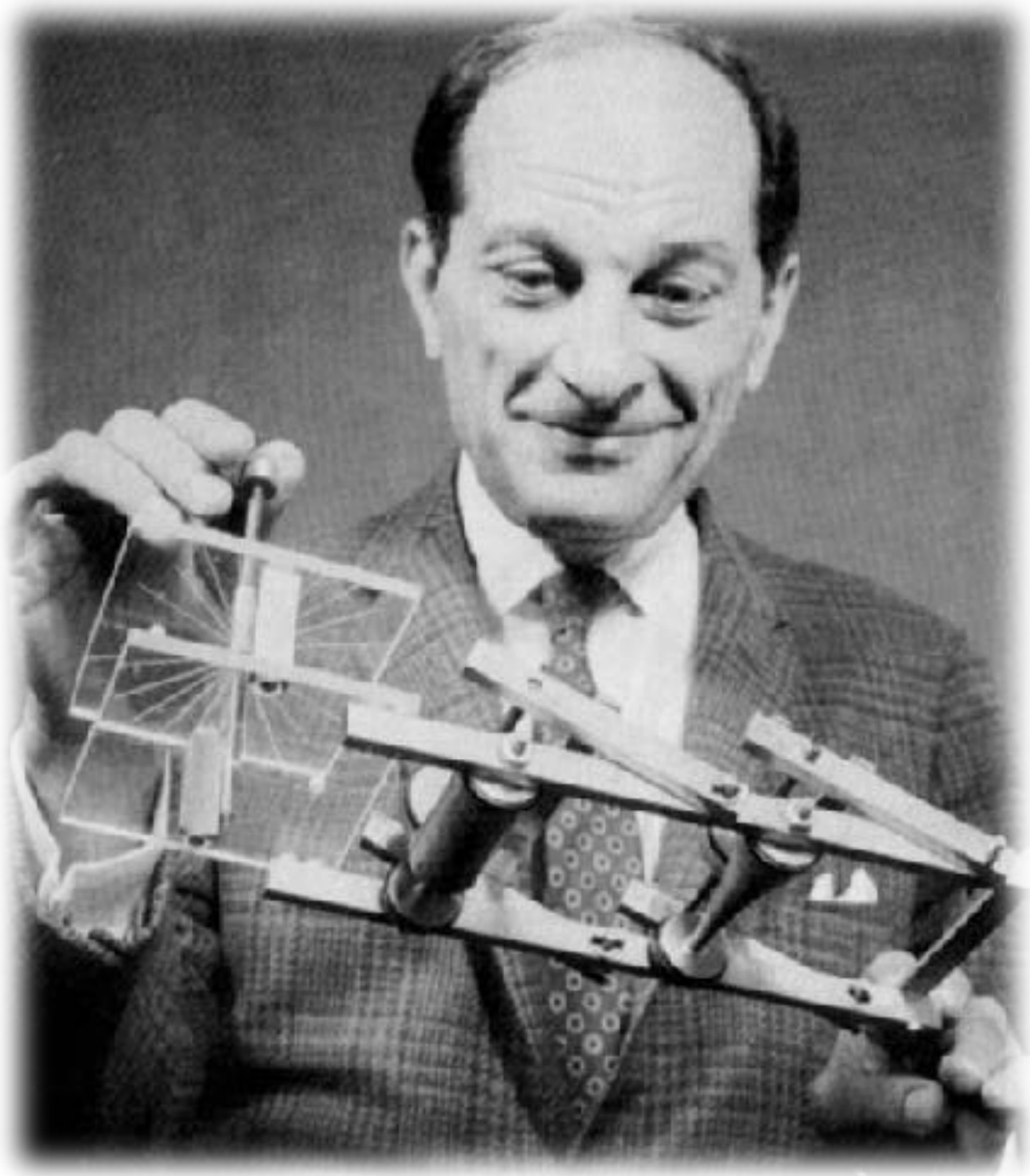
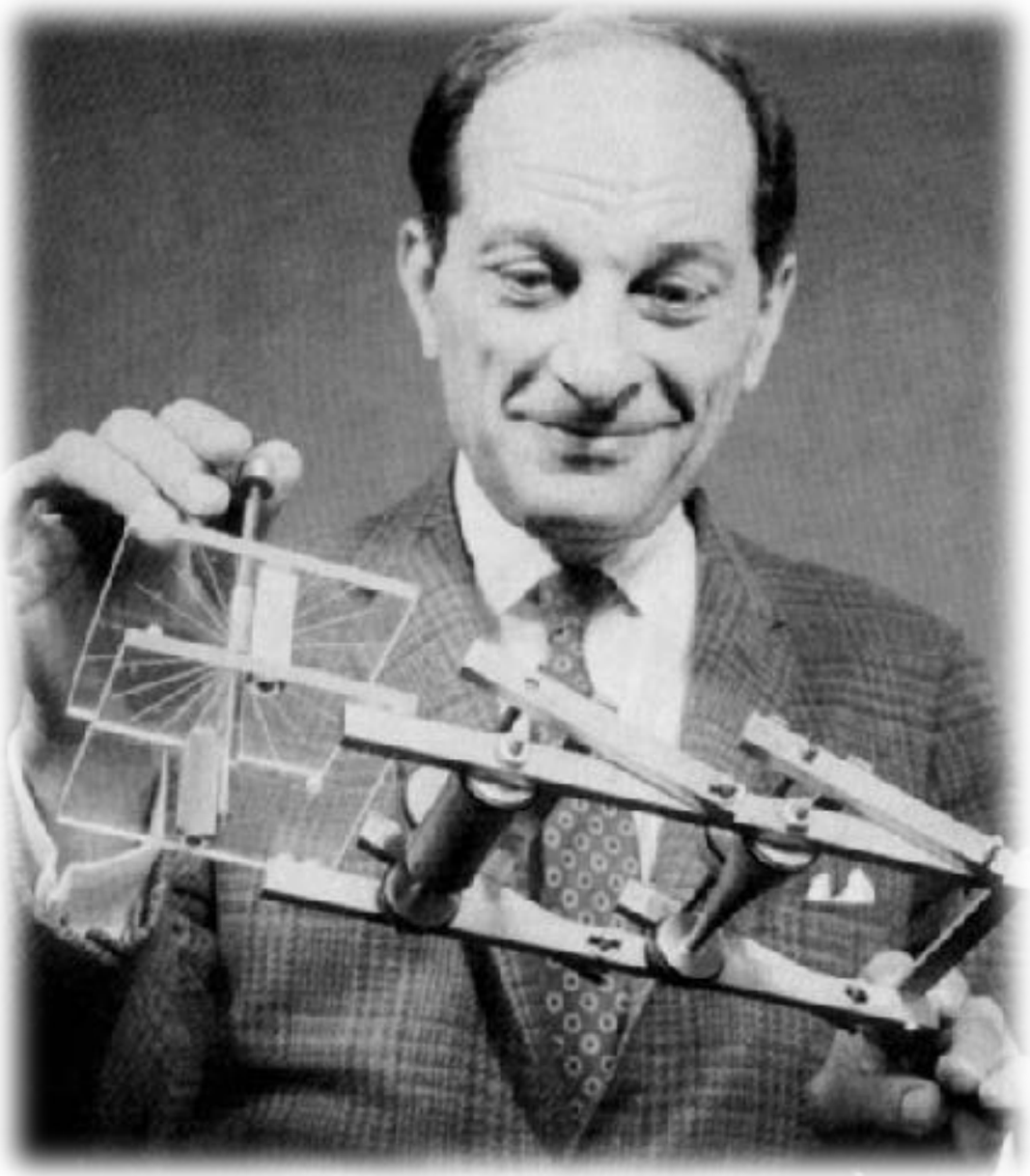
