LMS/BCS-FACS evening seminar

Programming language foundations for statistics

Sam Staton, Oxford

partly based on joint work with Ackerman, Dash, Freer, Jacobs, Kaddar, Moss, Paquet, Perrone, Roy, Sabok, Stein, Wolman, Yang, and others.

Programming language foundations for statistics

1. **Quick look at**
   probabilistic programming for statistics
   example; discussion; Monte Carlo

2. Function spaces ...

3. ... and understanding them.

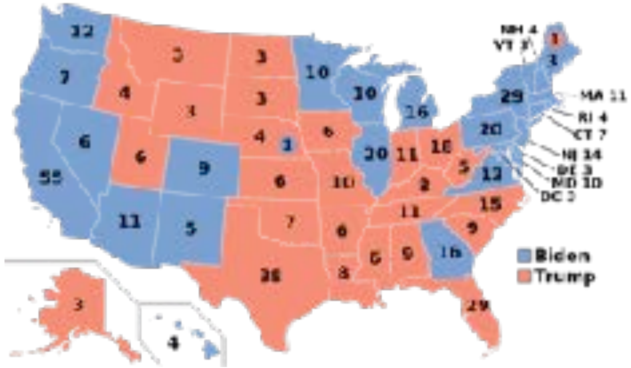
4. Symmetries

High level view: poll example

A very simple model deducing chance of win from poll.

Question:
A quick poll gives 51:49 votes. What is the chance of winning?

Simon Walker / HM Treasury & Simon Dawson / No10 Downing Street



High level view: poll example

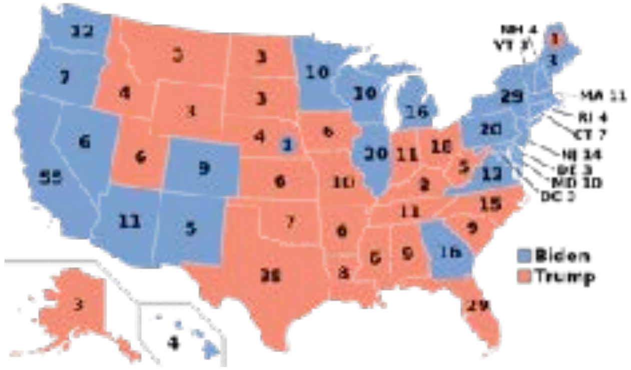
A very simple model deducing chance of win from poll.

Question:

A quick poll gives 51:49 votes. What is the chance of winning?

Clue: it’s not 51%!

Simon Walker / HM Treasury & Simon Dawson / No10 Downing Street

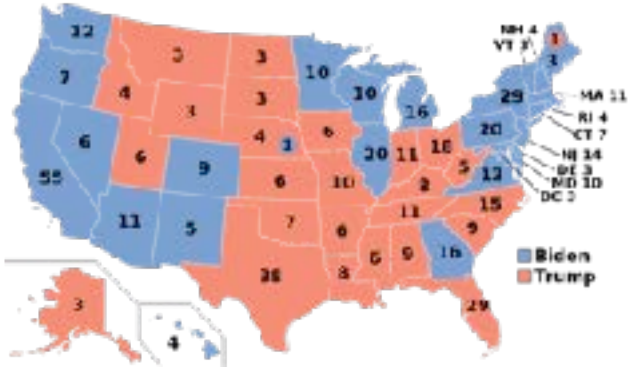


High level view: poll example

A very simple model deducing chance of win from poll.

```
model :: Prob ([Bool] , Bool)
model = do
  voteShare <- uniform 0 1
  votes <- repeat (bernoulli voteShare)
  return (take 100 votes , (voteShare > 0.5))
```

Simon Walker / HM Treasury & Simon Dawson / No10 Downing Street
Open Government Licence v3.0



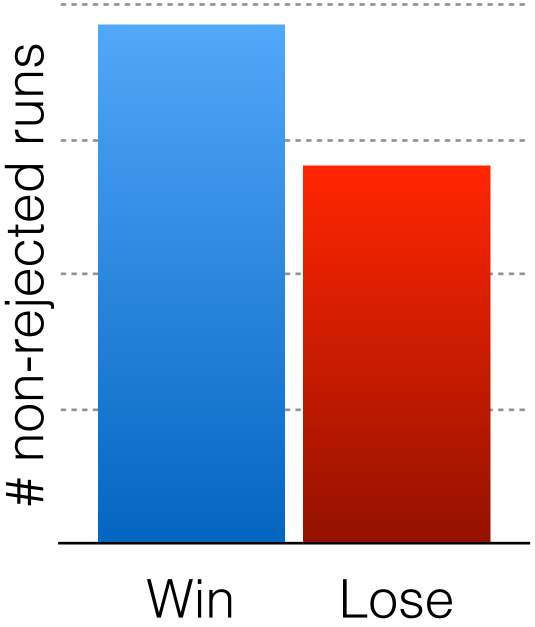
High level view: poll example

A very simple model deducing chance of win from poll.

```
model :: Prob ([Bool] , Bool)
model = do
  voteShare <- uniform 0 1
  votes <- repeat (bernoulli voteShare)
  return (take 100 votes , (voteShare > 0.5))
```

Crude rejection sampling Monte Carlo:

- Run 1000000s of times, each time getting (poll result, win?)
- Reject the runs that mis-predict poll
- What proportion of the remainder are winners?



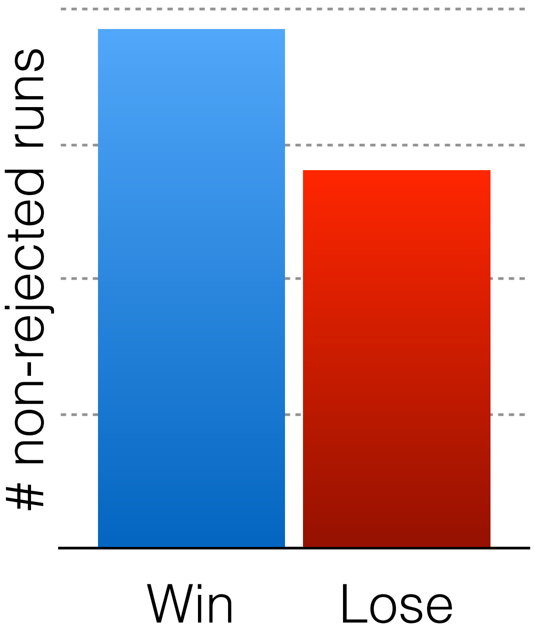
High level view: poll example

A very simple model deducing chance of win from poll.

Question:
A quick poll gives 51:49 votes. What is the chance of winning?
Answer: 0.579.

Crude rejection sampling Monte Carlo:

- Run 1000000s of times, each time getting (poll result, win?)
- Reject the runs that mis-predict poll
- What proportion of the remainder are winners?



High level view: poll example

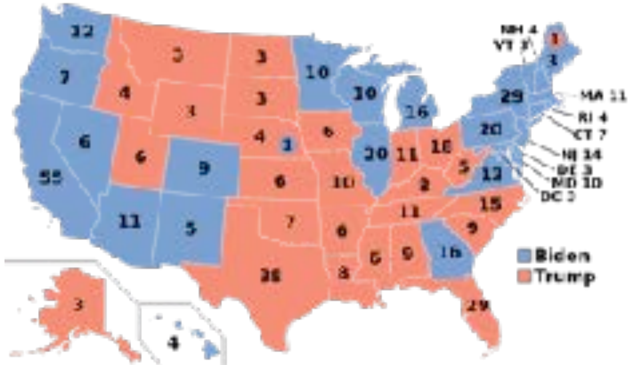
A very simple model deducing chance of win from poll.

Question:

A quick poll gives 51:49 votes. What is the chance of winning?

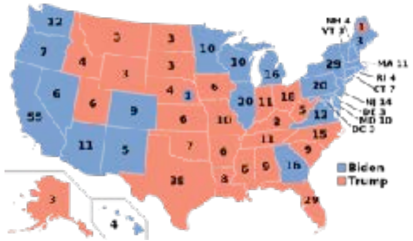
Answer: 0.579.

Simon Walker / HM Treasury & Simon Dawson / No10 Downing Street
Open Government Licence v3.0

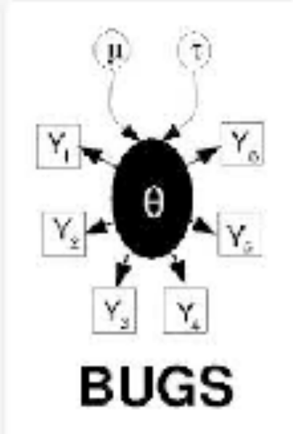


(See Andrew Gelman and coauthors for a proper discussion of using PPL for election modelling.)

Probabilistic programming in practice



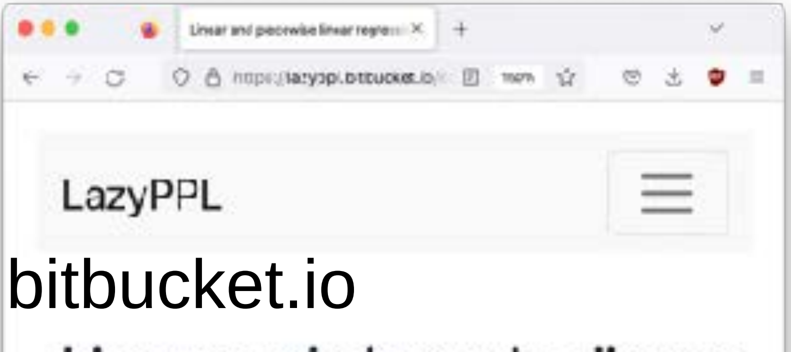
Applications to
social science,
biology,
physical sciences,
machine learning



Stan

PYRO

Church,
Anglican,
Hakaru,
MonadBayes,
Gen...
LazyPPL
https://lazyppl.bitbucket.io


Linear and piecewise linear regres...

https://lazyppl.bitbucket.io/

LazyPPL

PyMC

...

Dash, Kaddar, Paquet, Staton, POPL 2023

Abstraction in traditional programming

High level e.g. higher-order functions
abstract types

Low level e.g. machine code,
Boolean circuits

Abstraction in traditional programming

High level e.g. higher-order functions
abstract types

Low level e.g. machine code,
Boolean circuits

	Engineering	Foundational
High level	✓	✓
Low level	✓	✓

Abstraction in *probabilistic* programming

High level e.g. infinite dimensional systems
higher-order functions
abstract types

Low level e.g. bets, frequencies, decisions
Monte Carlo simulation

	ML / stats apps	Foundational
High level	✓	✓
Low level	✓	✓

Programming language foundations for statistics

1. **Quick look at probabilistic programming for statistics**

   example; discussion; ***Monte Carlo***

2. Function spaces ...

3. ... and understanding them.

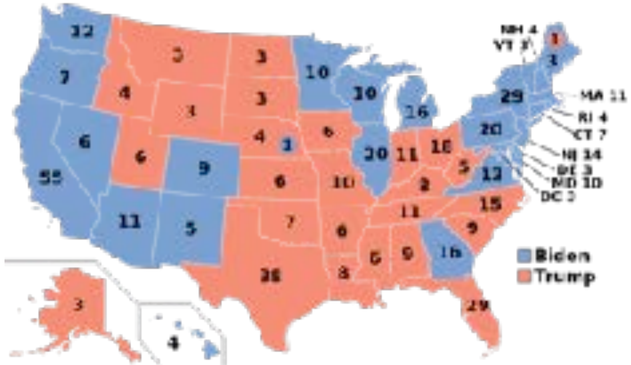
4. Symmetries

High level view: poll example

A very simple model deducing chance of win from poll.

Question:
A quick poll gives 51:49 votes. What is the chance of winning?
Answer: 0.579.

Simon Walker / HM Treasury & Simon Dawson / No10 Downing Street



(See Andrew Gelman and coauthors for a proper discussion of using PPL for election modelling.)

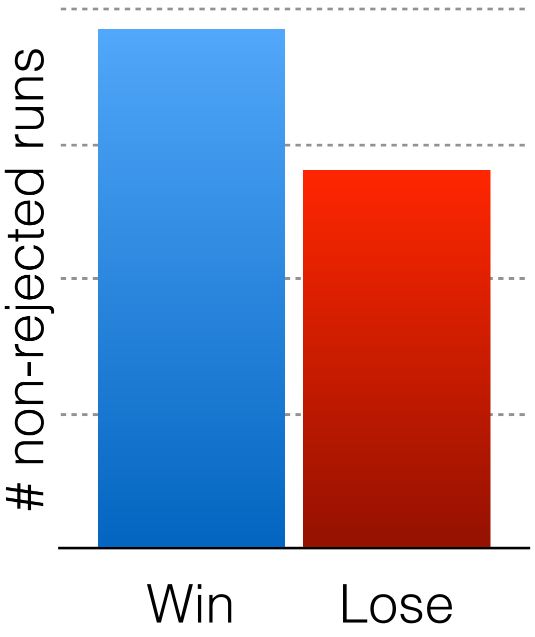
High level view: poll example

A very simple model deducing chance of win from poll.

Question:
A quick poll gives 51:49 votes. What is the chance of winning?
Answer: 0.579.

