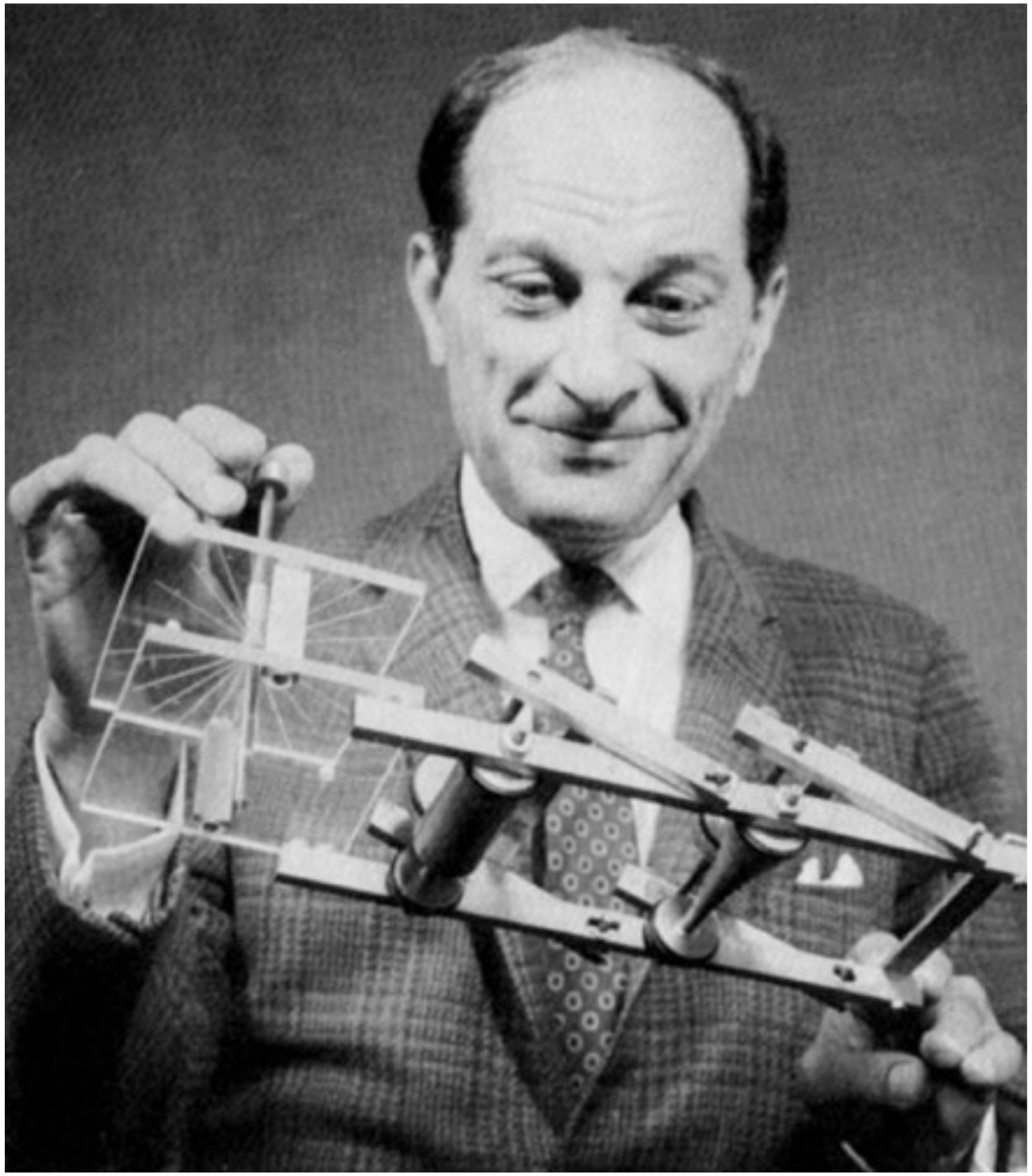
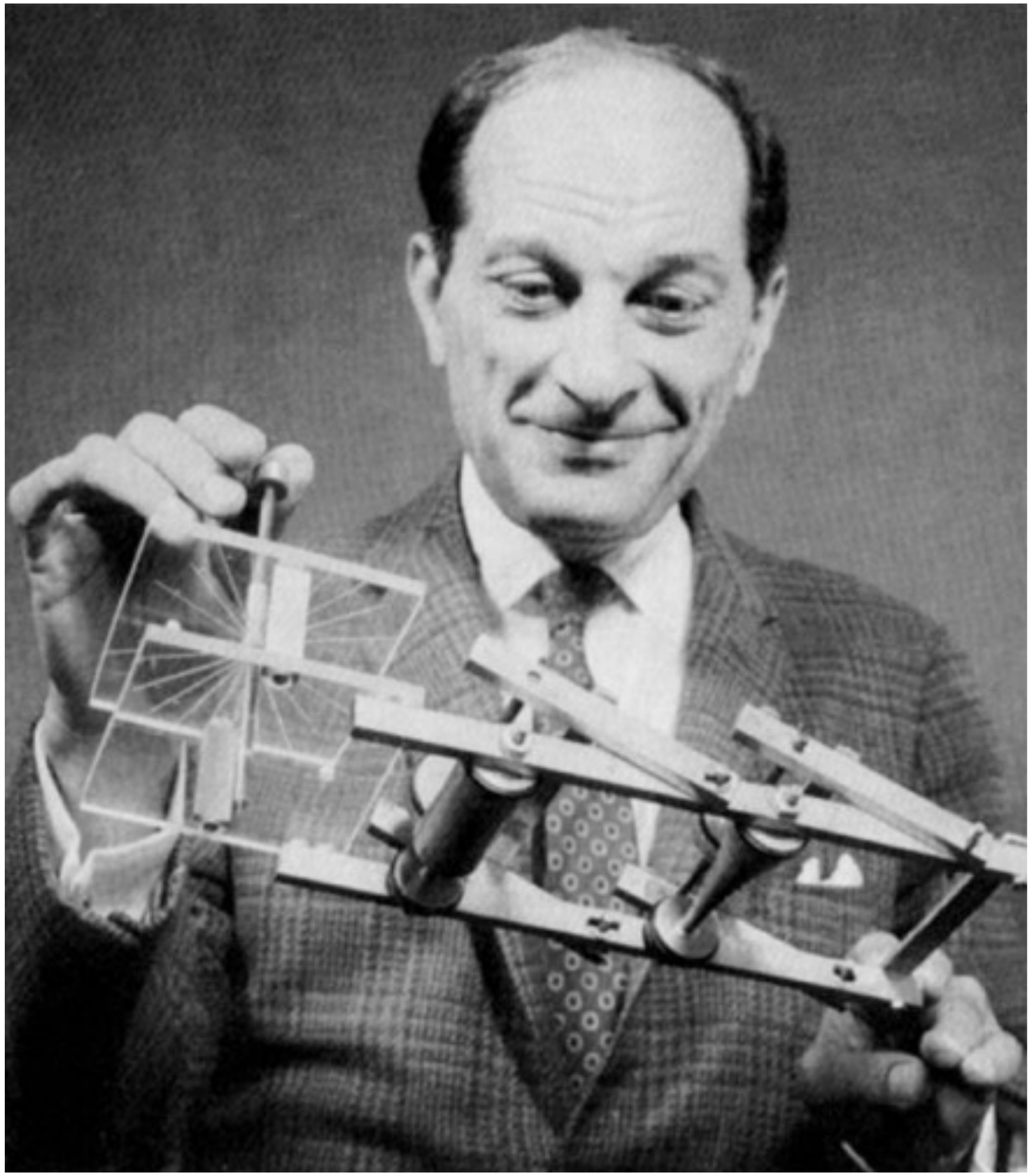