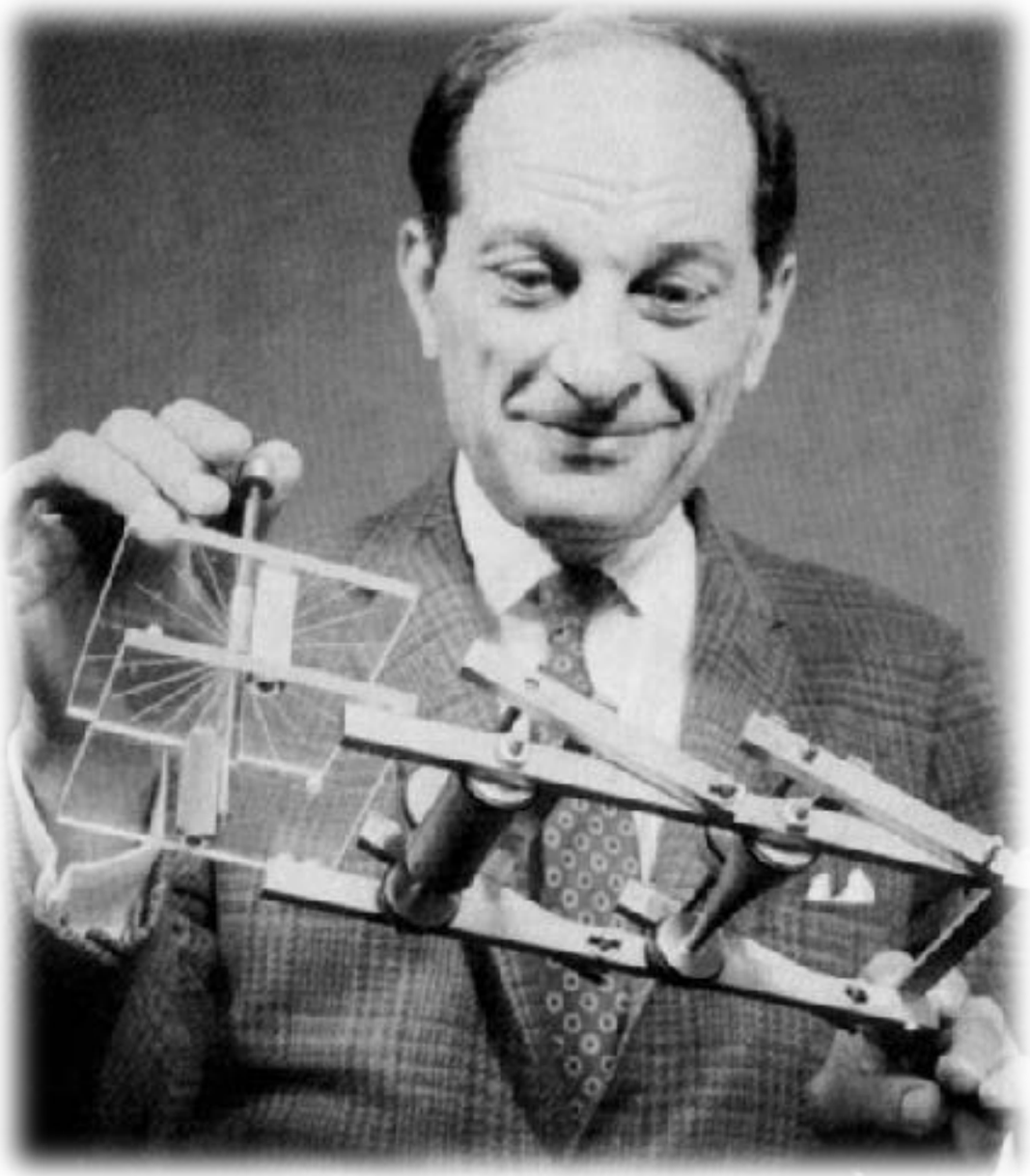
Stan