Crude **rejection sampling** Monte Carlo:

- Run 1000000s of times, each time getting (poll result, win?)
- Reject the runs that mis-predict poll
- What proportion of the remainder are winners?



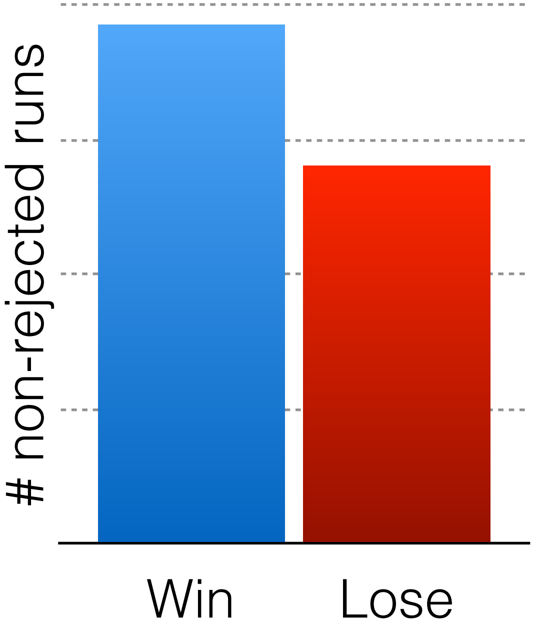
Towards weighted sampling

A very simple model deducing chance of win from poll.

```
model :: Prob ([Bool] , Bool)
model = do
  voteShare <- uniform 0 1
  votes <- repeat (bernoulli voteShare)
  return (take 100 votes , (voteShare > 0.5))
```

Crude **rejection sampling** Monte Carlo:

- Run 1000000s of times, each time getting (poll result, win?)
- Reject the runs that mis-predict poll
- What proportion of the remainder are winners?



Towards weighted sampling

A very simple model deducing chance of win from poll.

```
model :: Prob ([Bool] , Bool)
model = do
  voteShare <- uniform 0 1
  votes <- repeat (bernoulli voteShare)
  return (take 100 votes , (voteShare > 0.5))
```

Weighted sampling

```
model :: Prob ([Bool] , Bool)
model = do
  voteShare <- uniform 0 1
  forM poll (\actualVote->
      score (bernoulliPdf voteShare actualVote))
  return (voteShare > 0.5)
```

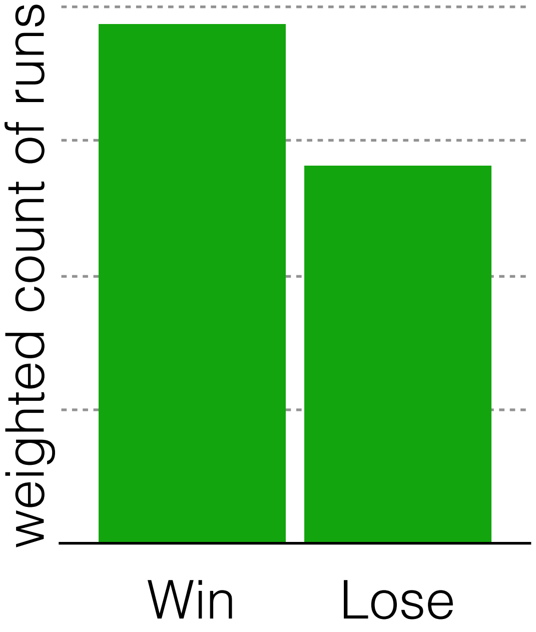
$$likelihood(v) = v^{51}(1 - v)^{49}$$

Weighted sampling

```
model :: Prob ([Bool] , Bool)
model = do
  voteShare <- uniform 0 1
  forM poll (\actualVote->
      score (bernoulliPdf voteShare actualVote))
  return (voteShare > 0.5)
```

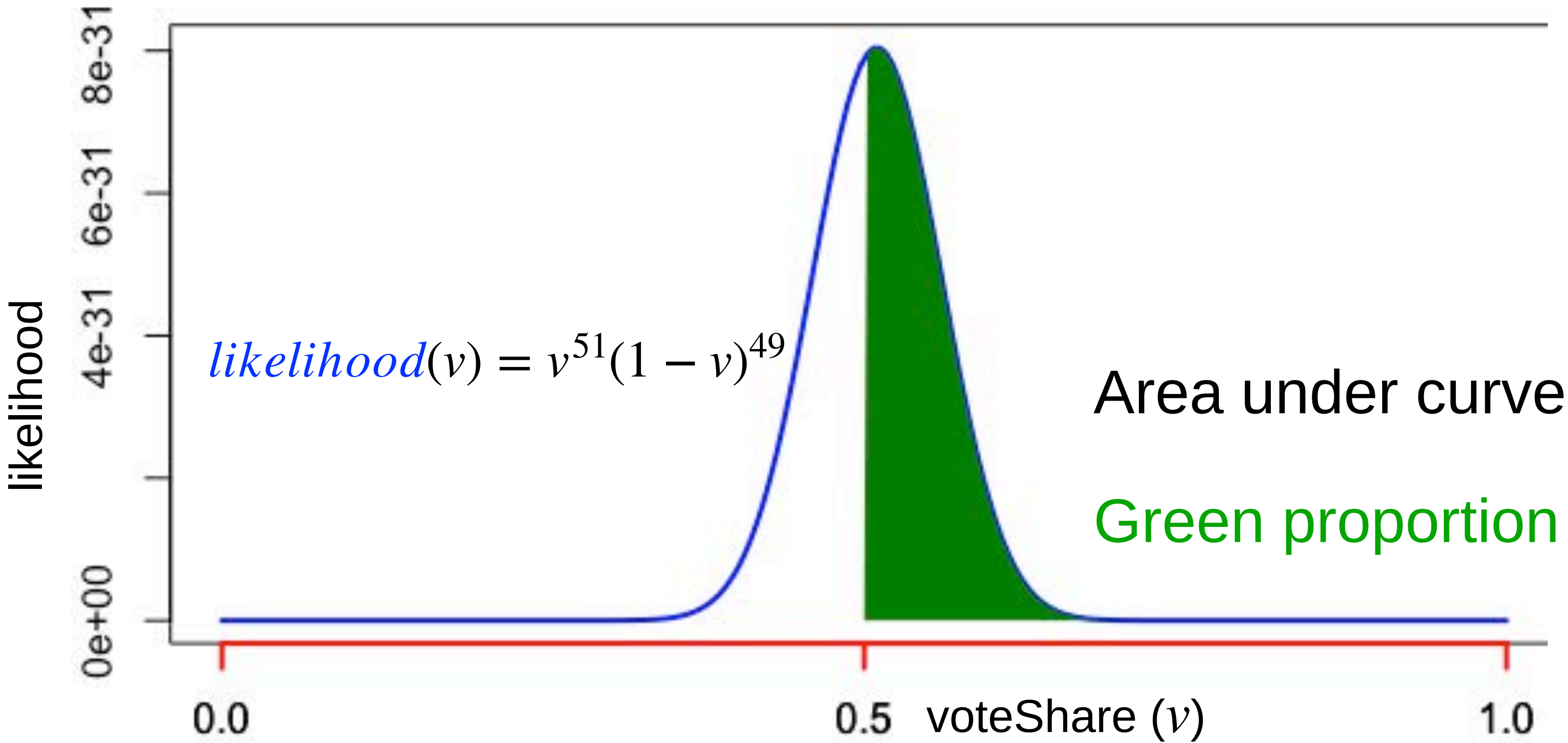
Weighted Monte Carlo:

- Run 1000000s of times, each time getting (win?)
- Each time pick a voteShare, and weight by the likelihood.
- Find weighted proportion of winners.



Weighted sampling

```
model :: Prob ([Bool] , Bool)
model = do
  voteShare <- uniform 0 1
  forM poll (\actualVote->
      score (bernoulliPdf voteShare actualVote))
  return (voteShare > 0.5)
```



$\textit{likelihood}(v) = v^{51}(1-v)^{49}$

Area under curve = $\frac{49!\,51!}{101!}$

Green proportion (win) ≈ 0.579

Programming language foundations for statistics

1. Quick look at
   probabilistic programming for statistics

2. **Function spaces ...**
   Examples ; higher-order functions

3. ... and understanding them.

4. Symmetries

Abstraction in *probabilistic* programming

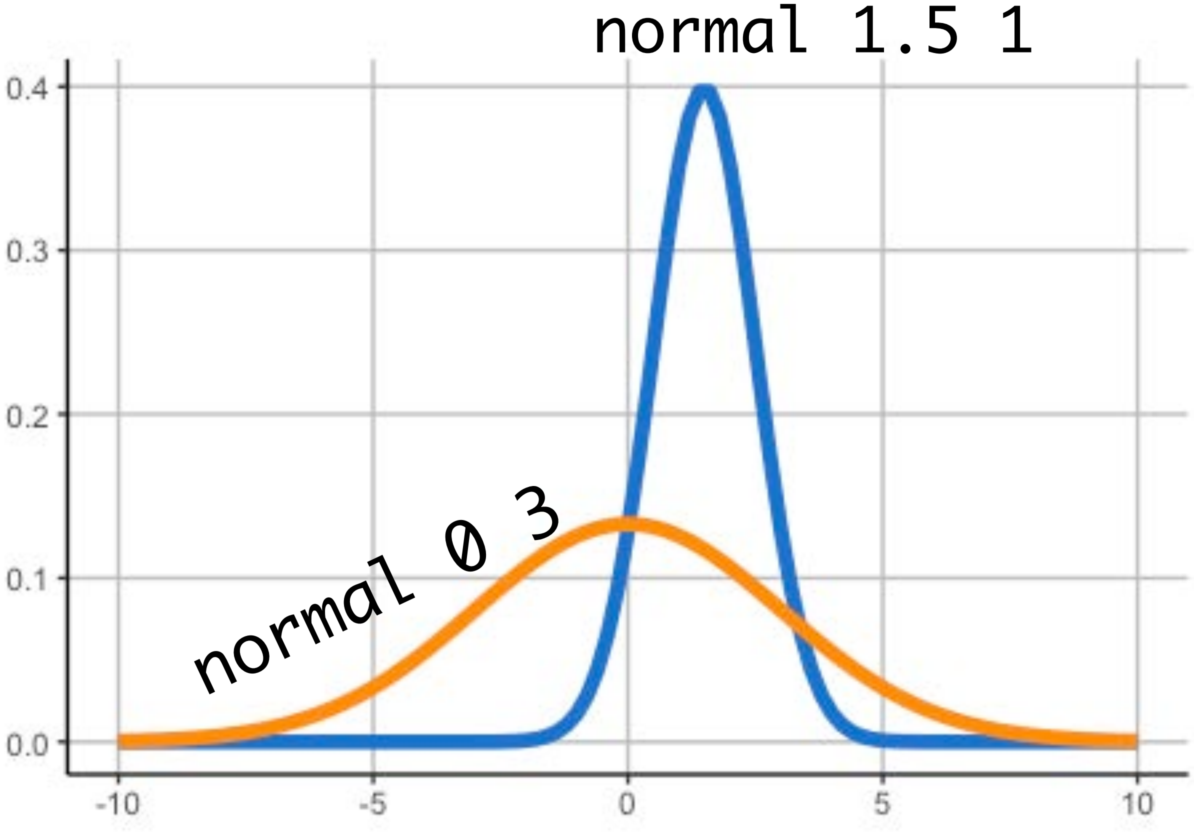
High level e.g. infinite dimensional systems
higher-order functions
abstract types

Low level e.g. bets, frequencies, decisions
Monte Carlo simulation

	ML / stats apps	Foundational
High level	✓	✓
Low level	✓	✓

Random linear functions

```
randlinear :: Prob (RealNum , RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     return (a,b)
```



```
type RealNum = Double
```