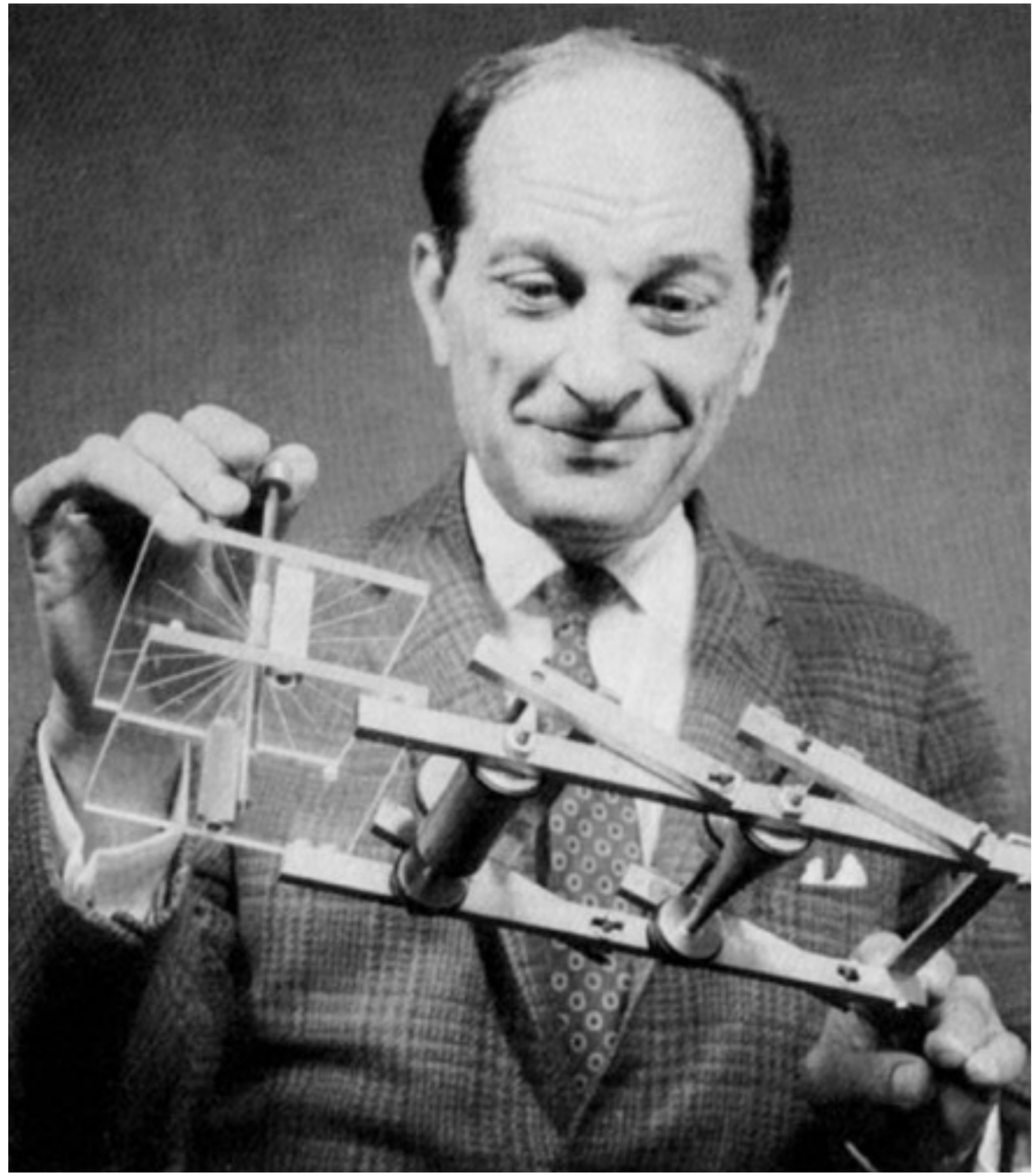
Intro to Stan