Software Ecosystem for Modern Bayesian Inference



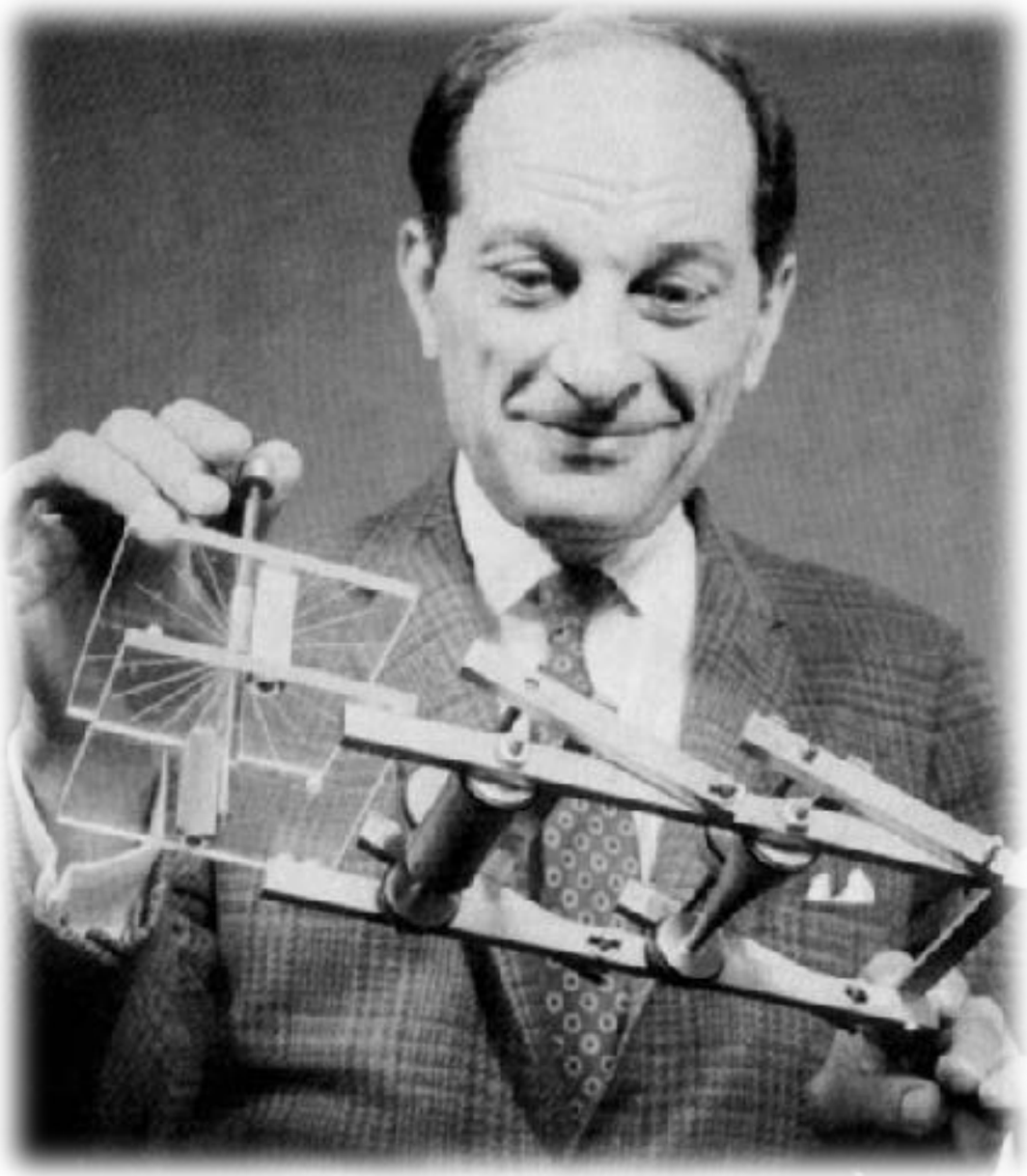


Stanislaw Ulam
(1909–1984)



Stanislaw Ulam
(1909–1984)

Monte Carlo
Method



Stanislaw Ulam
(1909–1984)

H-Bomb

Monte Carlo
Method

What is Stan?

What is Stan?

- Probabilistic **programming language** and **inference algorithms**

What is Stan?

- Probabilistic **programming language** and **inference algorithms**
- Stan **program**
  - declares data and (constrained) parameter variables
  - defines log posterior (or penalized likelihood)

What is Stan?

- Probabilistic **programming language** and **inference algorithms**
- Stan **program**
  - declares data and (constrained) parameter variables
  - defines log posterior (or penalized likelihood)
- Stan **inference**
  - MCMC for full Bayes
  - VB for approximate Bayes
  - Optimization for (penalized) MLE

What is Stan?

- Probabilistic **programming language** and **inference algorithms**
- Stan **program**
  - declares data and (constrained) parameter variables
  - defines log posterior (or penalized likelihood)
- Stan **inference**
  - MCMC for full Bayes
  - VB for approximate Bayes
  - Optimization for (penalized) MLE
- Stan **ecosystem**
  - lang, math library (C++)
  - interfaces and tools (**R**, Python, Julia, many more)
  - documentation (example model repo, user guide & reference manual, case studies, R package vignettes)
  - online community (Stan Forums on discourse)

Users

statistical analysis with Stan

Interfaces and Installation

In order to facilitate inference, Stan provides both a modeling language for specifying complex statistical models and a library of statistical algorithms for computing inferences with those models. These components are exposed through interfaces in environments such as R, Python, and the command line.

To get started with Stan head over to our interfaces page where you will find documentation and installation instructions about each of our interfaces:

- Stan Interfaces

Documentation, Tutorials, and Case Studies

Already playing with Stan in your favorite computing environment? Then it's time to put Stan to use and fit some models! For information on the Stan modeling language as well as tutorials, case studies, and examples, head over to our documentation page,

- Stan Documentation

Why Stan?

Why Stan?

- Fit rich Bayesian statistical models

Why Stan?

- Fit rich Bayesian statistical models
- Efficiency
  - HMC + NUTS
  - Compiled to C++

Why Stan?

- Fit rich Bayesian statistical models
- Efficiency
  - HMC + NUTS
  - Compiled to C++
- Flexible domain specific language

Why Stan?

- Fit rich Bayesian statistical models
- Efficiency
  - HMC + NUTS
  - Compiled to C++
- Flexible domain specific language
- "Freedom-respecting, open-source"
  - BSD (code)
  - CC-BY (doc & written materials)

Who is using Stan?

Who is using Stan?

Biological & physical sciences

Who is using Stan?

Biological & physical sciences

- clinical trials
- epidemiology
- genomics
- population ecology
- entomology
- ophthalmology
- neurology
- agriculture
- fisheries
- cancer biology
- astrophysics & cosmology
- molecular biology
- oceanography
- climatology

Who is using Stan?

Social Sciences

Who is using Stan?

Social Sciences

- population dynamics
- psycholinguistics
- social networks
- political science
- human development
- economics

Who is using Stan?

Many more…

Who is using Stan?

Many more…

- sports analytics
- public health
- publishing
- finance
- pharma
- actuarial
- recommender systems
- educational testing
- materials engineering

Who is using Stan?