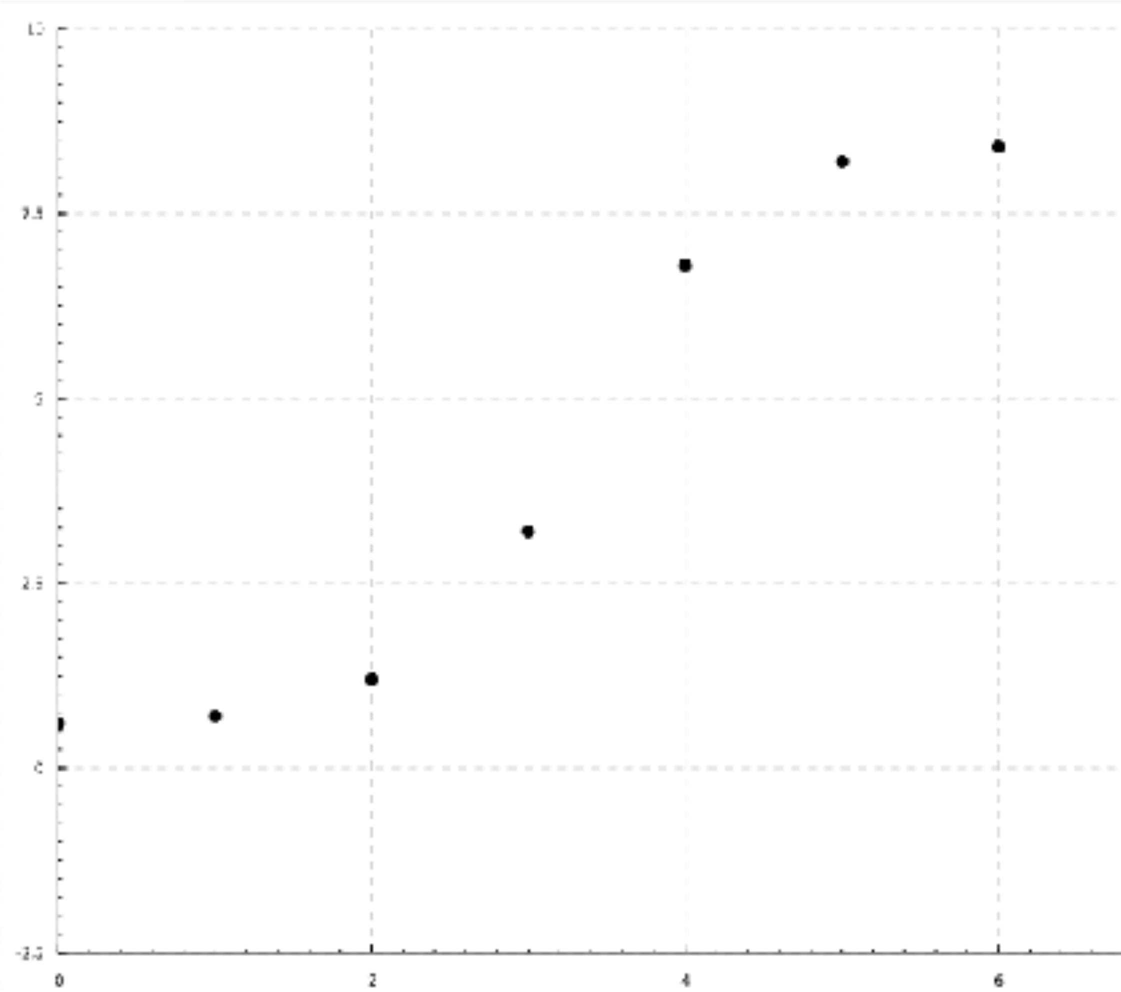
lazyppl.bitbucket.io

Random linear functions

```
randlinear :: Prob (RealNum , RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     return (a,b)
```

We will use this for a regression problem:

which function probably generated these points?

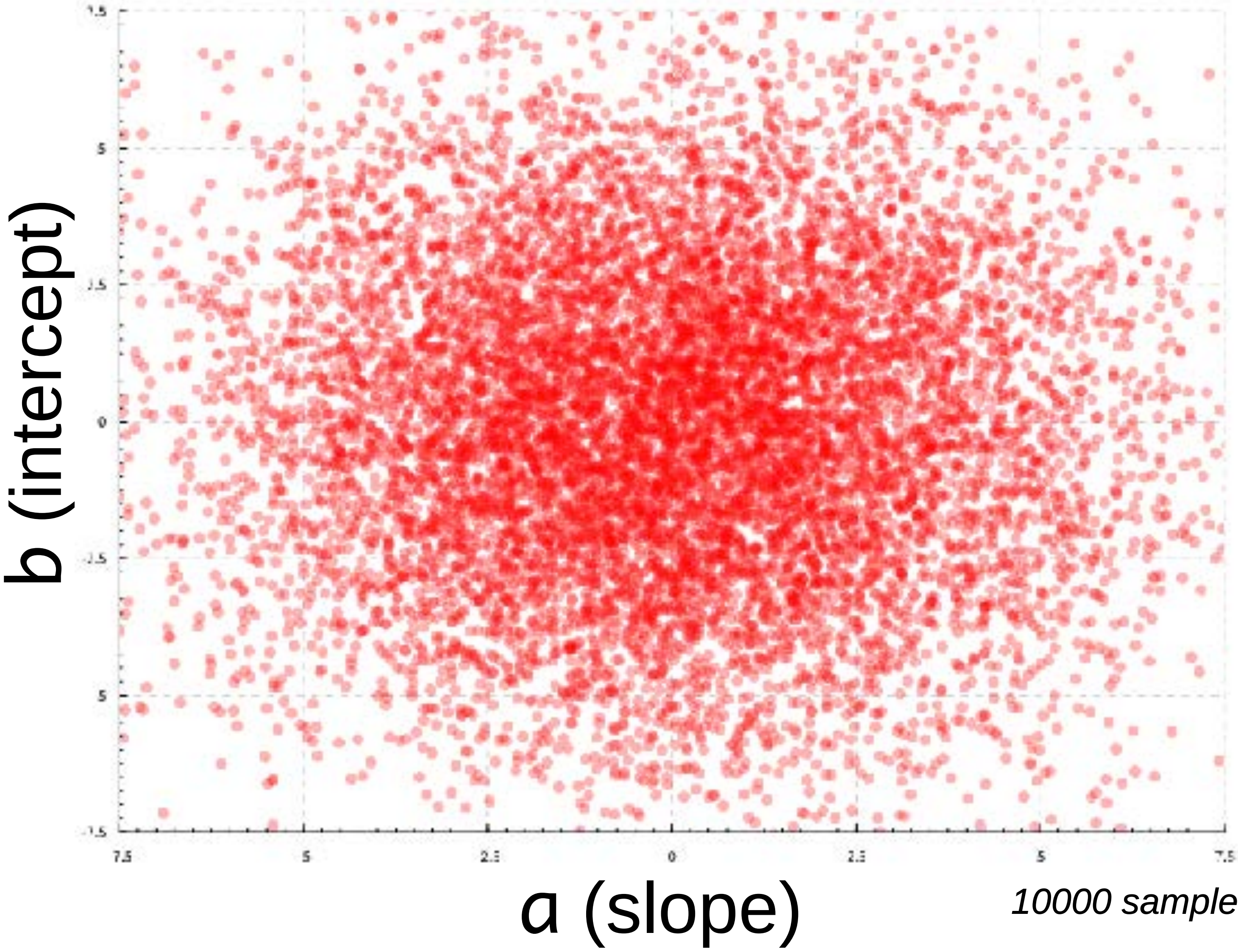


```
type RealNum = Double
```

lazyppl.bitbucket.io

Random linear functions

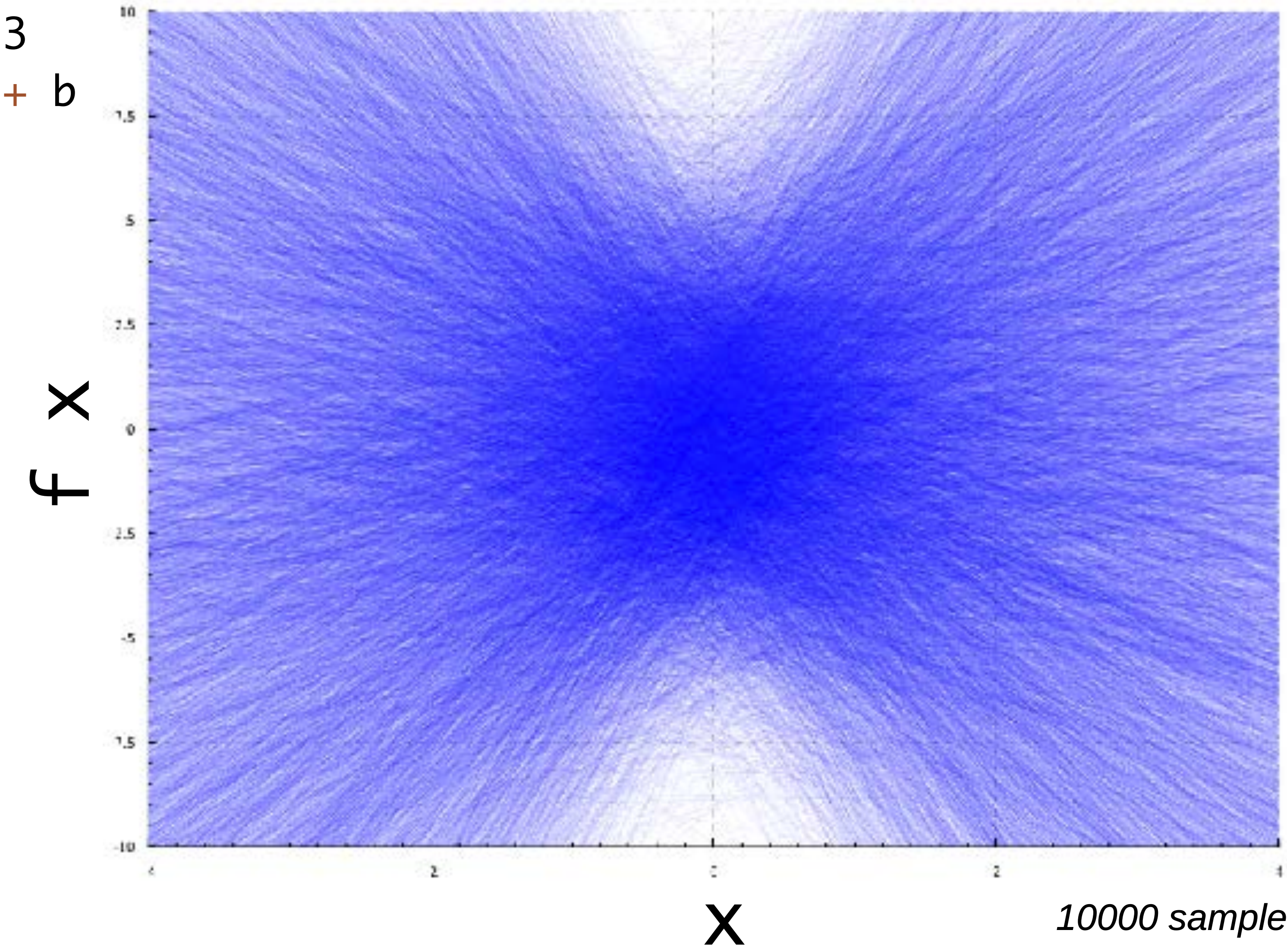
```
randlinear :: Prob (RealNum , RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     return (a,b)
```