Jonah Gabry
Columbia University



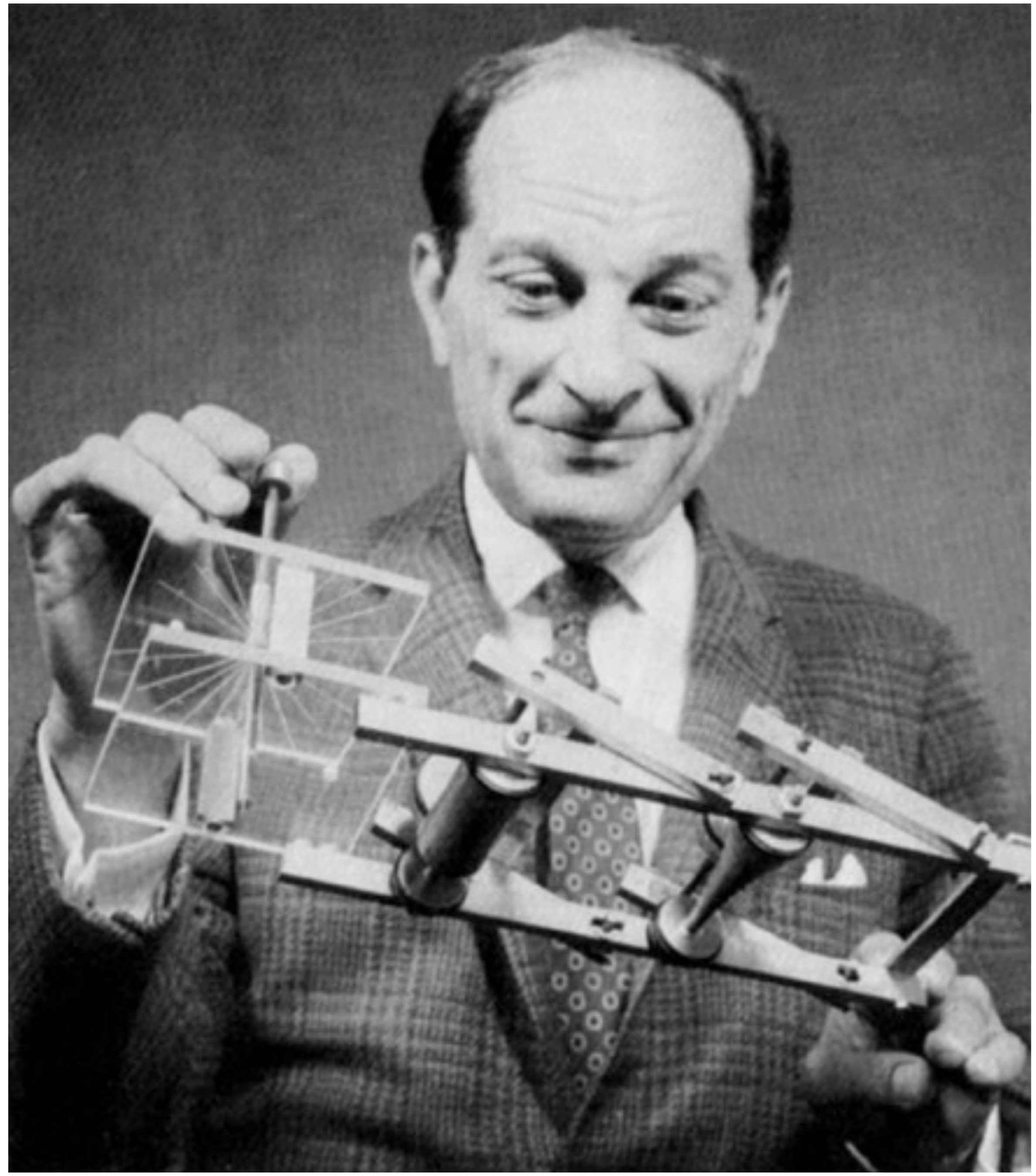


Stanislaw Ulam
(1909–1984)



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

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
  - Extensible
  - VB for approximate Bayes
  - Optimization for (penalized) MLE

Why Stan?

- Fit rich Bayesian statistical models
- Efficiency
  - HMC + NUTS
  - Compiled to C++
- Flexible domain specific language
  - Extensible
  - VB for approximate Bayes
  - Optimization for (penalized) MLE
- Open source
  - BSD
  - CC-BY

Who is using Stan?

Who is using Stan?

Biological sciences

- clinical trials
- epidemiology
- genomics
- population ecology
- entomology
- ophthalmology
- neurology
- agriculture
- fisheries
- cancer biology

Who is using Stan?

Physical sciences

Who is using Stan?

Physical sciences

- astrophysics
  - LIGO gravitational wave observation
- 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
  - textbook coming soon! (ish)

Who is using Stan?

More…

Who is using Stan?