mc-stan.org/users/citations

github.com/stan-dev/stancon_talks

Stan is many things

Stan is many things

```
Math
```

Stan is many things

Math ⟵ Language

Stan is many things

Math ⟵ **Language** ⟵ **Algorithms**

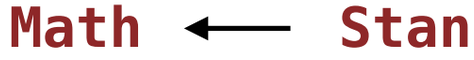
Stan is many things

Math ⟵ **Language** ⟵ **Algorithms** ⟵ **Services**

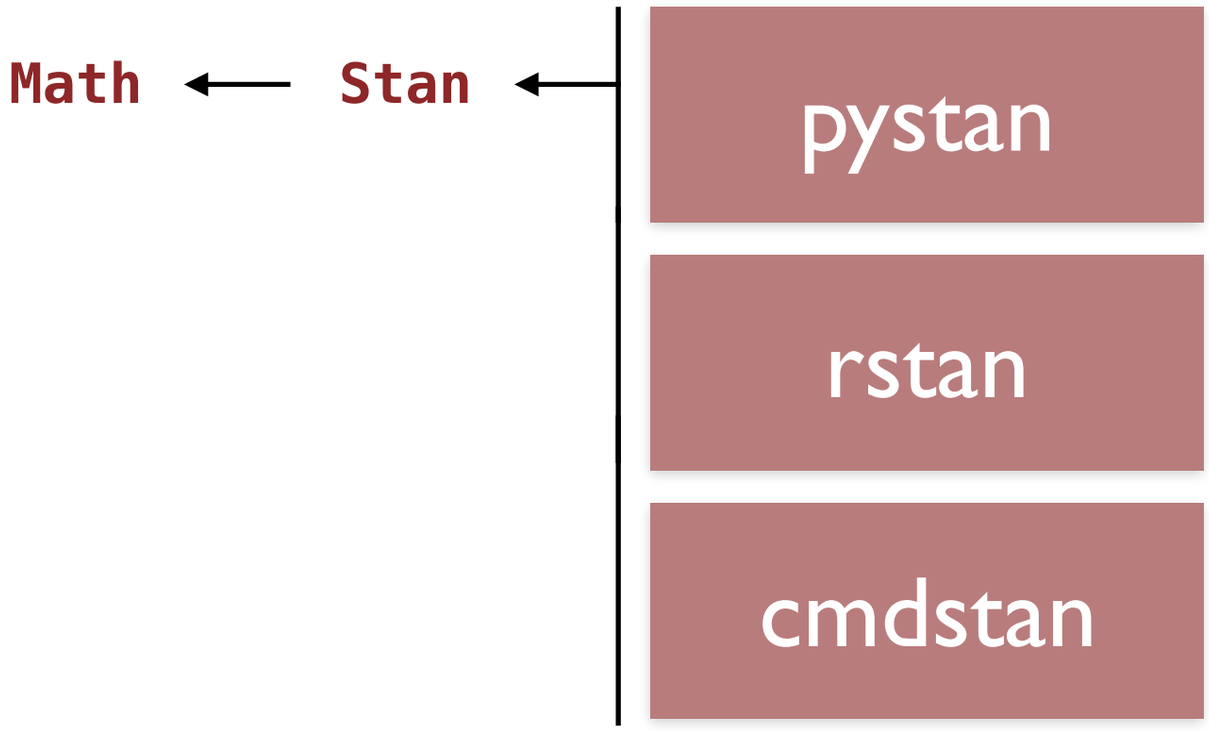
Stan is many things

Math ⟵ Stan

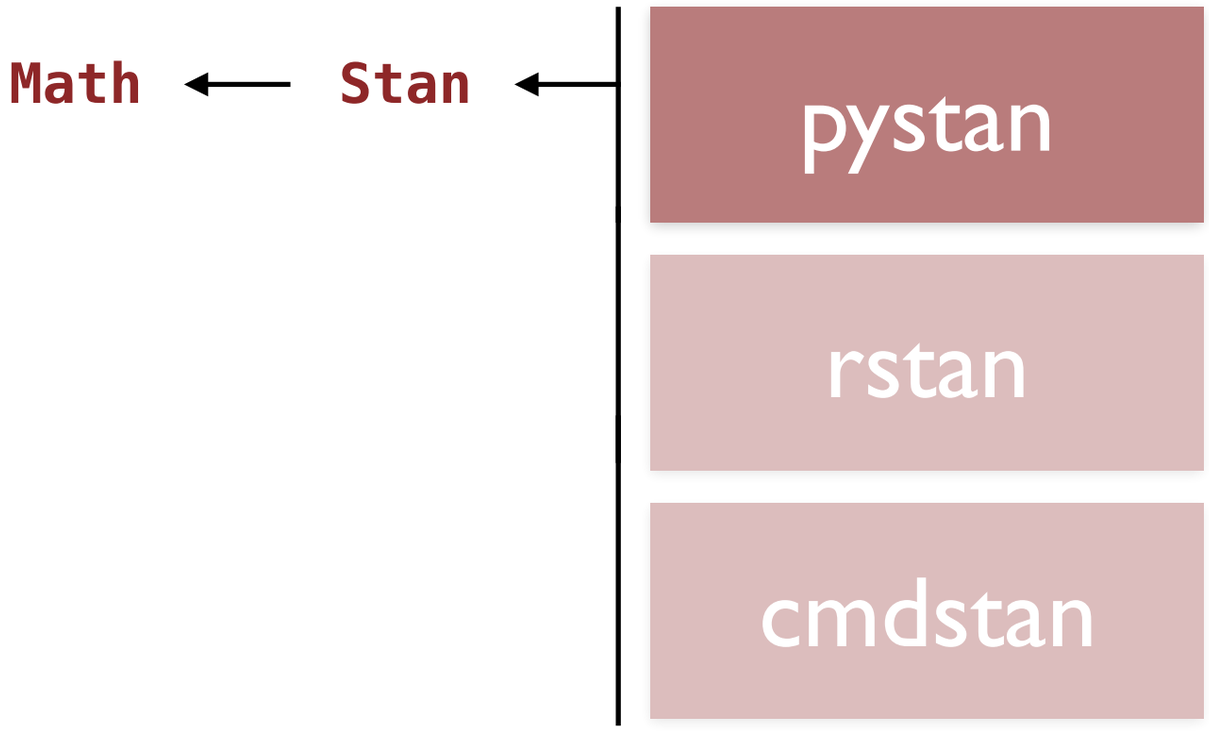
Interfaces



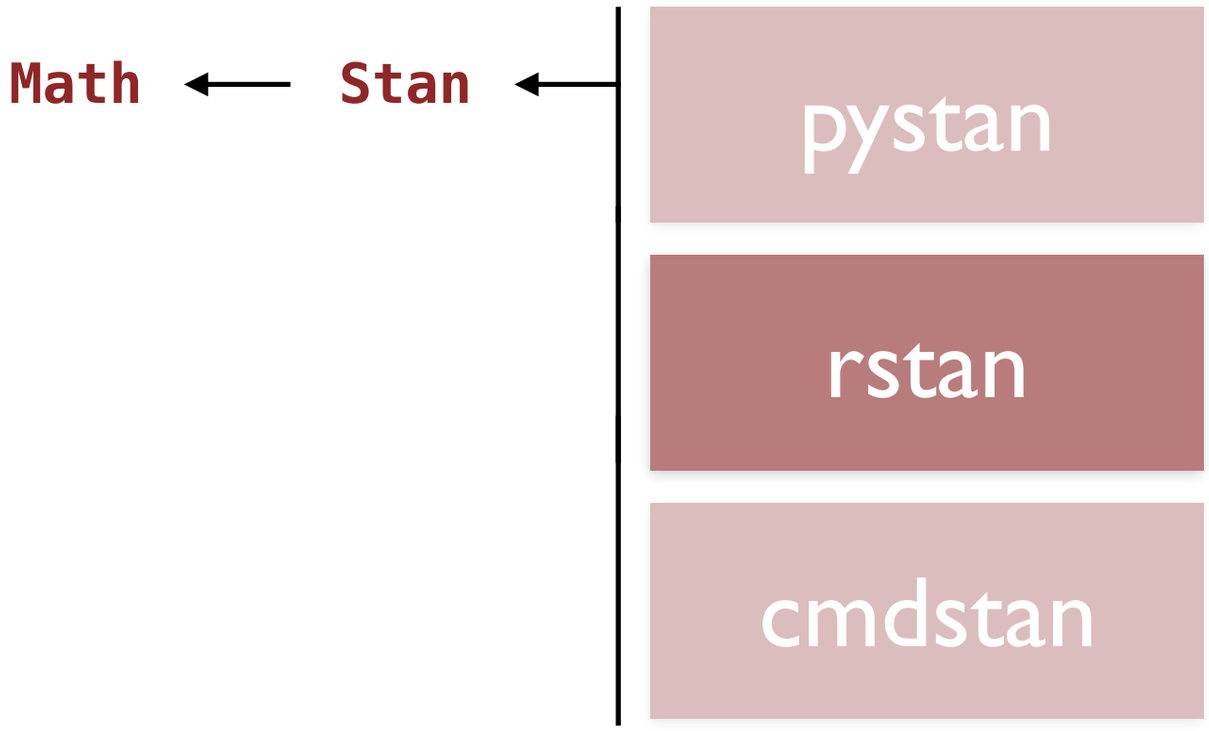
Interfaces



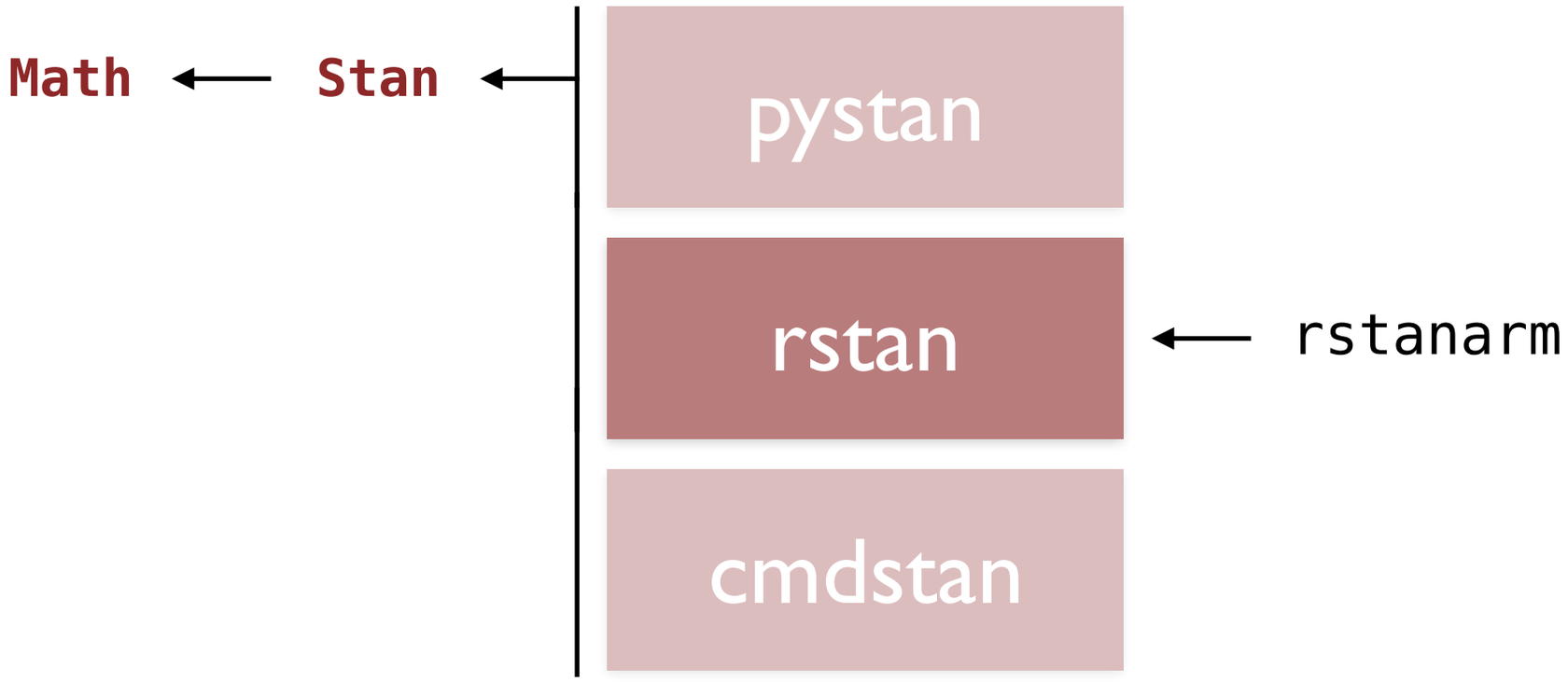
Interfaces



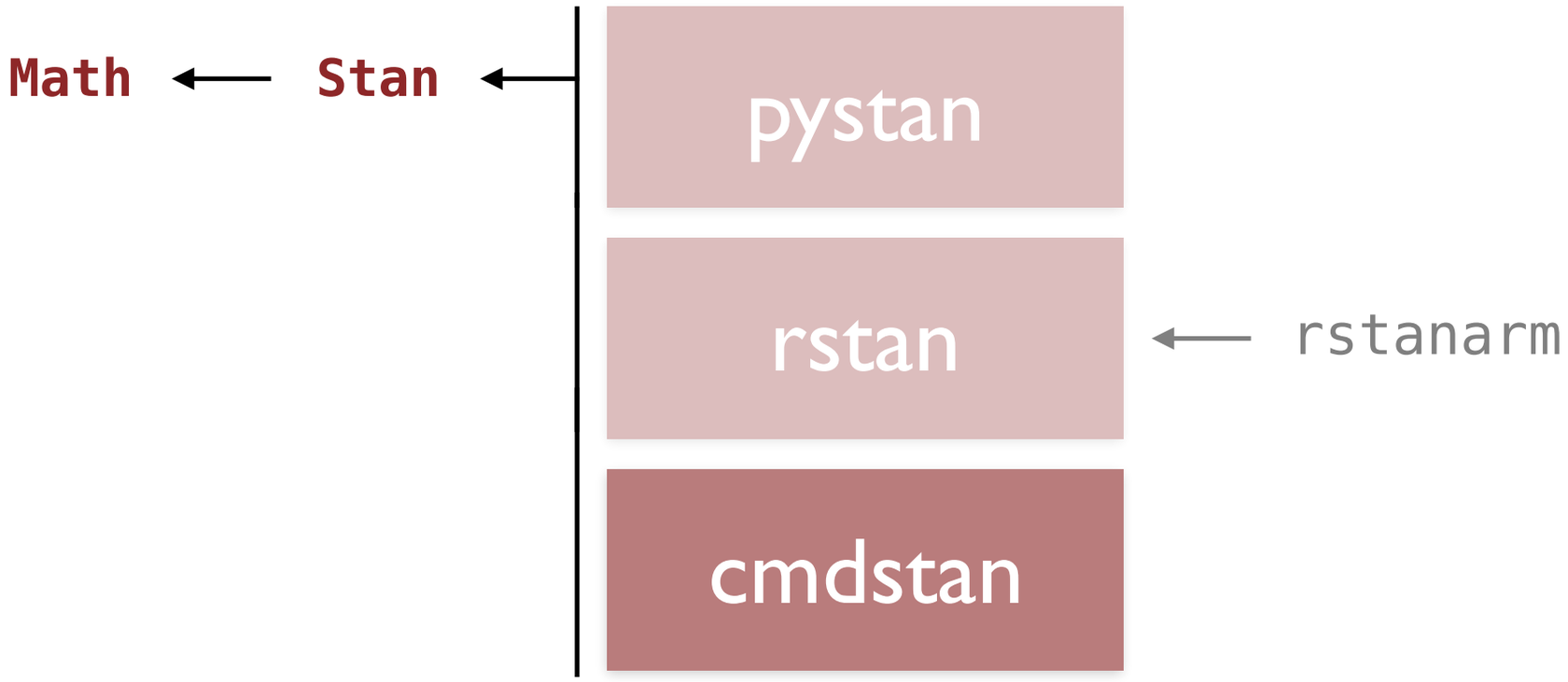
Interfaces



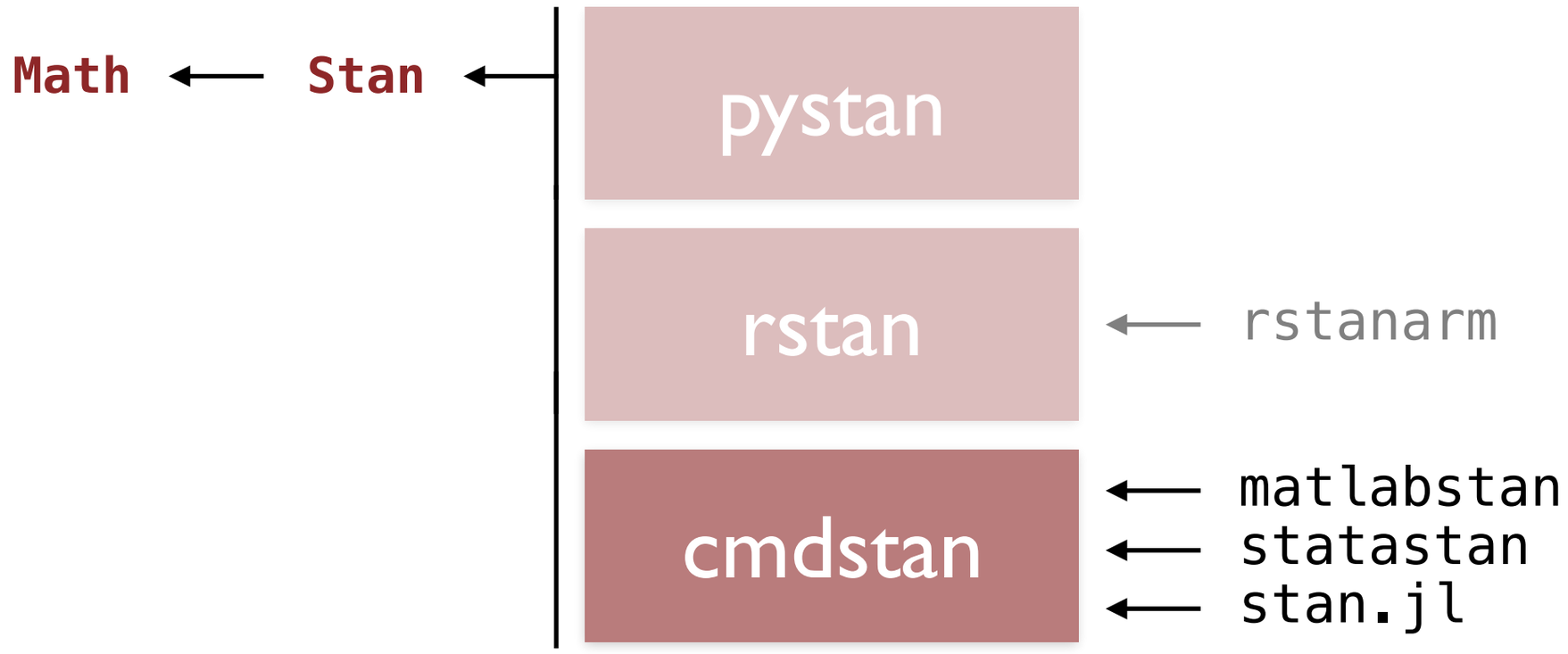
Interfaces



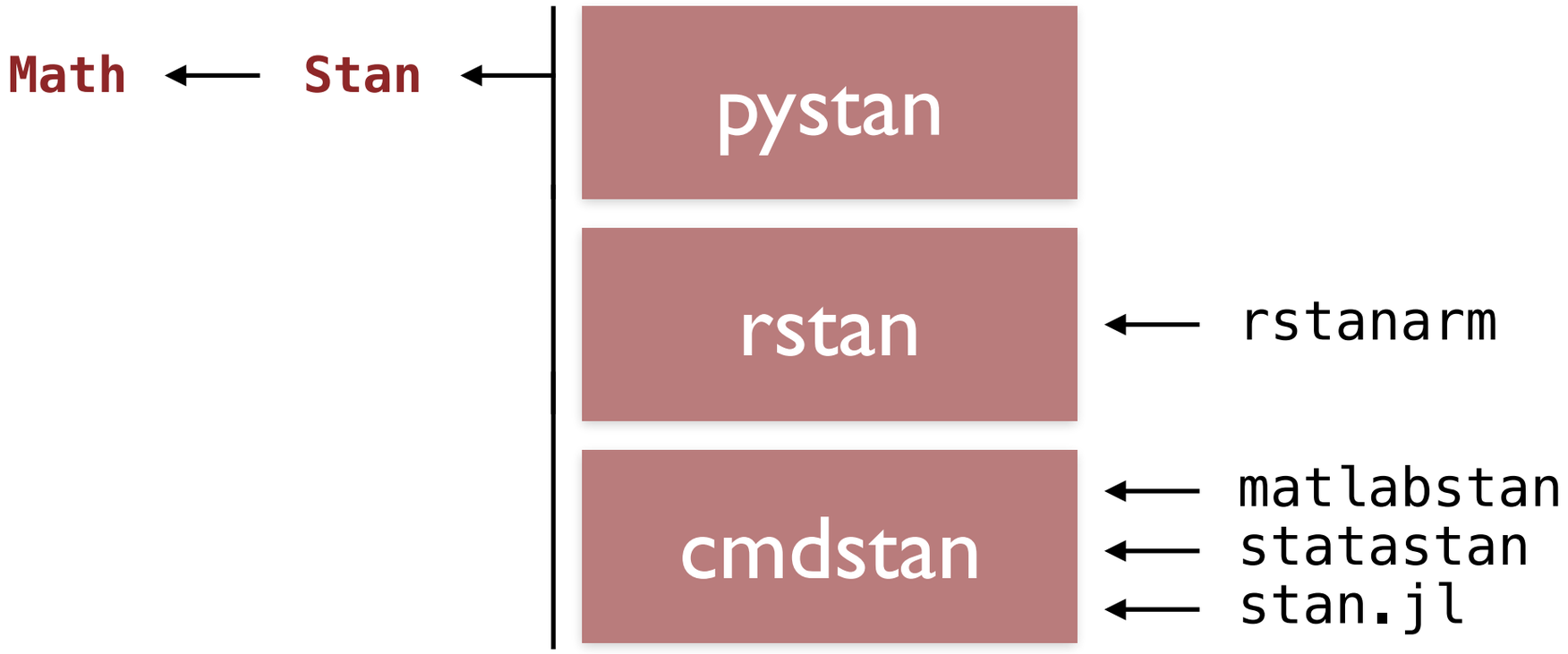
Interfaces



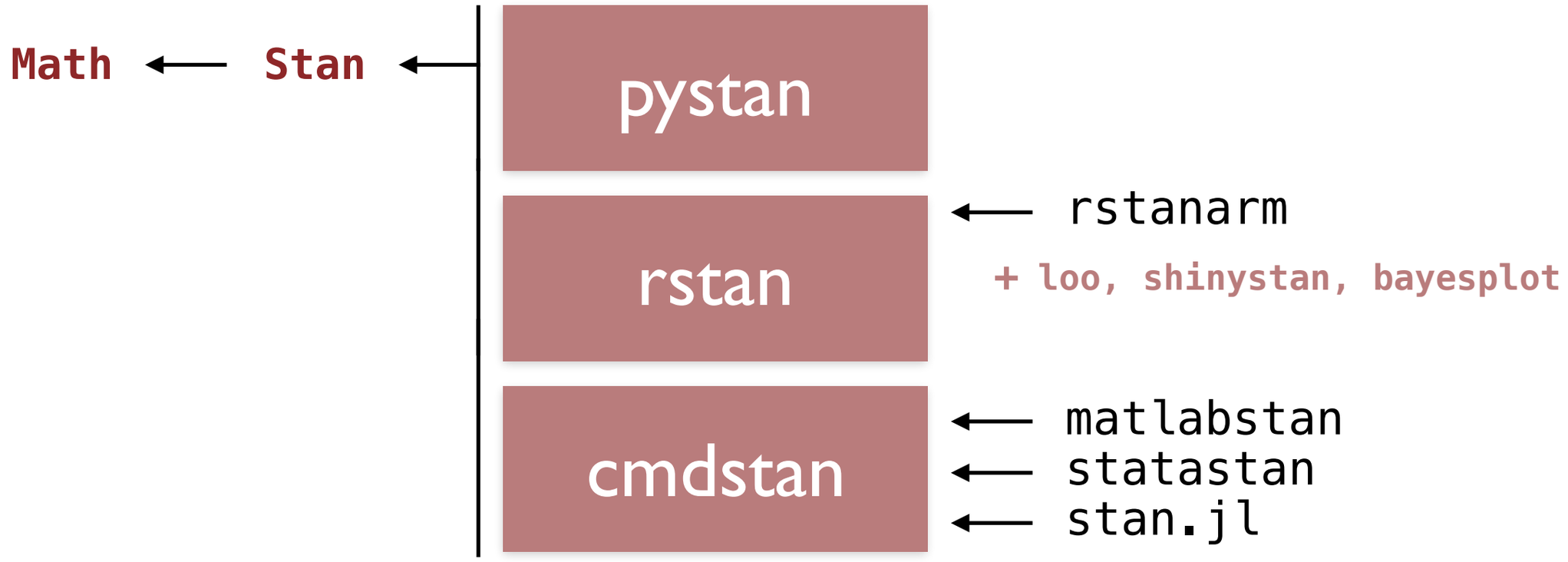
Interfaces


Math
Stan
pystan
rstan
rstanarm
cmdstan
matlabstan
statastan
stan.jl

Interfaces


Math
Stan
pystan
rstan
rstanarm
cmdstan
matlabstan
statastan
stan.jl

Interfaces
+ Tools



We're now going to write a Stan program together

We're now going to write a Stan program together

- Open a new empty file in RStudio

We're now going to write a Stan program together

- Open a new empty file in RStudio
- Save it as `linear-regression.stan`

Generative model of a system of interest

Generative model of a system of interest