10000 samples

Random linear functions

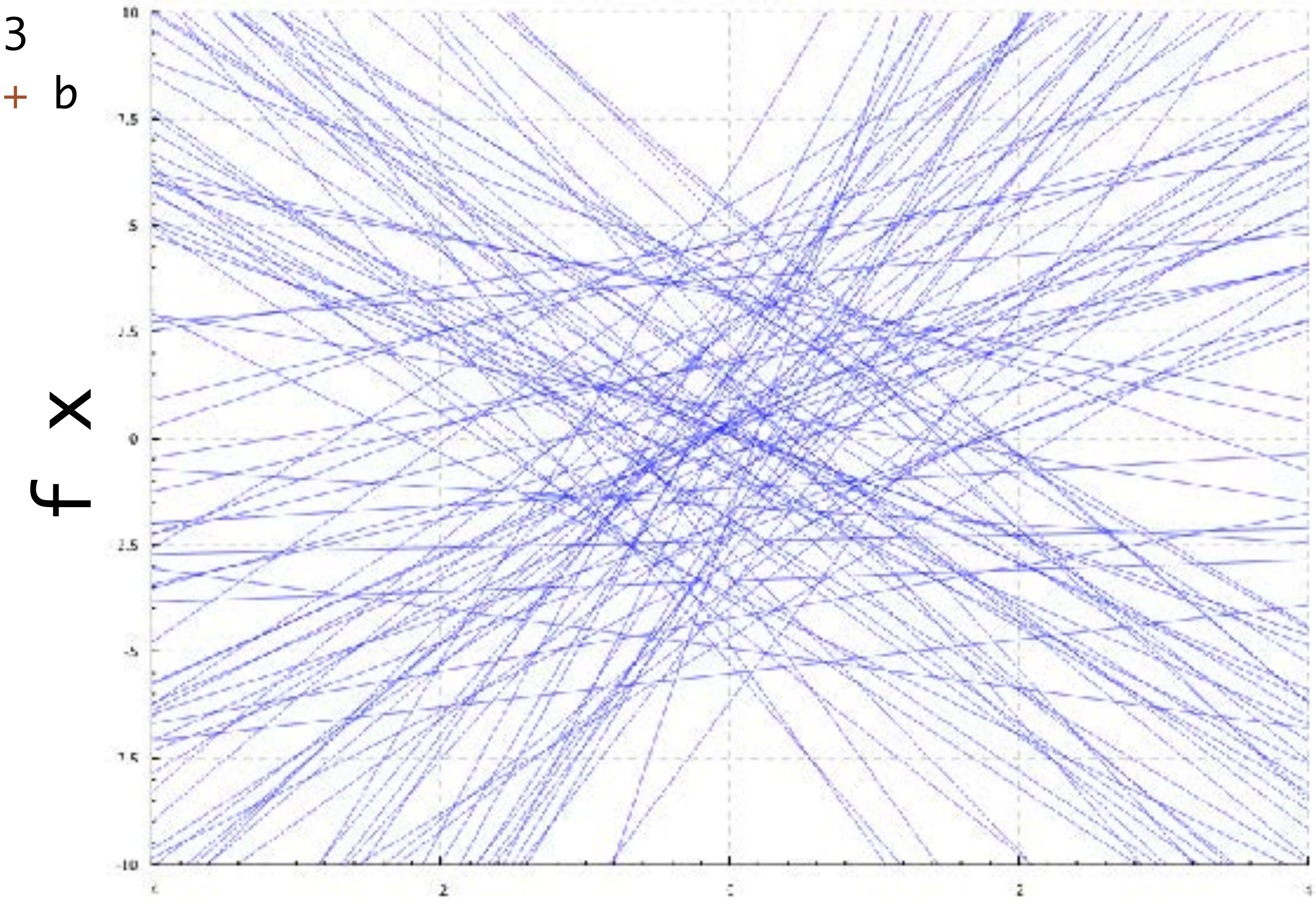
```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```



10000 samples

Random linear functions

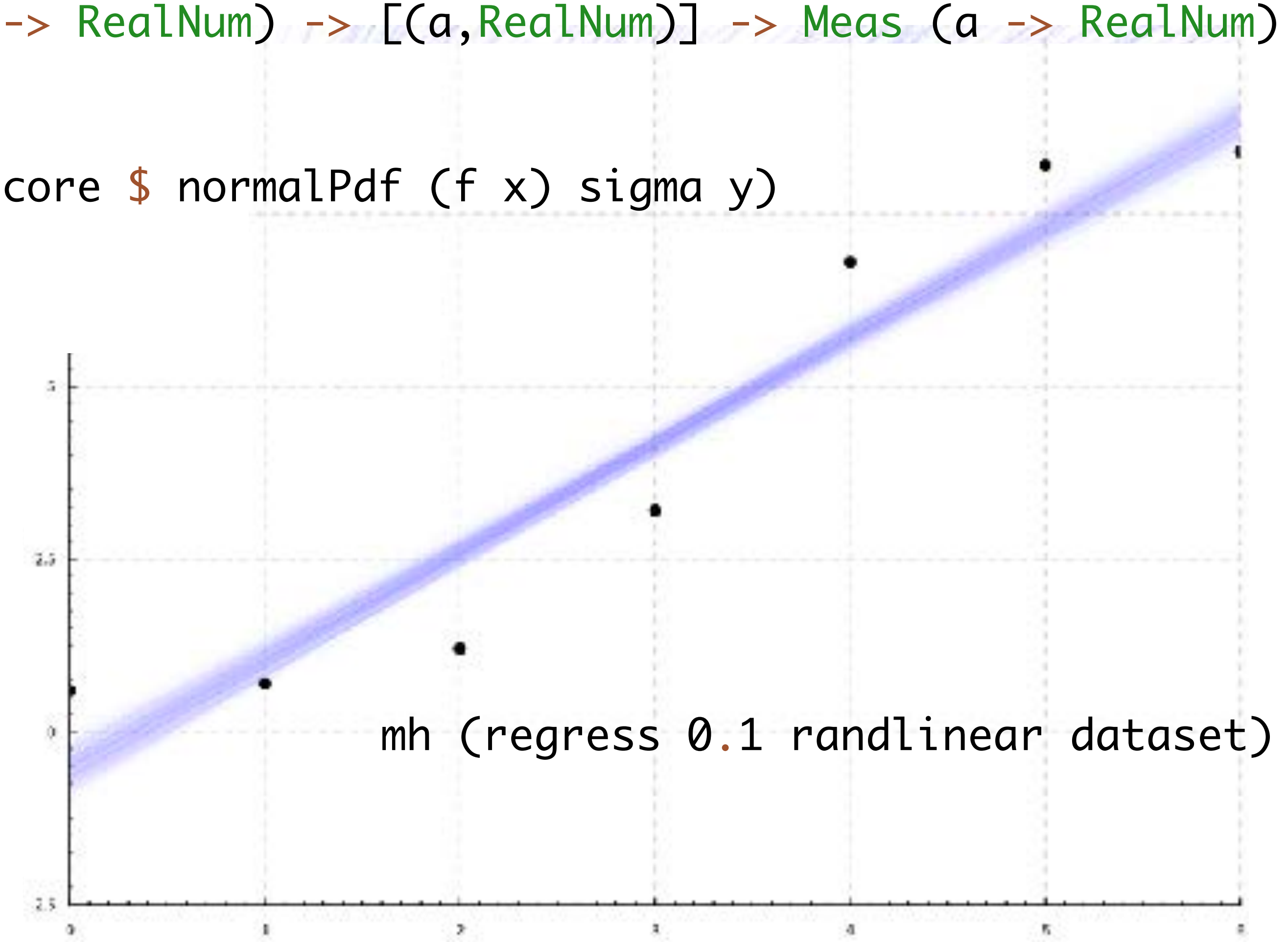
```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```



100 samples

Bayesian regression

```
randlinear :: Prob (RealNum -> RealNum)


regress :: RealNum -> Prob (a -> RealNum) -> [(a,RealNum)] -> Meas (a -> RealNum)
regress sigma prior dataset =
  do f <- sample prior
     forM dataset (\(x,y) -> score $ normalPdf (f x) sigma y)
     return f
```



`mh (regress 0.1 randlinear dataset)`

lazyppl *includes a type*

`Meas a`

of unnormalized measures and

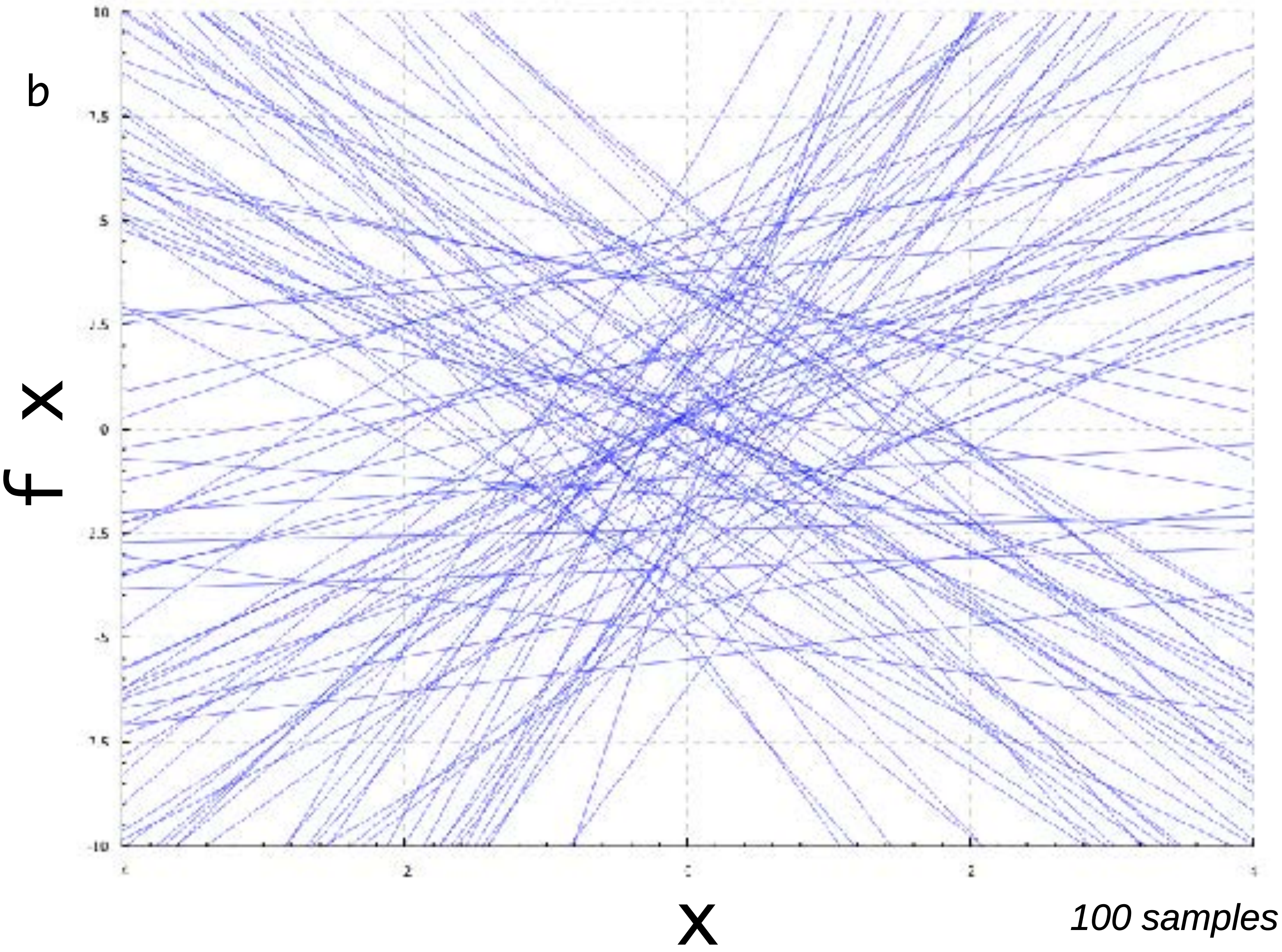
`mh`

a Metropolis-Hastings inference method.

lazyppl.bitbucket.io

Random linear functions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```



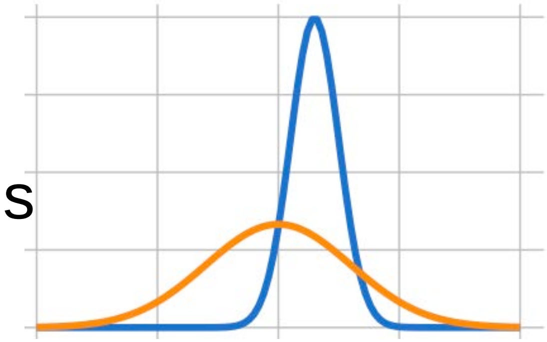
100 samples

Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…

- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)
- `Prob Bool` contains probability distributions like `bernoulli 0.5`

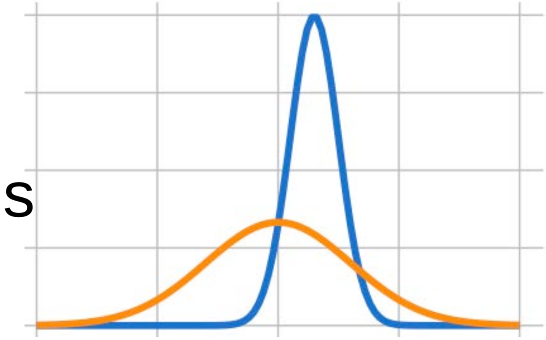


Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…

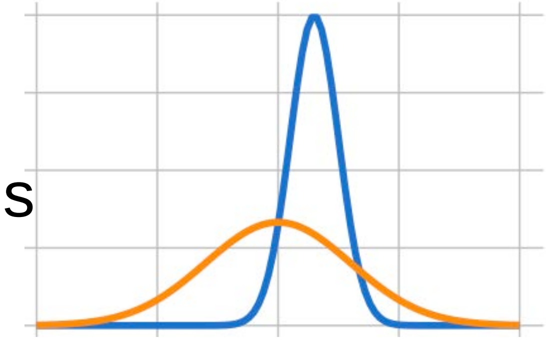
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)



lazyppl.bitbucket.io

Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```



There's a type constructor `Prob` (a monad), and…

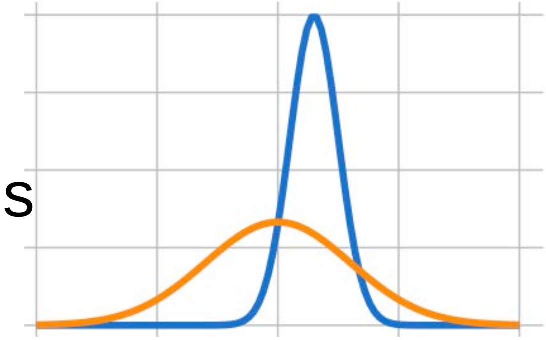
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)
- `normal :: RealNum -> RealNum -> Prob RealNum` is a parameterized distribution
- `bernoulli :: RealNum -> Prob Bool` is a parameterized distribution too

lazyppl.bitbucket.io

Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There’s a type constructor `Prob` (a monad), and…

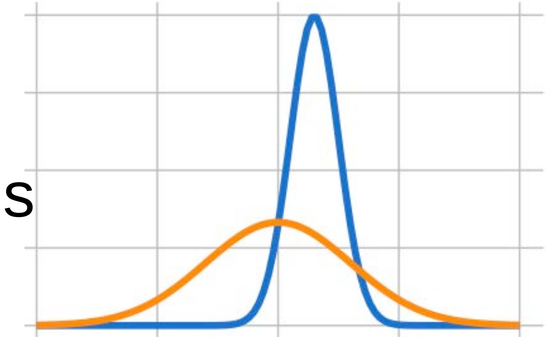


- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)

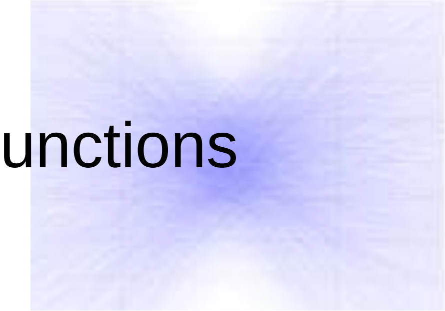
Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…