More…

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

Stan is many things

Stan is many things

`Math`

Stan is many things

Math $\longleftarrow$ **Language**

Stan is many things

Math ⟵ **Language** ⟵ **Algorithms**

Stan is many things

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

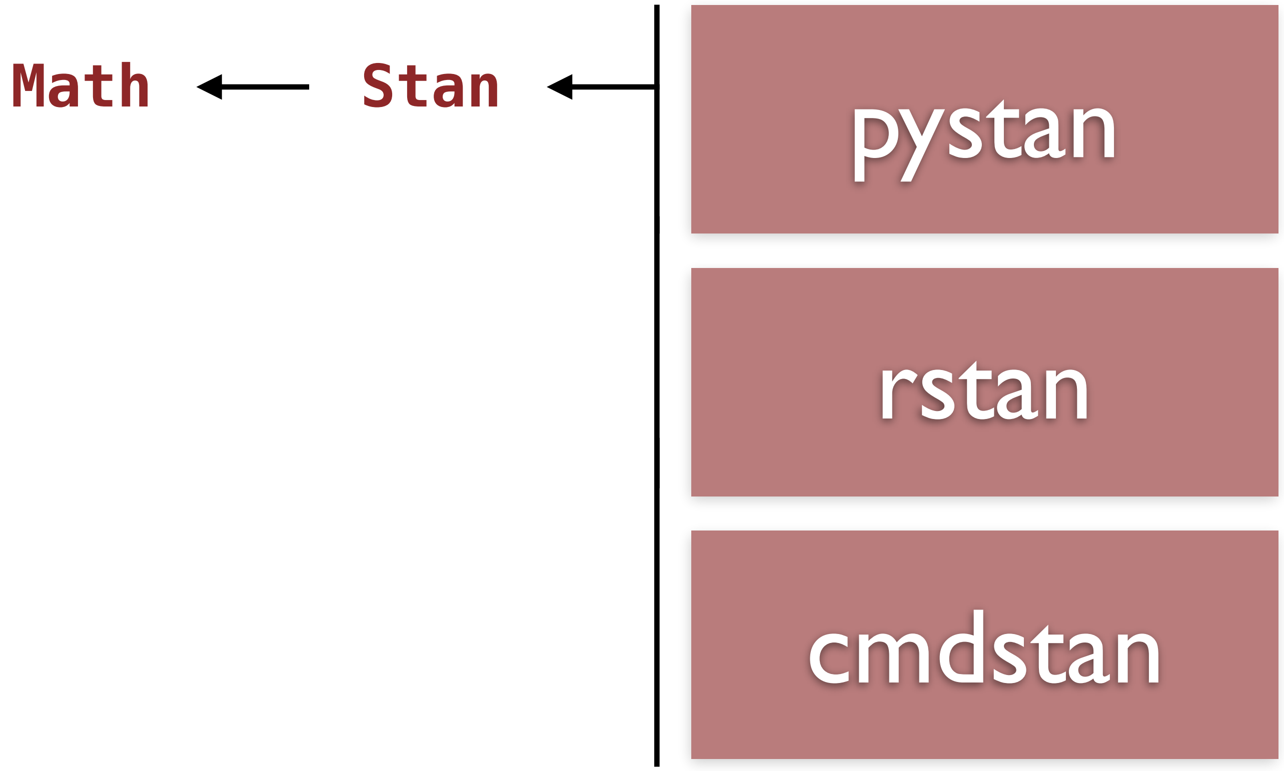
Stan is many things

Math ⟵ **Stan**

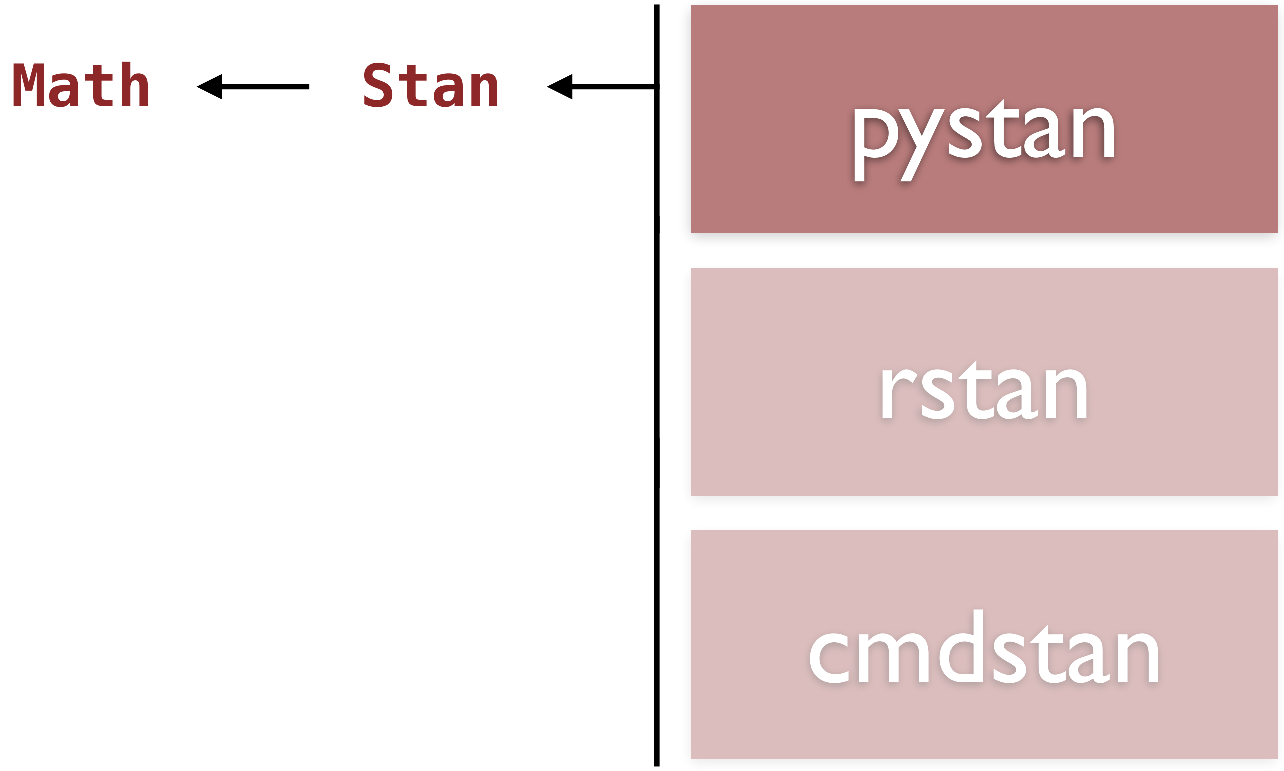
Interfaces

`Math` $\longleftarrow$ `Stan`

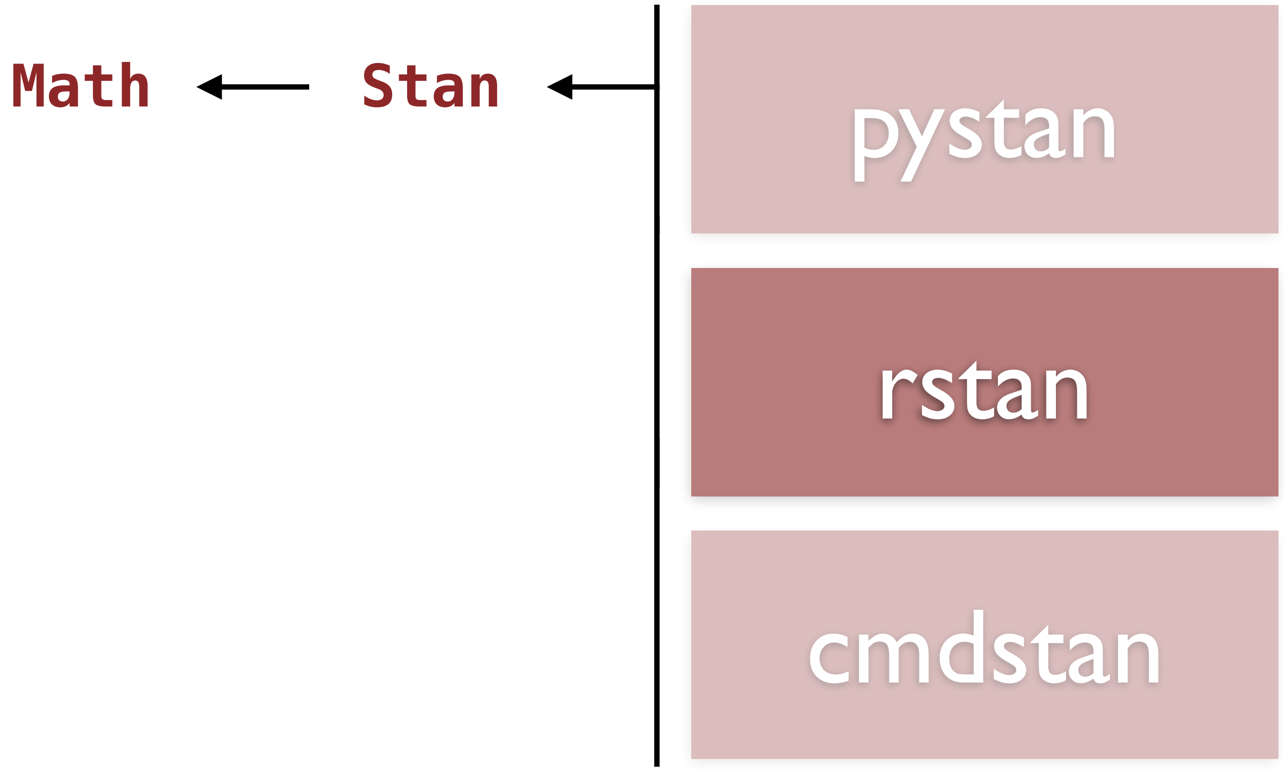
Interfaces



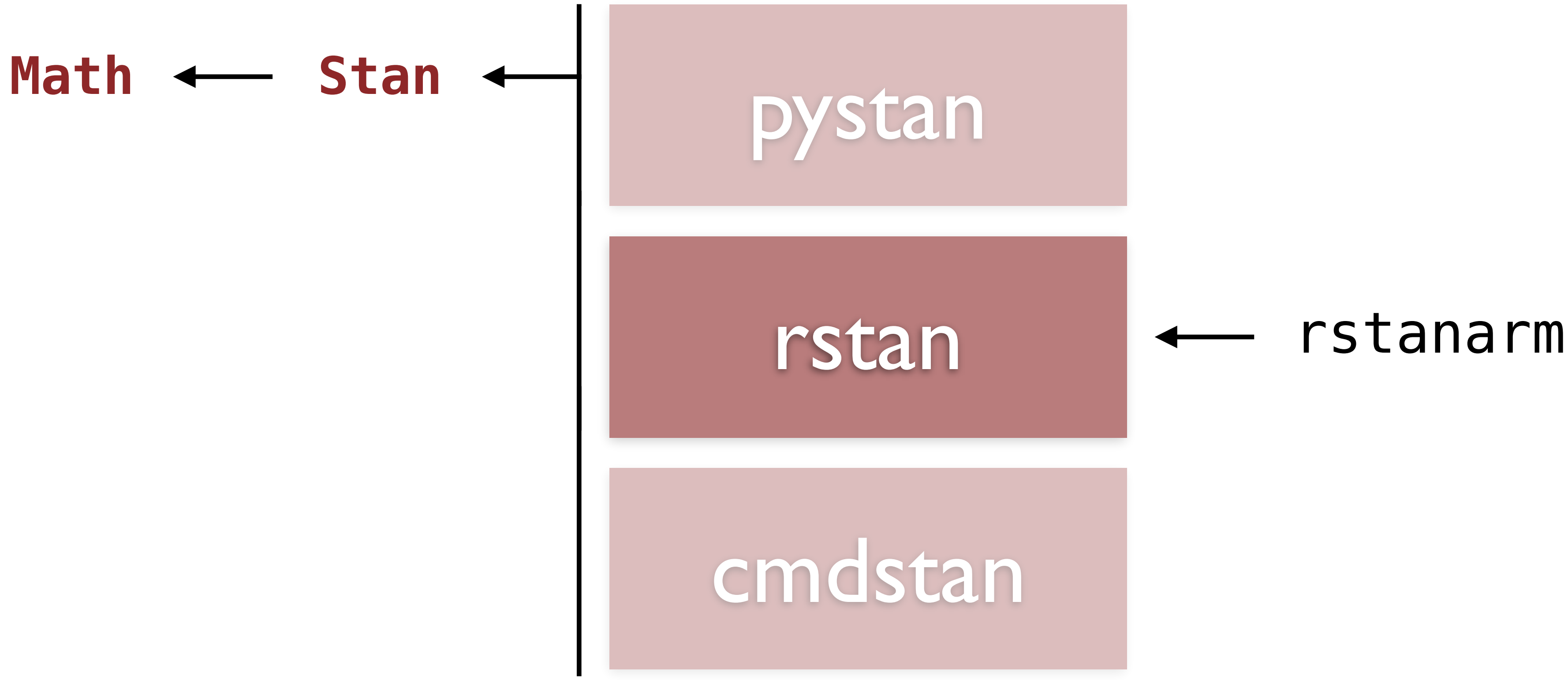