Expressed as joint probability distribution of **observed** and **unobserved** variables

$$\pi(\mathcal{D}, \theta)$$

Generative model of a system of interest

Expressed as joint probability distribution of
observed and **unobserved** variables

$$\pi(\mathcal{D}, \theta)$$

Decomposition (convenient but not required)

$$\pi(\mathcal{D} \mid \theta)\, \pi(\theta)$$

Generative model of a system of interest

Expressed as joint probability distribution of
observed and **unobserved** variables

$$\pi(\mathcal{D}, \theta)$$

Decomposition (convenient but not required)

$$\pi(\mathcal{D} \mid \theta)\, \pi(\theta)$$

Posterior is proportional to this joint distribution

$$\pi(\theta \mid \mathcal{D}) \propto \pi(\mathcal{D} \mid \theta)\pi(\theta)$$

All computations are actually done on *log* scale

$$\log \pi(\theta|\mathcal{D}) = \log \pi(\mathcal{D}|\theta) + \log \pi(\theta) + \text{constant}$$

All computations are actually done on *log* scale

$$\log \pi(\theta|\mathcal{D}) = \log \pi(\mathcal{D}|\theta) + \log \pi(\theta) + \text{constant}$$

Products become sums of logs

$$\pi(\mathcal{D}|\theta) = \prod_{n=1}^{N} \pi(\mathcal{D}_n|\theta) \longrightarrow \log \pi(\mathcal{D}|\theta) = \sum_{n=1}^{N} \log \pi(\mathcal{D}_n|\theta)$$

Stan programs are organized into *blocks*

Stan programs are organized into *blocks*

```
block name {

  contents …

}
```

data block