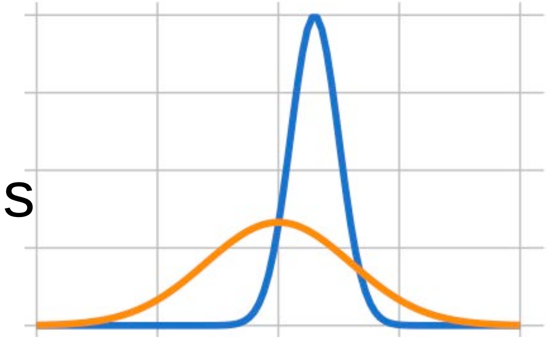
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)



lazyppl.bitbucket.io

Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
            :: RealNum -> RealNum
```



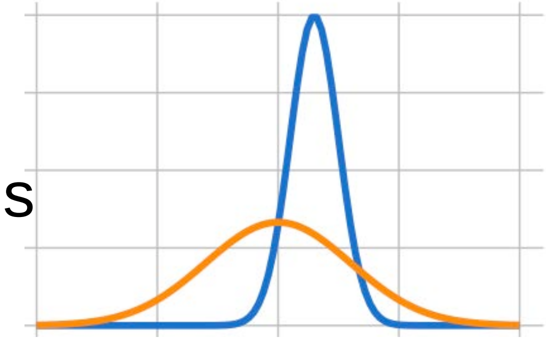
There’s a type constructor `Prob` (a monad), and…

- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)

Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
  :: Prob (RealNum -> RealNum)
```

There's a type constructor `Prob` (a monad), and…

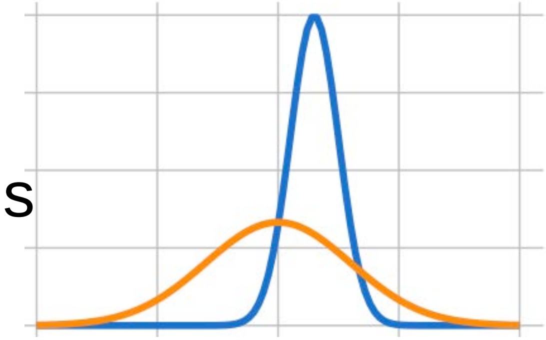


- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)

Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…



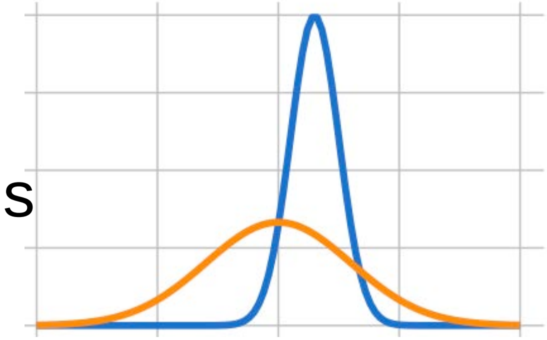
- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)

lazyppl.bitbucket.io

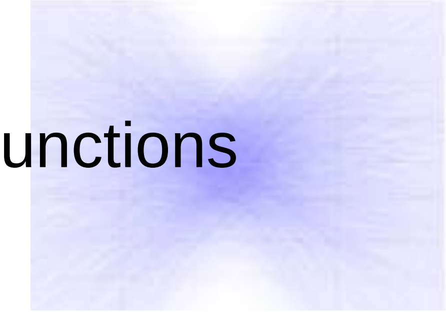
Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

There's a type constructor `Prob` (a monad), and…



- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)



- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)
- `Prob (Prob Bool)` contains random distributions, etc..

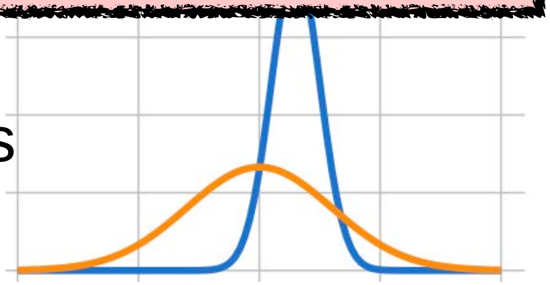
Types as spaces of distributions

```
randlinear :: Prob (RealNum -> RealN
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

Challenge:
Aumann (1961) showed that measure-theoretic probability does not support function spaces properly!

There's a type constructor `Prob` (a monad), and...

- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)



- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)

- `Prob (Prob Bool)` contains random distributions, etc..

Random functions & program synthesis

```
data Expr = Var | Constt RealNum | Add Expr Expr | Mult Expr Expr
          | IfLess RealNum Expr Expr

eval :: Expr -> (RealNum -> RealNum)
eval Var x = x
eval (Constt r) _ = r
eval (Add e1 e2) x = (eval e1 x) + (eval e2 x)
eval (Mult e1 e2) x = (eval e1 x) * (eval e2 x)
eval (IfLess r e1 e2) x = if x < r then eval e1 x else eval e2 x
```

Random functions & program synthesis

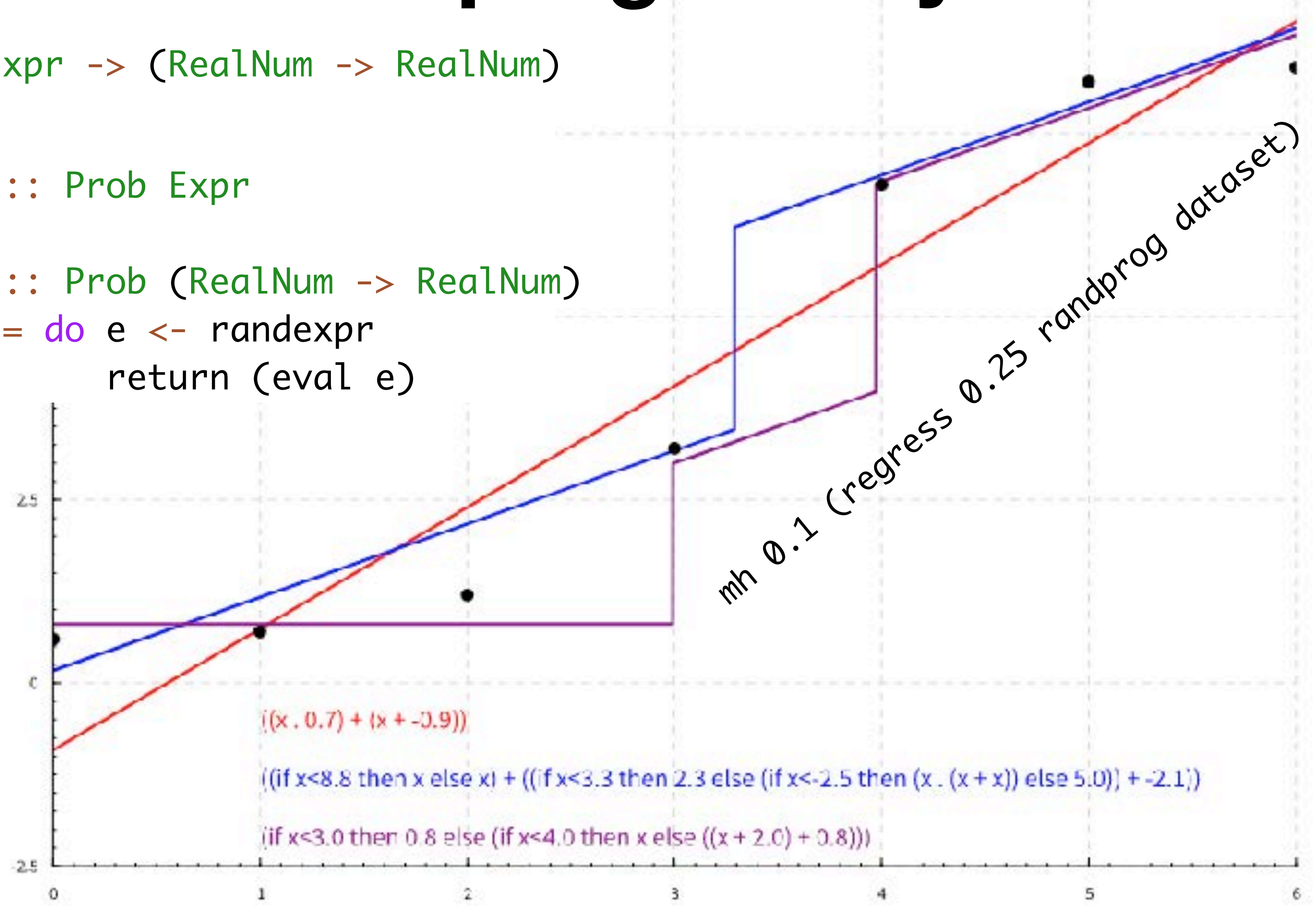
```
data Expr = Var | Constt RealNum | Add Expr Expr | Mult Expr Expr
          | IfLess RealNum Expr Expr

eval :: Expr -> (RealNum -> RealNum)
eval Var x = x
eval (Constt r) _ = r
eval (Add e1 e2) x = (eval e1 x) + (eval e2 x)
eval (Mult e1 e2) x = (eval e1 x) * (eval e2 x)
eval (IfLess r e1 e2) x = if x < r then eval e1 x else eval e2 x


randexpr :: Prob Expr

randprog :: Prob (RealNum -> RealNum)
randprog = do e <- randexpr
              return (eval e)
```

Random functions & program synthesis

```
data Expr = ...

eval :: Expr -> (RealNum -> RealNum)


randexpr :: Prob Expr

randprog :: Prob (RealNum -> RealNum)
randprog = do e <- randexpr
              return (eval e)
```

Random functions & program synthesis

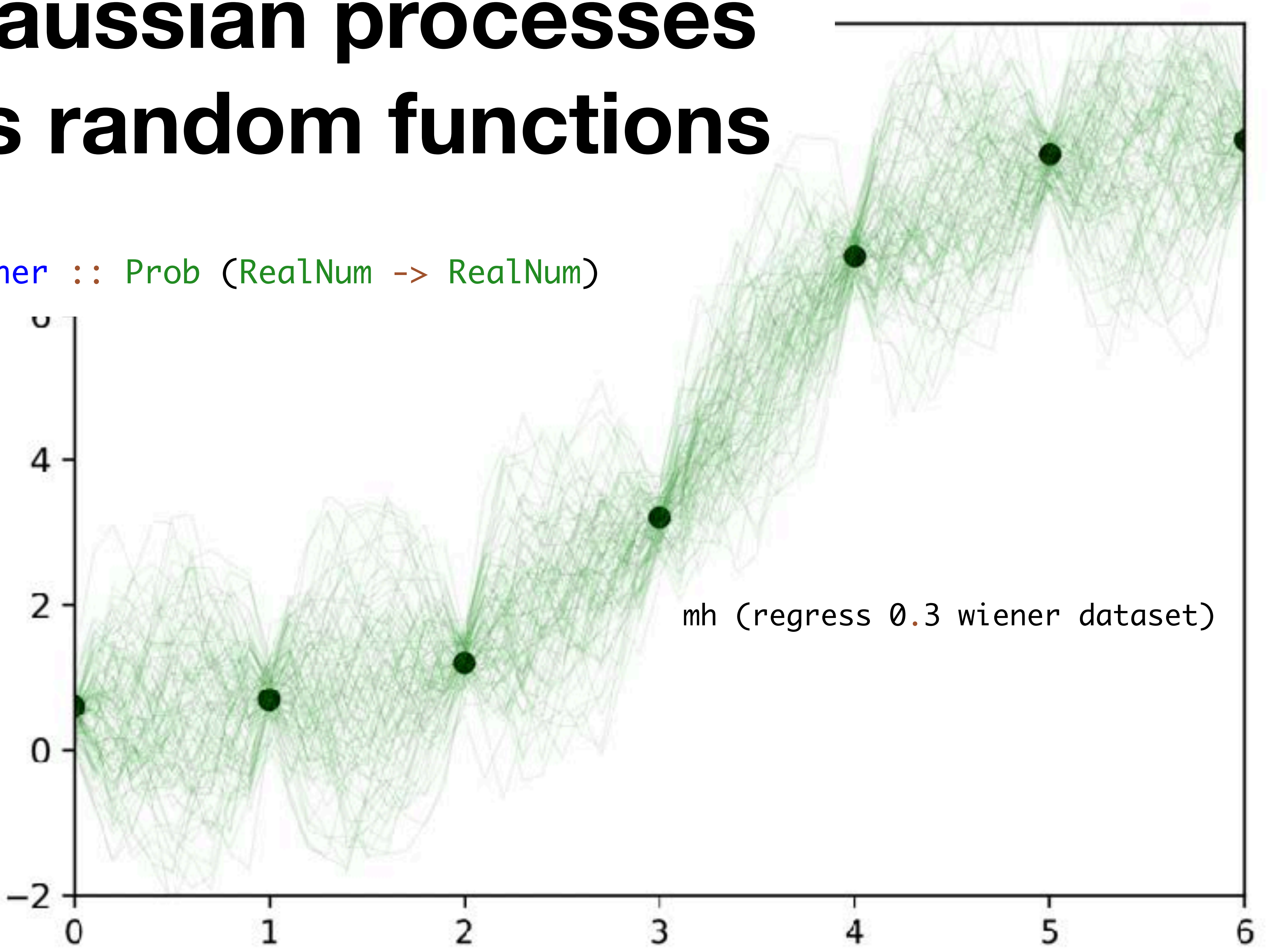
```
data Expr = ...