Interfaces



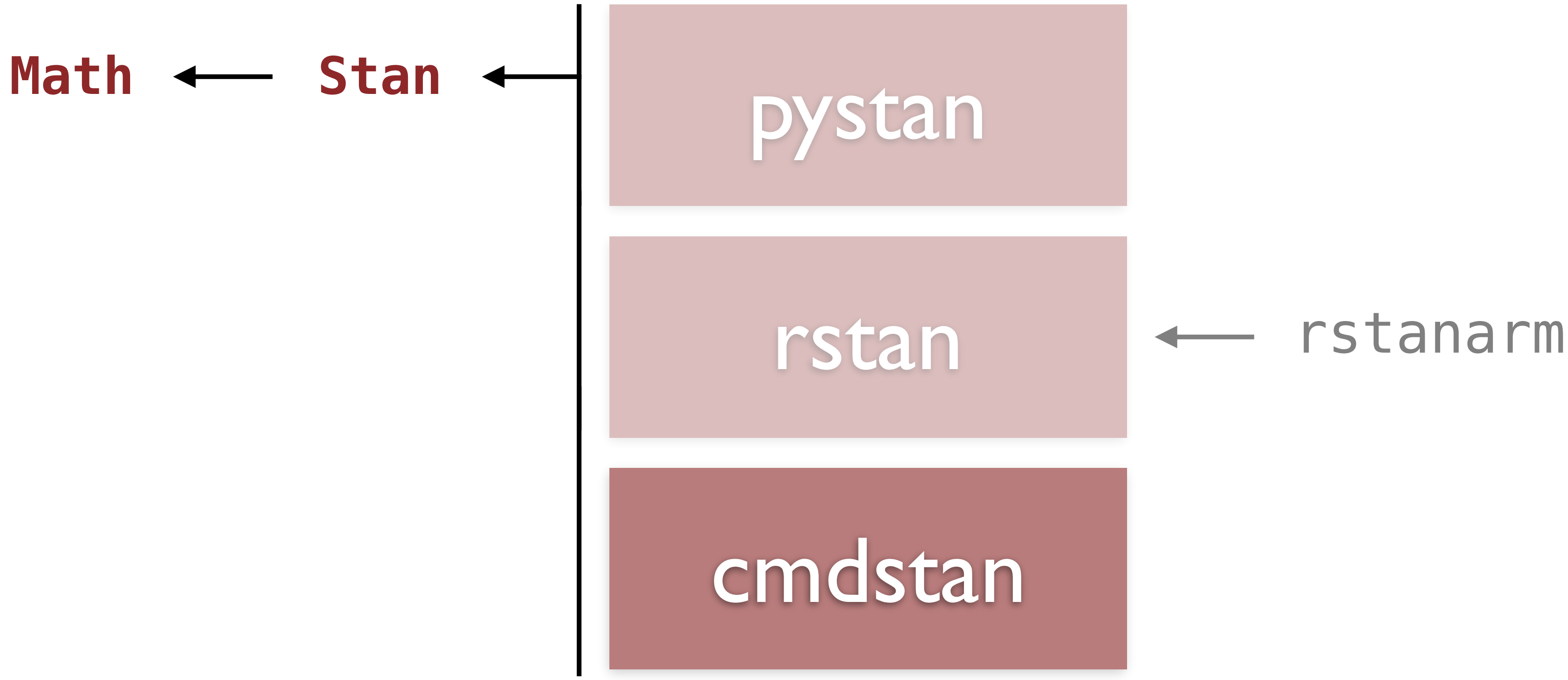
Interfaces



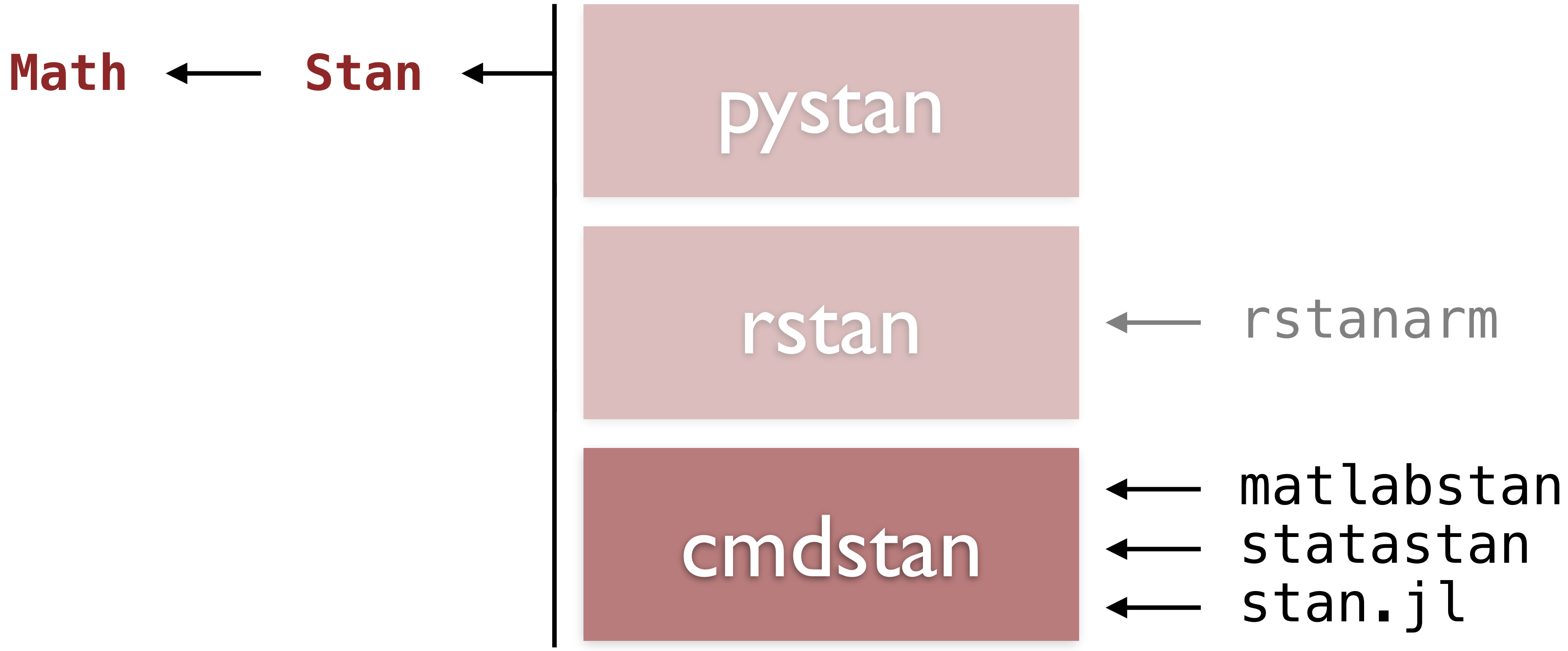
Interfaces



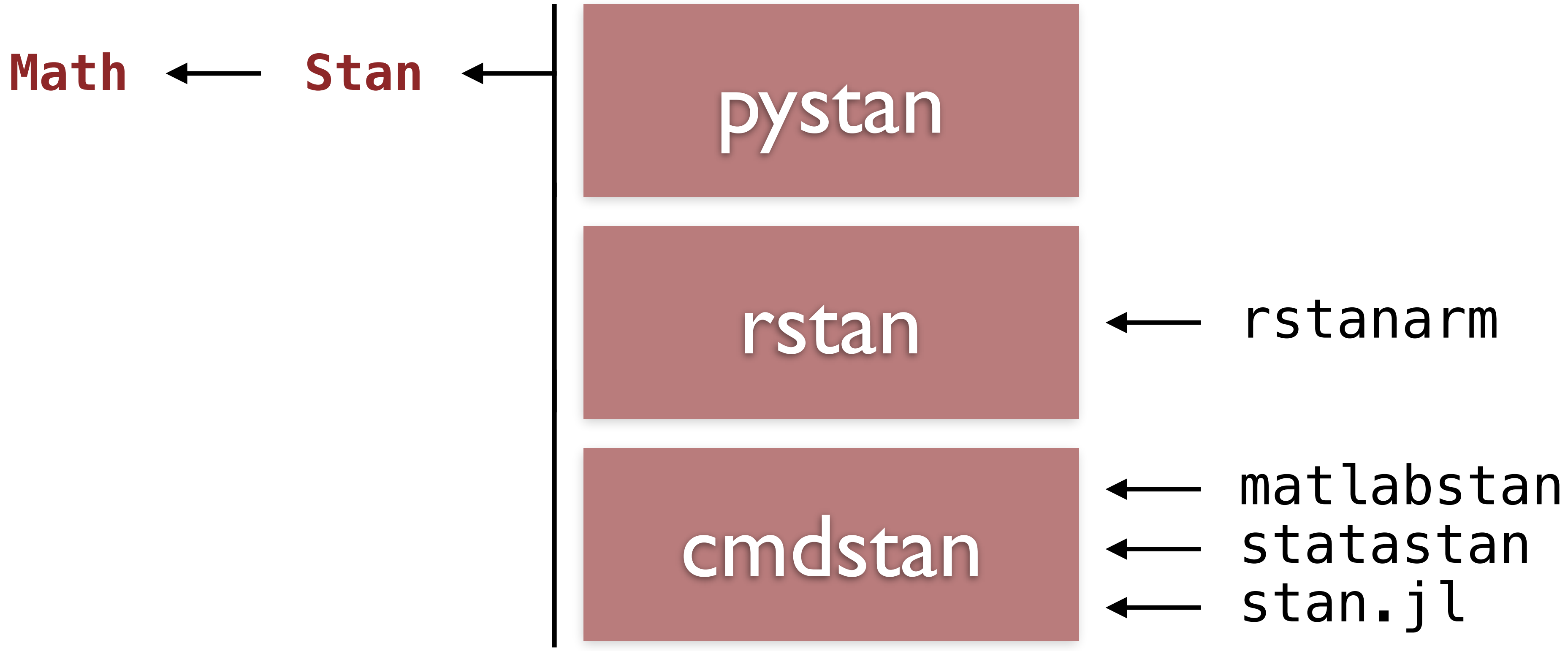
Interfaces



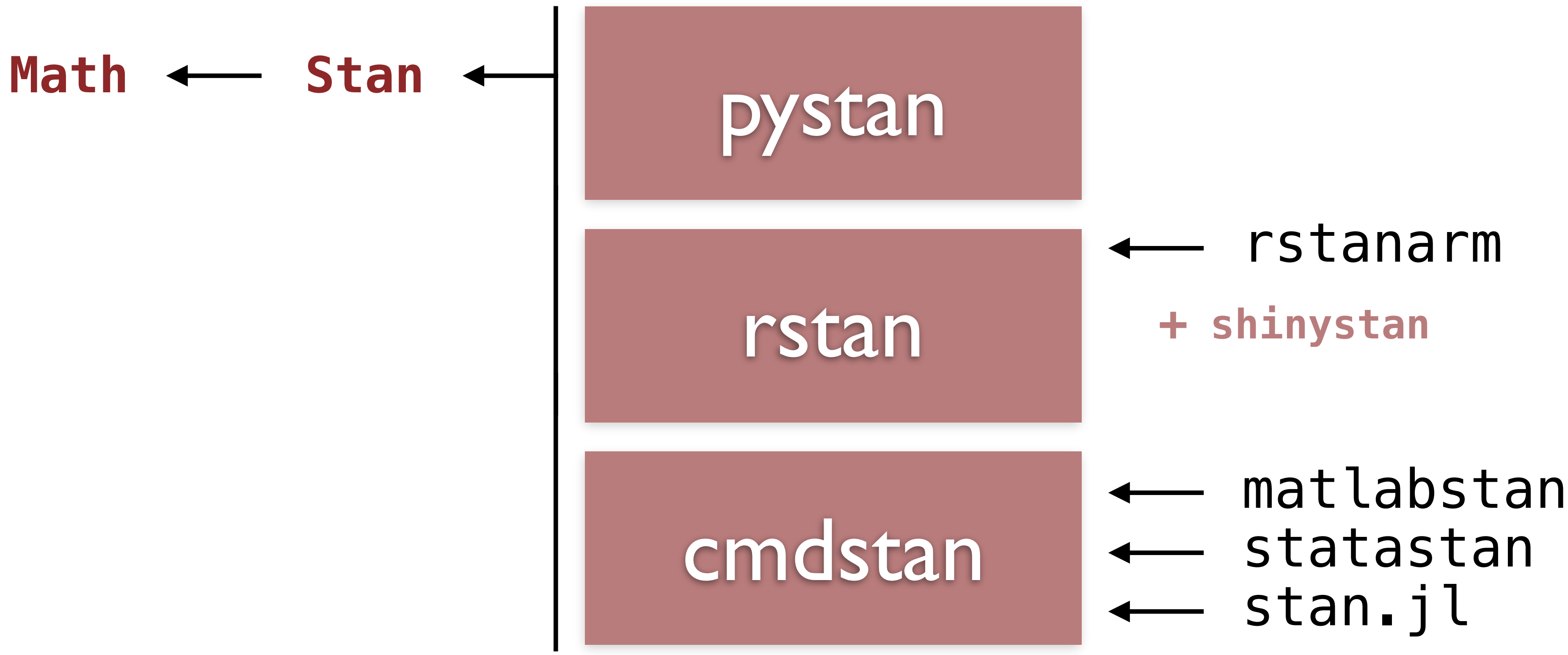
Interfaces


Math
Stan
pystan
rstan
cmdstan