data block

- Declare data types, sizes, and constraints

data block

- Declare data types, sizes, and constraints
- Read from data source and constraints validated

data block

- Declare data types, sizes, and constraints
- Read from data source and constraints validated
- Evaluated:
  - once

```
data {
  // Dimensions


  // Variables

}
```

```
data {
  // Dimensions
  int<lower=1> N;


  // Variables

}
```

```
data {
  // Dimensions
  int<lower=1> N;
  int<lower=1> K;

  // Variables
}
```

```
data {
  // Dimensions
  int<lower=1> N;
  int<lower=1> K;

  // Variables
  matrix[N, K] X;
}
```

```
data {
  // Dimensions
  int<lower=1> N;
  int<lower=1> K;

  // Variables
  matrix[N, K] X;
  vector[N] y;
}
```

```
data {
  // Dimensions
  int<lower=1> N;
  int<lower=1> K;

  // Variables
  matrix[N, K] X;
  vector[N] y;
}
```

```
//  single line comment
```

```
data {
  // Dimensions
  int<lower=1> N;
  int<lower=1> K;

  // Variables
  matrix[N, K] X;
  vector[N] y;
}
```

```
//  single line comment
/* multiple lines of
comments  */
```

parameters block

parameters block

- Declare parameter types, sizes, and constraints

parameters block

- Declare parameter types, sizes, and constraints
- Transformations (under the hood) for constrained parameters

parameters block

- Declare parameter types, sizes, and constraints