eval :: Expr -> (RealNum -> RealNum)


randexpr :: Prob Expr

randprog :: Prob (RealNum -> RealNum)
randprog = do e <- randexpr
              return (eval e)
```



```
mh 0.1 (regress 0.25 randprog dataset)
```

Gaussian processes as random functions

```
wiener :: Prob (RealNum -> RealNum)
```

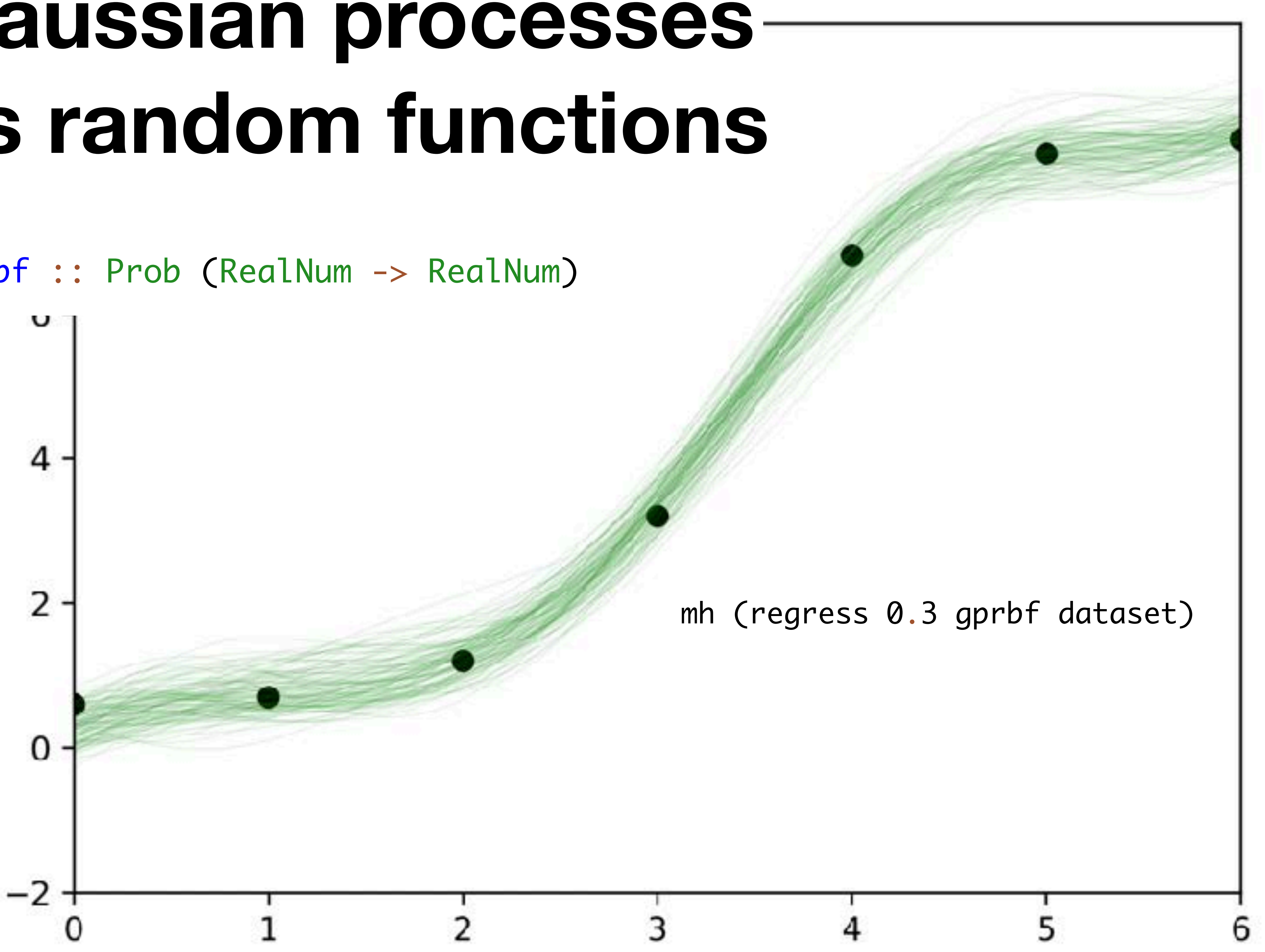


```
mh (regress 0.3 wiener dataset)
```

lazyppl.bitbucket.io

Gaussian processes as random functions

```
gprbf :: Prob (RealNum -> RealNum)
```



```
mh (regress 0.3 gprbf dataset)
```

lazyppl.bitbucket.io

Programming language foundations for statistics

1. Quick look at
   probabilistic programming for statistics

2. **Function spaces ...**
   *Examples ; **higher-order functions***

3. ... and understanding them.

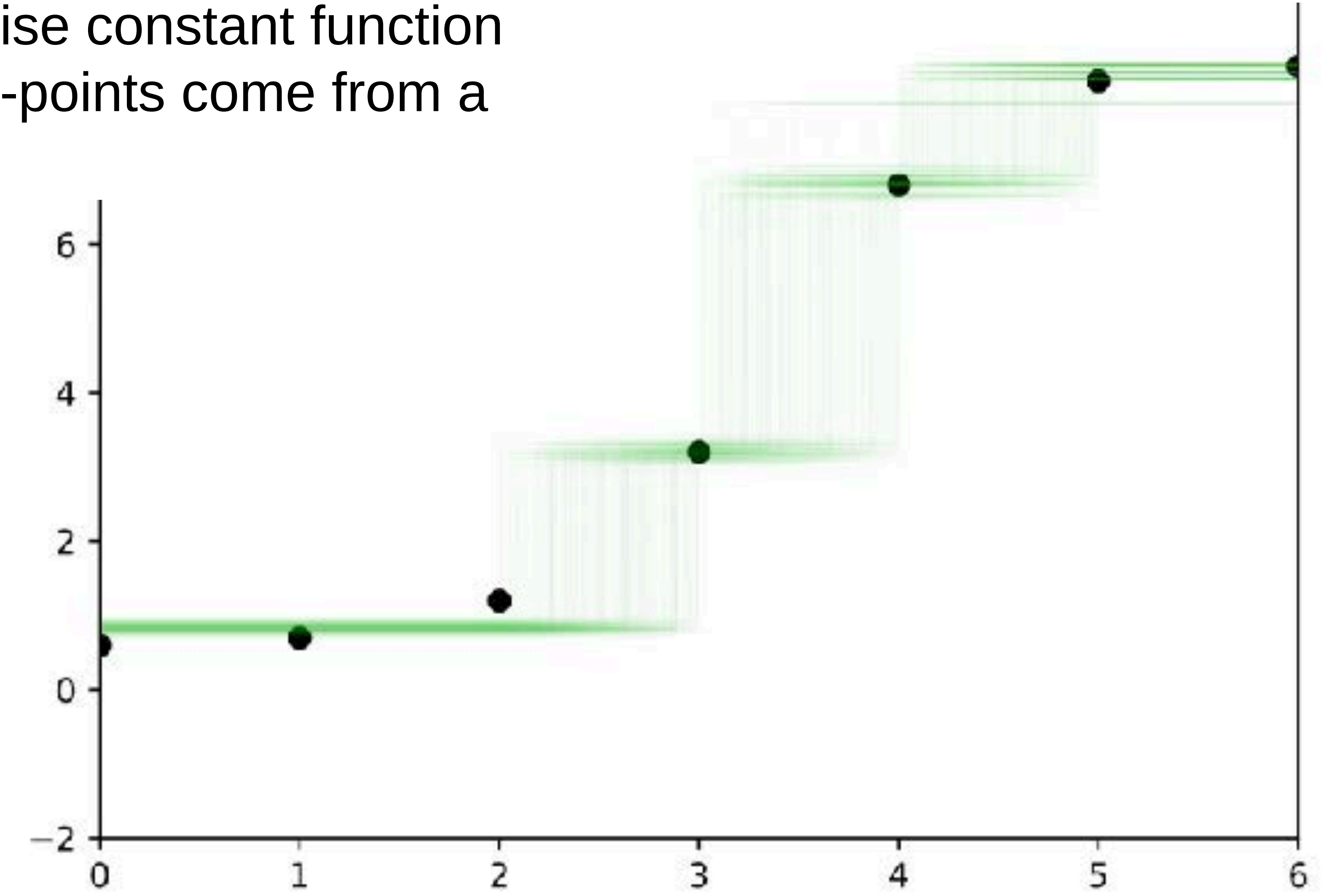
4. Symmetries

Piecewise constant regression

Defn. A *point process* on a is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

Dash, Staton. ACT 2020.

Idea: Fit a piecewise constant function where the change-points come from a point process.

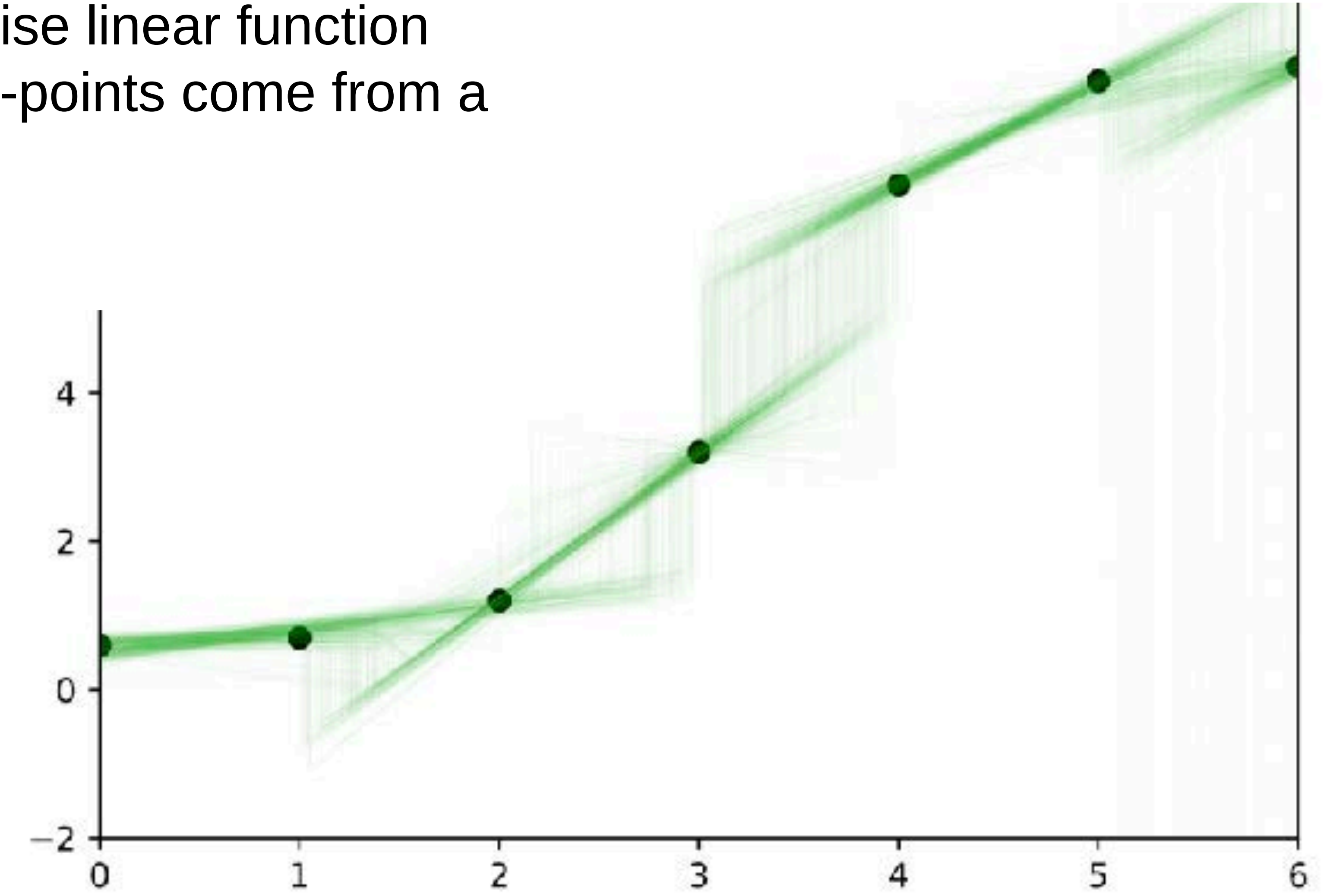


Piecewise linear regression

Defn. A *point process* on `a` is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

Dash, Staton. ACT 2020.

Idea: Fit a piecewise linear function where the change-points come from a point process.



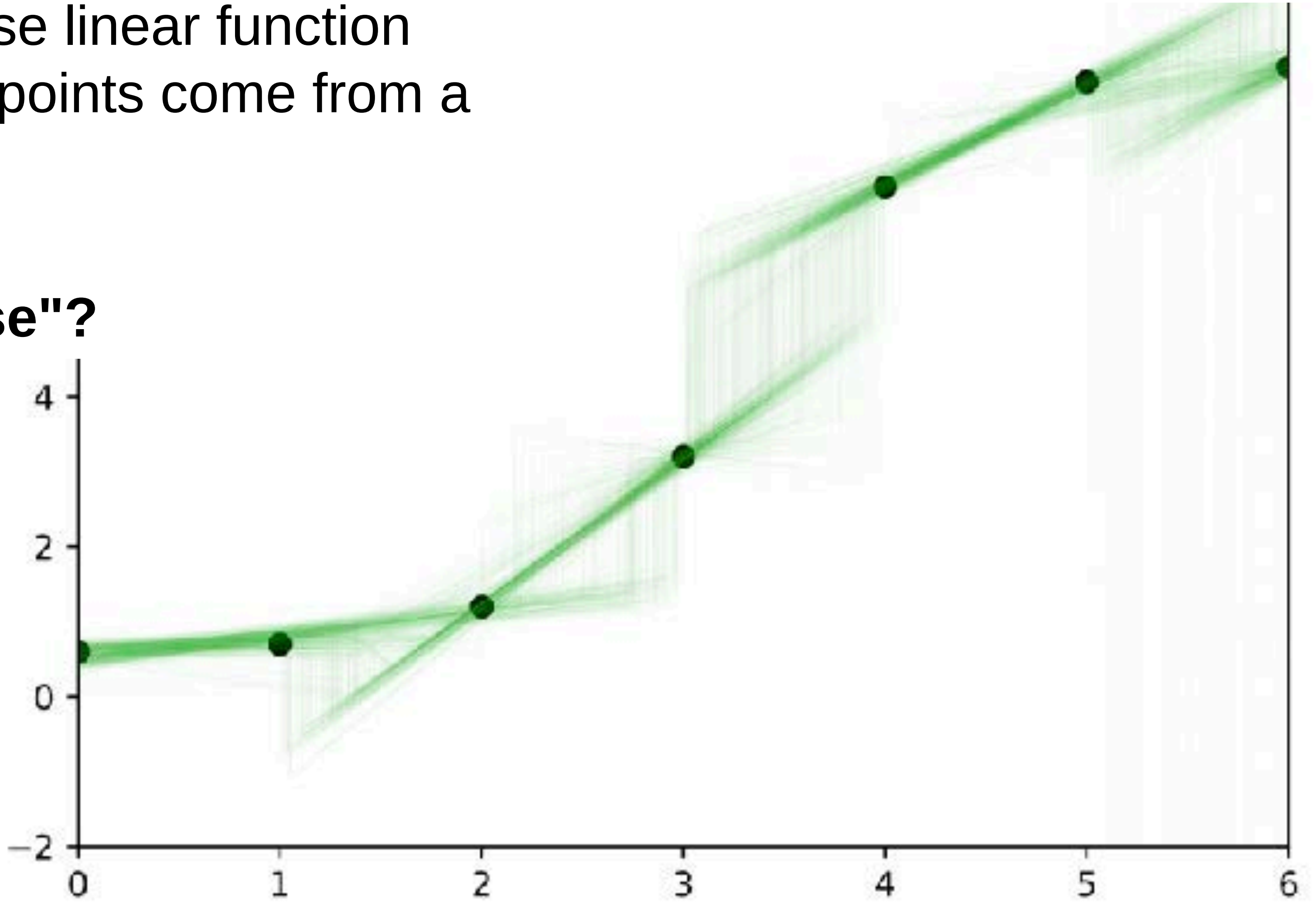
Piecewise linear regression

Defn. A *point process* on `a` is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

Dash, Staton. ACT 2020.

Idea: Fit a piecewise linear function where the change-points come from a point process.

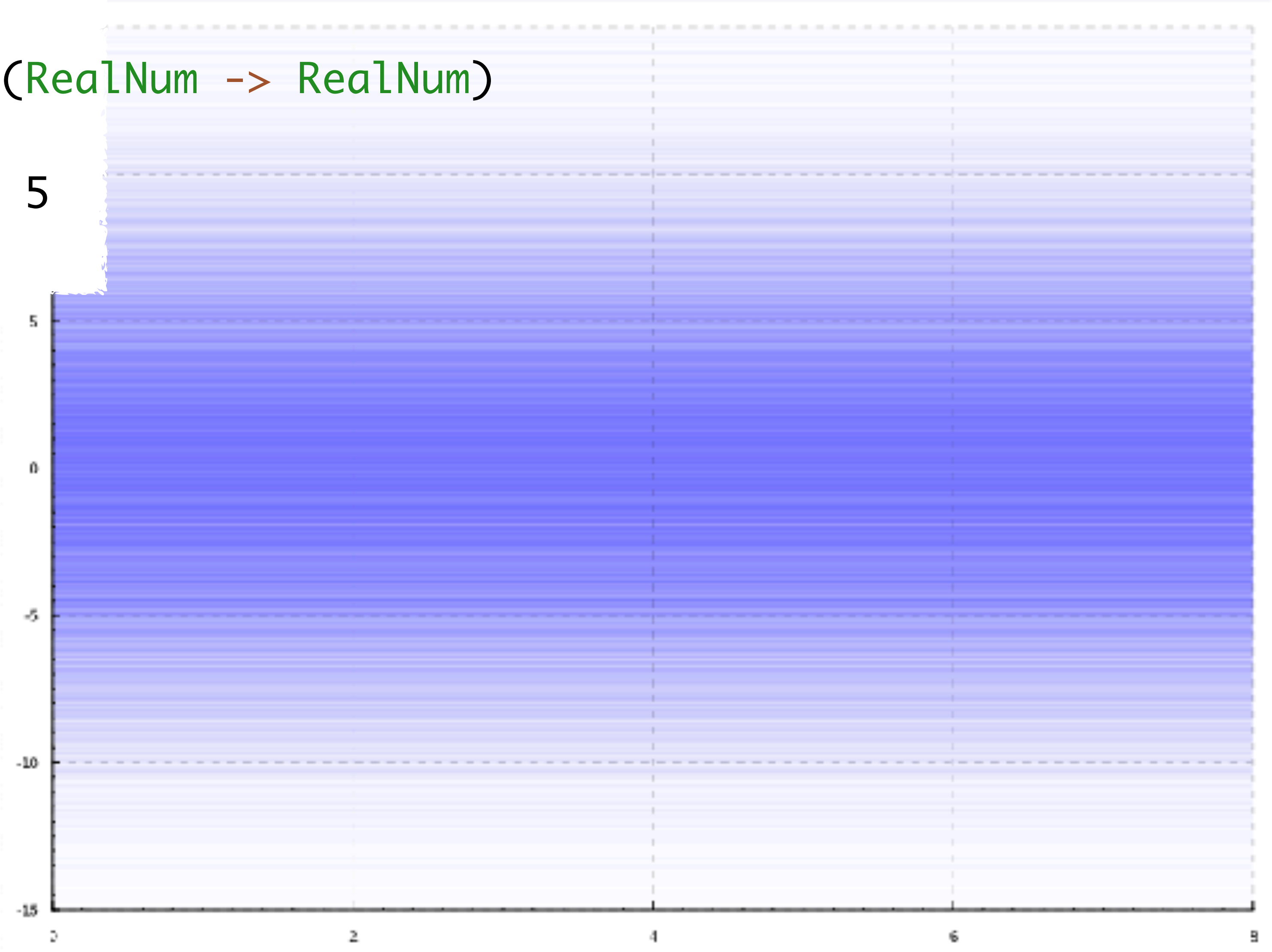
What is "piecewise"?



Piecewise constant regression

Defn. A *point process* on a is an inhabitant of Prob [a] (or Prob (Bag a)).