rstanarm
matlabstan
statastan
stan.jl

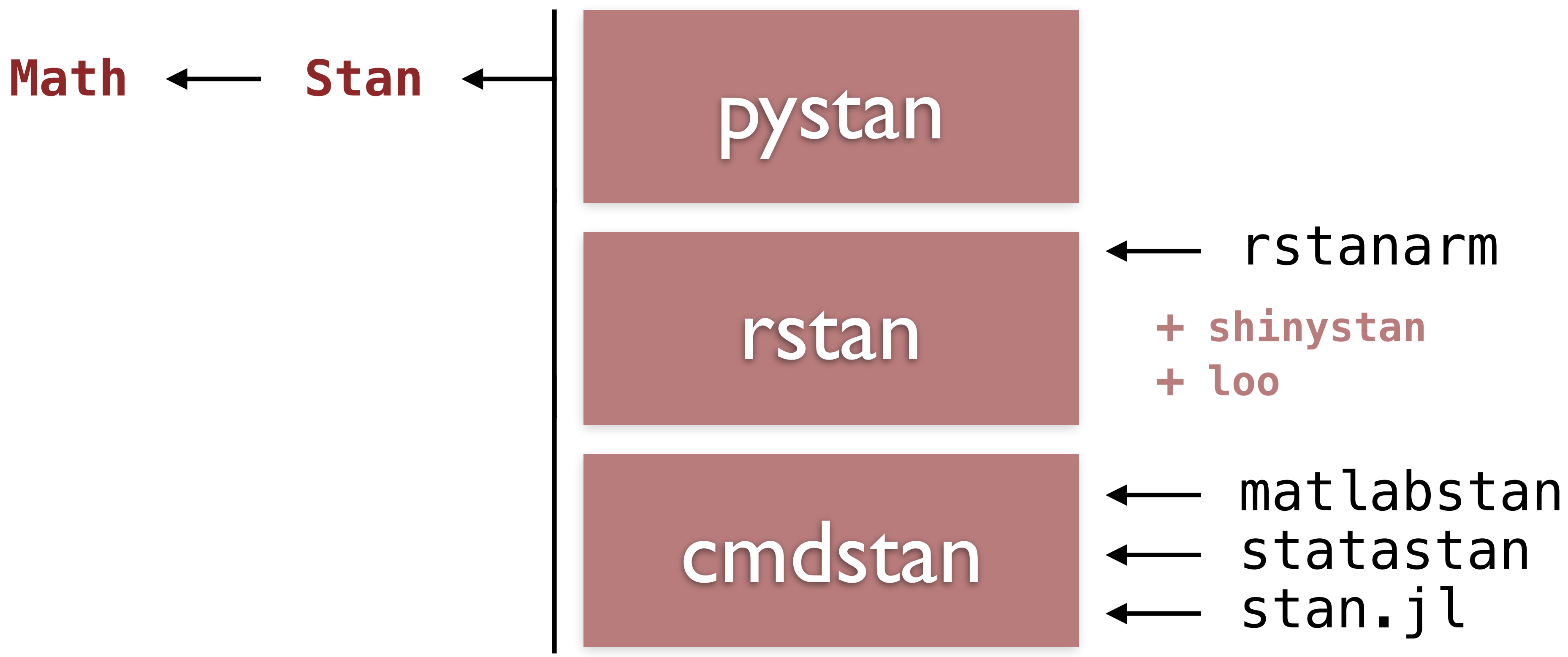
Interfaces


Math
Stan
pystan
rstan
cmdstan
rstanarm
matlabstan
statastan
stan.jl

Interfaces
+ Tools



Interfaces
+ Tools



Components of a
Stan Program

Before we continue, install a few things:

Before we continue, install a few things:

- If you haven't already, go to https://github.com/jgabry/Bayes-Stan-Course

Before we continue, install a few things:

- If you haven't already, go to https://github.com/jgabry/Bayes-Stan-Course

- And then copy and run the code in the README file:

```
# install 'devtools' and then 'bayesplot'
if (!require("devtools")) install.packages("devtools")
library("devtools")
install_github("jgabry/bayesplot")

# also install 'shinystan'
install.packages("shinystan")
```

Posterior densities are specified in a comprehensive user-oriented probabilistic programming language.

$$p(q \mid \mathcal{D})$$

When writing a Stan program we always have three fundamental components

$$p(q \mid \mathcal{D})$$

When writing a Stan program we always have three fundamental components

$$p(q \mid \mathcal{D})$$

What are we conditioning on?

When writing a Stan program we always have three fundamental components

$$p(q \mid \mathcal{D})$$

What are we conditioning on?

What are the parameters?

When writing a Stan program we always have three fundamental components

$$p(q \mid \mathcal{D})$$

What are we conditioning on?

What are the parameters?

How are they related?

We're now going to write a Stan program

We're now going to write a Stan program

- Open a new empty file in RStudio

We're now going to write a Stan program

- Open a new empty file in RStudio

- Save it as `linear-regression.stan`

Stan programs are organized into *blocks*

Stan programs are organized into *blocks*

```
block name {

  block contents

}
```

Data

Data

- Declare data types, sizes, and constraints

Data

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

Data

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

Data

```
data {
  // Dimensions



  // Variables



}
```

Data

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


  // Variables


}
```

Data

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

  // Variables



}
```

Data

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

  // Variables
  matrix[N,K] X;

}
```

Data

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

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

Data

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

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

```
// single line comment
```

Data

```
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

Parameters

- Declare parameter types, sizes, and constraints

Parameters

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

Parameters

- Declare parameter types, sizes, and constraints
- Transformations (under the hood) for constrained parameters
- Evaluated:
  - every log prob evaluation

Parameters

```
parameters {



}
```

Parameters

```
parameters {
  real alpha;


}
```

Parameters

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

}
```

Parameters

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

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

Model

Model

- Statements defining the posterior density
  - log scale

Model

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

Model

```
model {



}
```

Model

```
model {
   y ~ normal(X * beta + alpha, sigma);




}
```

Model

```
model {
   y ~ normal(X * beta + alpha, sigma);

   // priors  (flat, uniform, if omitted)




}
```

Model

```
model {
   y ~ normal(X * beta + alpha, sigma);

   // priors  (flat, uniform, if omitted)



}
```

Why is the default automatically uniform?

Model

```
model {
   y ~ normal(X * beta + alpha, sigma);

   // priors  (flat, uniform, if omitted)




}
```

Why is the default automatically uniform?

- $p(\theta) \propto 1$

Model

```
model {
   y ~ normal(X * beta + alpha, sigma);

   // priors  (flat, uniform, if omitted)

}
```

Why is the default automatically uniform?

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

Model

```
model {
   y ~ normal(X * beta + alpha, sigma);

   // priors  (flat, uniform, if omitted)
   alpha ~ normal(0, 10);



}
```

Why is the default automatically uniform?

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

Model

```
model {
   y ~ normal(X * beta + alpha, sigma);

   // priors  (flat, uniform, if omitted)
   alpha ~ normal(0, 10);
   beta ~ normal(0, 10);

}
```

Why is the default automatically uniform?

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

Model

```
model {
   y ~ normal(X * beta + alpha, sigma);

   // priors  (flat, uniform, if omitted)
   alpha ~ normal(0, 10);
   beta ~ normal(0, 10);
   sigma ~ cauchy(0, 10);
}
```

Why is the default automatically uniform?

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

Generated Quantities

Generated Quantities

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

Generated Quantities

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

Generated Quantities

- Declare and define derived variables
  - (P)RNGs, predictions, event probabilities, decision making
- Constraints validated
- Evaluated:
  - once per draw

Generated Quantities

```
generated quantities {



}
```

Generated Quantities

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


}
```

Generated Quantities

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

}
```

Generated Quantities

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

Generated Quantities

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

Generated Quantities

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

Generated Quantities

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

linear-regression.stan

linear-regression.stan

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

linear-regression.stan

```
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;
}
```

linear-regression.stan

```
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 {
  y ~ normal(X * beta + alpha, sigma);
}
```

linear-regression.stan

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

Transformed Data

Transformed Data

- Declare and define transformed data variables

Transformed Data

- Declare and define transformed data variables
- Constraints validated

Transformed Data

- Declare and define transformed data variables
- Constraints validated
- Evaluated:
  - once (after `data`)

Transformed Data

```
transformed data {

}
```

Transformed Data

If we had declared `X` as `K` by `N` instead of `N` by `K` we could transpose it here

```
transformed data {


}
```

Transformed Data

If we had declared `X` as `K` by `N` instead of `N` by `K` we could transpose it here

```
transformed data {
  matrix[N,K] Xt;

}
```

Transformed Data

If we had declared `X` as `K` by `N` instead of `N` by `K` we could transpose it here

```
transformed data {
   matrix[N,K] Xt;
   Xt = X';
}
```

Transformed Data

If we had declared `X` as `K` by `N` instead of `N` by `K` we could transpose it here

```
transformed data {
   matrix[N,K] Xt;
   Xt = X';   // '=' for assignment since 2.10
}
```

`Xt <- X'` deprecated in latest release (still allowed)

Transformed Parameters

Transformed Parameters

- Declare and define transformed parameter variables

Transformed Parameters

- Declare and define transformed parameter variables
- Constraints validated

Transformed Parameters

- Declare and define transformed parameter variables
- Constraints validated
- Evaluated:
  - every log prob evaluation

Transformed Parameters

```
transformed parameters {

}
```

Transformed Parameters

```
transformed parameters {
  vector[N] eta;

}
```

Transformed Parameters

```
transformed parameters {
  vector[N] eta;
  eta = X * beta + alpha;
}
```

Functions

Functions

- Declare and define functions to use in the body of the program

Functions

- Declare and define functions to use in the body of the program

- Compiled with the model

Functions

Functions

```
functions {

  vector linreg_DGP_rng(real alpha, vector beta,
                        real sigma, matrix X) {

    vector[rows(X)] y;
    for (i in 1:rows(X))
      y[i] = normal_rng(alpha + X[i] * beta, sigma);

    return y;
  }

}
```