- Transformations (under the hood) for constrained parameters

- Evaluated:
  - every log prob evaluation

```
parameters {

}
```

```
parameters {
   real alpha;


}
```

```
parameters {
   real alpha;
   vector[K] beta;

}
```

```
parameters {
   real alpha;
   vector[K] beta;
   real<lower=0> sigma;
}
```

constraints *required* in **`parameters`** block

model block

model block

- Statements defining the posterior density

  - log scale

model block

- Statements defining the posterior density
  - log scale
- Evaluated:
  - every log prob evaluation

```
model {

}
```

```
model {

  sigma ~ exponential(1);

}
```

```
model {

  sigma ~ exponential(1);
  alpha ~ normal(0, 10);

}
```

```
model {

  sigma ~ exponential(1);
  alpha ~ normal(0, 10);
  for (k in 1:K) beta[k] ~ normal(0, 5);

}
```

```
model {
  // priors  (flat, uniform, if omitted)
  sigma ~ exponential(1);
  alpha ~ normal(0, 10);
  for (k in 1:K) beta[k] ~ normal(0, 5);
}
```

```
model {
  // priors  (flat, uniform, if omitted)
  sigma ~ exponential(1);
  alpha ~ normal(0, 10);
  for (k in 1:K) beta[k] ~ normal(0, 5);
}
```

Why is the default automatically uniform?

```
model {
  // priors  (flat, uniform, if omitted)
  sigma ~ exponential(1);
  alpha ~ normal(0, 10);
  for (k in 1:K) beta[k] ~ normal(0, 5);
}
```

Why is the default automatically uniform?