```
randconst :: Prob (RealNum -> RealNum)
randconst =
  do a <- normal 0 5
     let f x = a
     return f
```



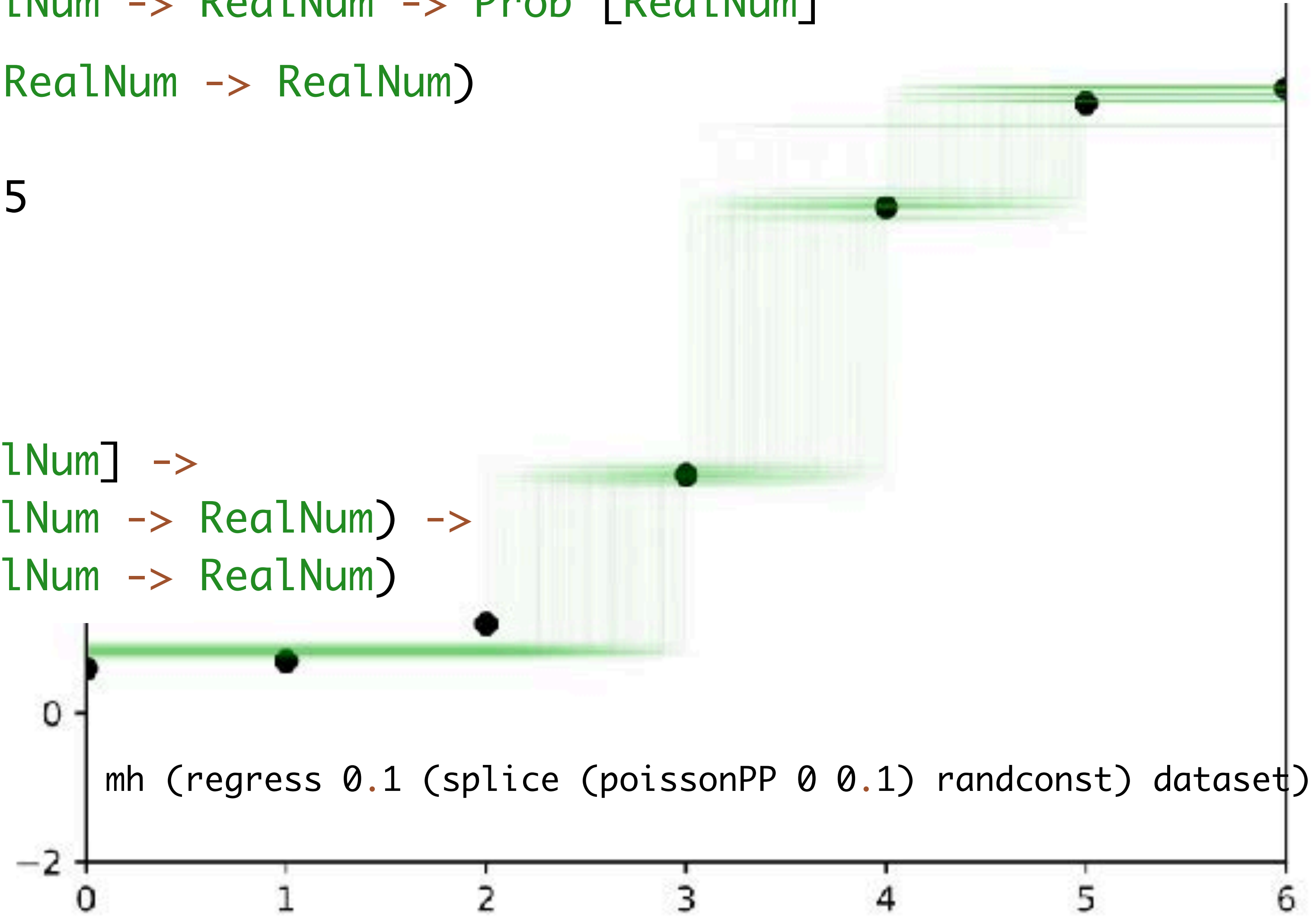
Piecewise constant regression

Defn. A *point process* on `a` is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

e.g. `poissonPP :: RealNum -> RealNum -> Prob [RealNum]`

```
randconst :: Prob (RealNum -> RealNum)
randconst =
  do a <- normal 0 5
     let f x = a
     return f
```

```
splice :: Prob [RealNum] ->
          Prob (RealNum -> RealNum) ->
          Prob (RealNum -> RealNum)
```