- $p(\theta) \propto 1$ (0 on log scale)

```
model {
  // priors  (flat, uniform, if omitted)
  sigma ~ exponential(1);
  alpha ~ normal(0, 10);
  for (k in 1:K) beta[k] ~ normal(0, 5);
}
```

Why is the default automatically uniform?

- $p(\theta) \propto 1$ (0 on log scale)
- Nothing added to log prob

```
model {
  // priors  (flat, uniform, if omitted)
  sigma ~ exponential(1);
  alpha ~ normal(0, 10);
  for (k in 1:K) beta[k] ~ normal(0, 5);
  for (n in 1:N) {
    y[n] ~ normal(X[n, ] * beta + alpha, sigma);
  }
}
```

Why is the default automatically uniform?

- $p(\theta) \propto 1$ (0 on log scale)
- Nothing added to log prob

generated quantities block

generated quantities block

- Declare and define derived variables

  - (P)RNGs, predictions, event probabilities, decision making

generated quantities block

- Declare and define derived variables
  - (P)RNGs, predictions, event probabilities, decision making
- Constraints validated

generated quantities block

- Declare and define derived variables
  - (P)RNGs, predictions, event probabilities, decision making
- Constraints validated

- Evaluated:
  - once per draw

```
generated quantities {

}
```

```
generated quantities {
   vector[N] y_rep;

}
```

```
generated quantities {
   vector[N] y_rep;
   for (n in 1:N) {
   }
}
```

```
generated quantities {
   vector[N] y_rep;
   for (n in 1:N) {
     real y_hat = X[n,] * beta + alpha;  // local/temp
     }
}
```

```
generated quantities {
   vector[N] y_rep;
   for (n in 1:N) {
     real y_hat = X[n,] * beta + alpha;  // local/temp
     y_rep[n] =
   }
}
```

```
generated quantities {
   vector[N] y_rep;
   for (n in 1:N) {
     real y_hat = X[n,] * beta + alpha;  // local/temp
     y_rep[n] = normal_rng(y_hat, sigma;)
   }
}
```

The complete Stan program

The complete Stan program

Observed variables

Unobserved variables

$$\log \pi(\theta)$$

$$+$$

$$\log \pi(\mathcal{D}|\theta)$$

Simulate from generative model

The complete Stan program

```
data {
  int<lower=1>  N;
  int<lower=1>  K;
  matrix[N, K] X;
  vector[N] y;
}
```

Observed variables

Unobserved variables

$$\log \pi(\theta)$$

$$+$$

$$\log \pi(\mathcal{D}|\theta)$$

Simulate from generative model

The complete Stan program

```
data {
  int<lower=1>  N;
  int<lower=1>  K;
  matrix[N, K] X;
  vector[N] y;
}
parameters {
  real alpha;
  vector[K] beta;
  real<lower=0> sigma;
}
```

Observed variables

Unobserved variables

$$\log \pi(\theta)$$

$$+$$

$$\log \pi(\mathcal{D}|\theta)$$

Simulate from generative model

The complete Stan program

```
data {
  int<lower=1>  N;
  int<lower=1>  K;
  matrix[N, K] X;
  vector[N] y;
}
parameters {
  real alpha;
  vector[K] beta;
  real<lower=0> sigma;
}
model {
  sigma ~ exponential(1);
  alpha ~ normal(0, 10);
  for (k in 1:K) beta[k] ~ normal(0, 5);

  for (n in 1:N)
    y[n] ~ normal(alpha + X[n, ] * beta, sigma);
}
```

Observed variables

Unobserved variables

$$\log \pi(\theta)$$

$$+$$

$$\log \pi(\mathcal{D}|\theta)$$

Simulate from generative model

The complete Stan program

```
data {
  int<lower=1>  N;
  int<lower=1>  K;
  matrix[N, K] X;
  vector[N] y;
}
parameters {
  real alpha;
  vector[K] beta;
  real<lower=0> sigma;
}
model {
  sigma ~ exponential(1);
  alpha ~ normal(0, 10);
  for (k in 1:K) beta[k] ~ normal(0, 5);

  for (n in 1:N)
    y[n] ~ normal(alpha + X[n, ] * beta, sigma);
}
generated quantities {
  vector[N] y_rep;
  for (n in 1:N)
    y_rep[n] = normal_rng(alpha + X[n,] * beta, sigma);
}
```

Observed variables

Unobserved variables

$$\log \pi(\theta)$$

$$+$$

$$\log \pi(\mathcal{D}|\theta)$$

Simulate from generative model

Best Practices

1a. Maintain reproducibility by saving the model, data, and inits in files and the R commands in scripts.

1a. Maintain reproducibility by saving the model, data, and inits in files and the R commands in scripts.

1b. Use version control on your files and scripts.

1a. Maintain reproducibility by saving the model, data, and inits in files and the R commands in scripts.

1b. Use version control on your files and scripts.

2. Start simple! Build your model in stages, ensuring good fits at each stage.

1a. Maintain reproducibility by saving the model, data, and inits in files and the R commands in scripts.

1b. Use version control on your files and scripts.

2. Start simple! Build your model in stages, ensuring good fits at each stage.

3. Fit your model to simulated data to ensure that Stan can recover the true values.

1a. Maintain reproducibility by saving the model, data, and inits in files and the R commands in scripts.

1b. Use version control on your files and scripts.

2. Start simple! Build your model in stages, ensuring good fits at each stage.

3. Fit your model to simulated data to ensure that Stan can recover the true values.

4. Keep an eye on the diagnostics!

http://mc-stan.org/workshops/learn_bayes/

Exercise: same model using `target +=`

Under the hood, sampling statements just add to the total log probability

Under the hood, sampling statements just add to the total log probability

```
z ~ normal(0, 1);
```

Under the hood, sampling statements just add to the total log probability

```
z ~ normal(0, 1);
```

```
target += normal_lpdf(z | 0, 1);
```

Under the hood, sampling statements just add to the total log probability

```
z ~ normal(0, 1);
```

```
target += normal_lpdf(z | 0, 1);
```

Evaluate log-pdf of normal (with mean 0, sd 1) at the value z

Under the hood, sampling statements just add to the total log probability

```
z ~ normal(0, 1);
```

```
target += normal_lpdf(z | 0, 1);
```

Evaluate log-pdf of normal (with mean 0, sd 1) at the value z

```
log( 1/sqrt(2 * pi) * exp(-z^2 / 2) )
```

Under the hood, sampling statements just add to the total log probability

```
z ~ normal(0, 1);
```

```
target += normal_lpdf(z | 0, 1);
```

Evaluate log-pdf of normal (with mean 0, sd 1) at the value z

```
log( 1/sqrt(2 * pi) * exp(-z^2 / 2) )
```

Then add this value to the total log-probability