```
mh (regress 0.1 (splice (poissonPP 0 0.1) randconst) dataset)
```

lazyppl.bitbucket.io

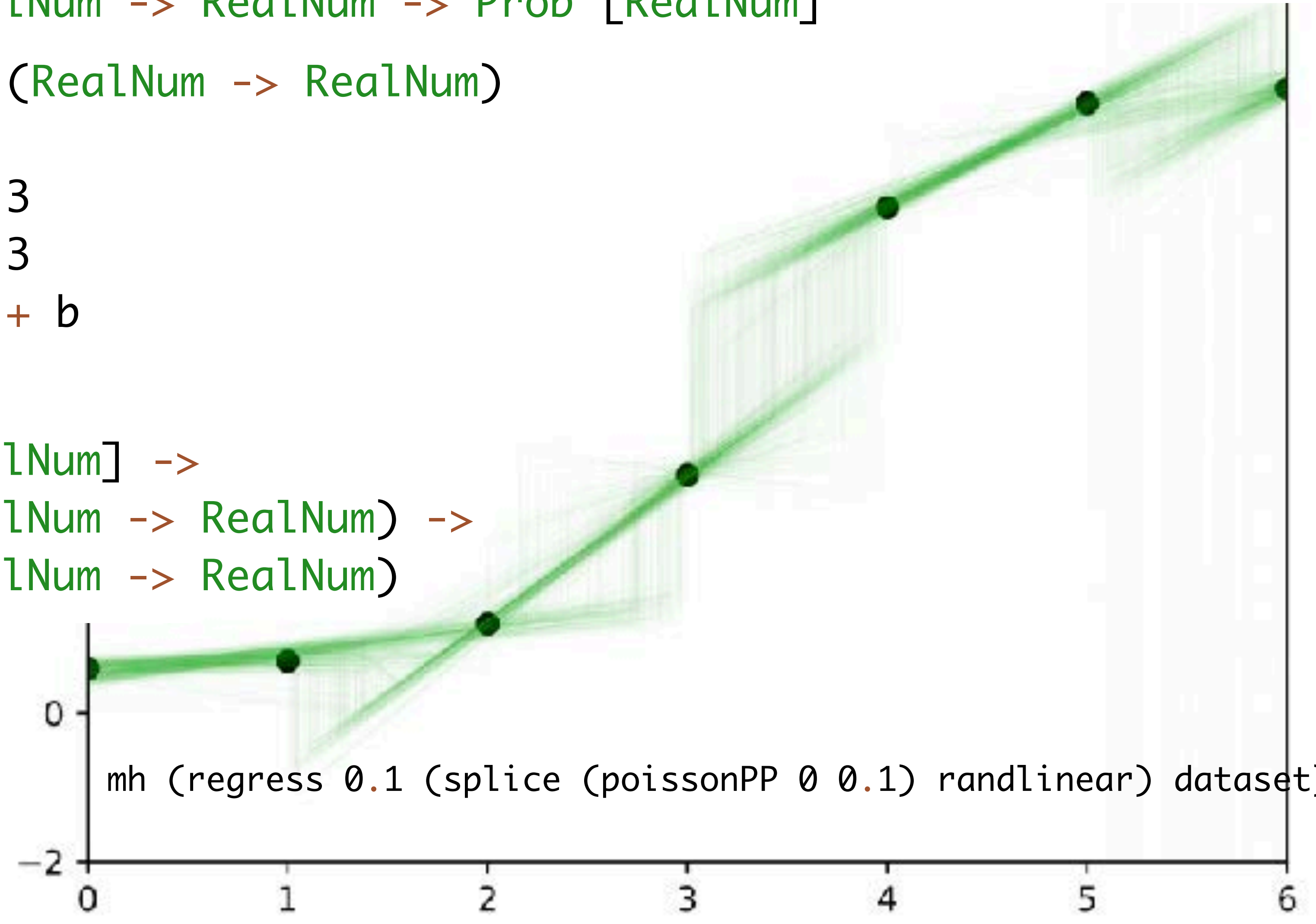
Piecewise constant regression

Defn. A *point process* on `a` is an inhabitant of `Prob [a]` (or `Prob (Bag a)`).

e.g. `poissonPP :: RealNum -> RealNum -> Prob [RealNum]`

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 3
     b <- normal 0 3
     let f x = a*x + b
     return f
```

```
splice :: Prob [RealNum] ->
          Prob (RealNum -> RealNum) ->
          Prob (RealNum -> RealNum)
```



```
mh (regress 0.1 (splice (poissonPP 0 0.1) randlinear) dataset
```

lazyppl.bitbucket.io

Programming language foundations for statistics

1. Quick look at
   probabilistic programming for statistics

2. Function spaces ...

3. **... and understanding them.**
   models in the abstract ; quasi-Borel spaces

4. Symmetries

Curry-Howard correspondence

Programming	Maths	Category theory	Logic
Types	Spaces	Objects	Propositions
Programs	Continuous functions	Morphisms	Proofs

Curry-Howard correspondence

Programming	Maths	Category theory	Logic
Types	Spaces	Objects	Propositions
Programs	Continuous functions	Morphisms	Proofs
Probabilistic programs	Measures	?	?

Desiderata for a theory of Prob

Dataflow property:

Program lines can be ***reordered*** *and* ***discarded***
if dataflow is preserved.

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 2
     b <- normal 0 3
     let f x = a*x + b
     return f
```

Dataflow property:

Program lines can be ***reordered*** *and* ***discarded*** *if dataflow is preserved.*

Desiderata for a theory of Prob

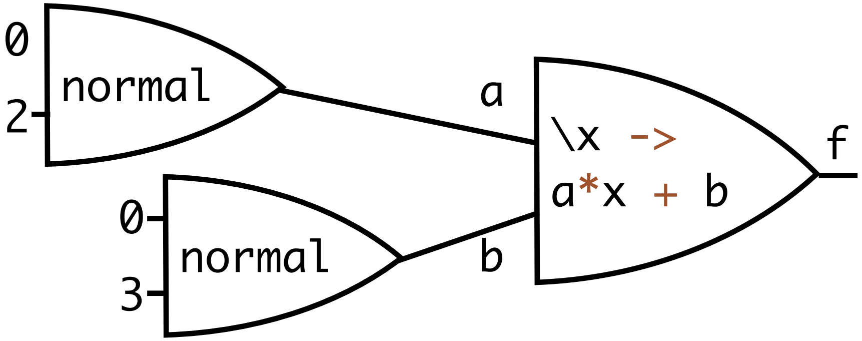
```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 2      do b <- normal 0 3
     b <- normal 0 3         a <- normal 0 2
     let f x = a*x + b       let f x = a*x + b
     return f                return f
```

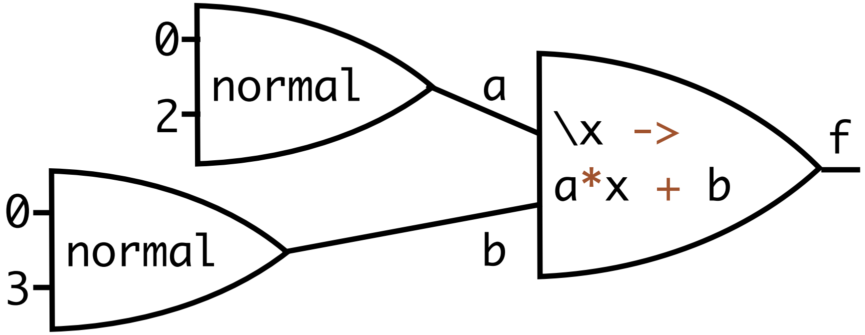
Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 2       do b <- normal 0 3
     b <- normal 0 3          a <- normal 0 2
     let f x = a*x + b        let f x = a*x + b
     return f                 return f
```





Dataflow property:

Program lines can be ***reordered*** *and* ***discarded*** *if dataflow is preserved.*

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 2
     b <- normal 0 3
     let f x = a*x + b
     return f
```

```
do b <- normal 0 3
   a <- normal 0 2
   let f x = a*x + b
   return f
```

```
do c <- normal 0 4
   b <- normal 0 3
   a <- normal 0 2
   let f x = a*x + b
   return f
```

Dataflow property:

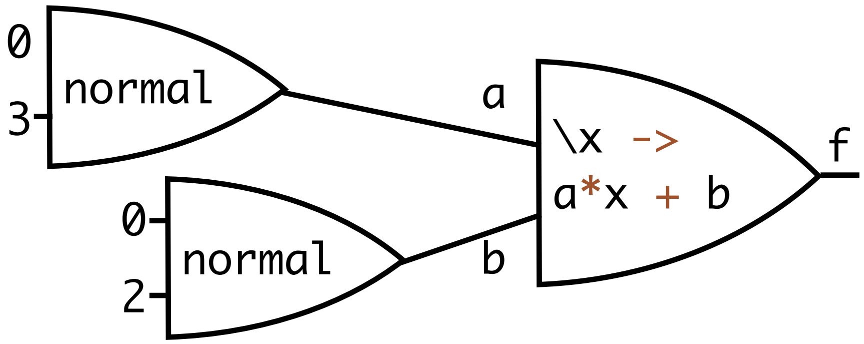
*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

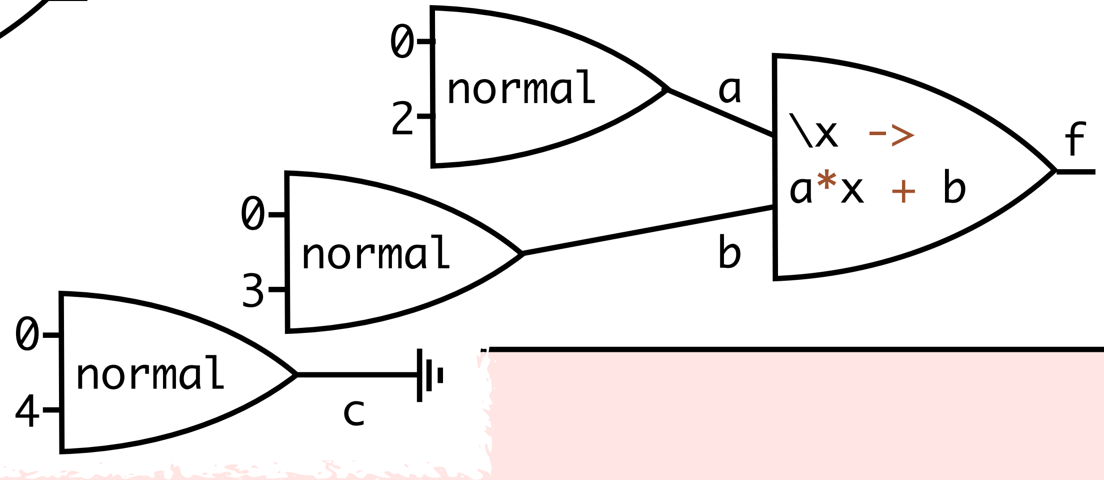
Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 2
     b <- normal 0 3
     let f x = a*x + b
     return f
```

```
do b <- normal 0 3
   a <- normal 0 2
   let f x = a*x + b
   return f
```

```
do c <- normal 0 4
   b <- normal 0 3
   a <- normal 0 2
   let f x = a*x + b
   return f
```





Dataflow property:

Program lines can be ***reordered*** *and* ***discarded*** *if dataflow is preserved.*

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 2        do b <- normal 0 3
     b <- normal 0 3           a <- normal 0 2
     let f x = a*x + b         let f x = a*x + b
     return f                  return f
```

$$\int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}b\,\mathrm{d}a \qquad \int\int k(\lambda x\,.\,ax+b)\boxed{\mathrm{d}a\,\mathrm{d}b}$$

Dataflow property:

Program lines can be ***reordered*** *and* ***discarded*** *if dataflow is preserved.*

Related to Fubini’s theorem.

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 2
     b <- normal 0 3
     let f x = a*x + b
     return f
```

```
do b <- normal 0 3
   a <- normal 0 2
   let f x = a*x + b
   return f
```

```
do c <- normal 0 4
   b <- normal 0 3
   a <- normal 0 2
   let f x = a*x + b
   return f
```

$$\int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}b\,\mathrm{d}a$$

$$\int\int k(\lambda x\,.\,ax+b)\boxed{\mathrm{d}a\,\mathrm{d}b}$$

$$\int\int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}a\,\mathrm{d}b\boxed{\mathrm{d}c}$$

Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Related to Fubini's theorem.

Also related to
Cho & Jacobs MSCS 2019.
Fritz Adv Math 2020.
Kock TAC 2012

Programming language foundations for statistics

1. Quick look at
   probabilistic programming for statistics

2. Function spaces ...

3. **... and understanding them.**
   models in the abstract ; ***quasi-Borel spaces***

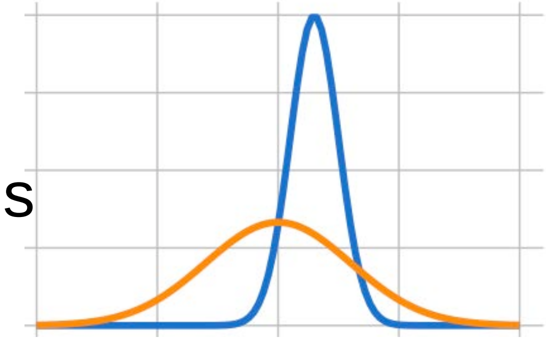
4. Symmetries

A semantic model:

Quasi-Borel spaces

Heunen, Kammar, Staton, Yang, LICS 2017

There's a type constructor `Prob` (a monad), and…

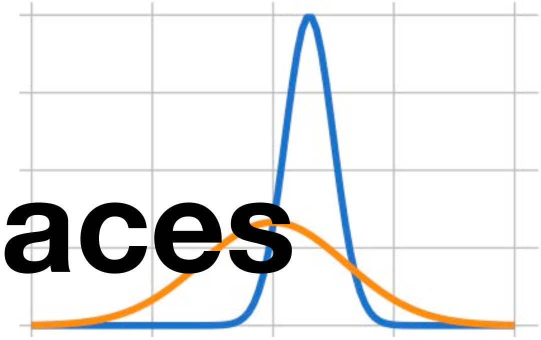


- `Prob RealNum` contains probability distributions (e.g. `normal 0 3`, `uniform 0 1`)
- `RealNum -> Prob RealNum` contains parameterized distributions (e.g. `normal 0`)
- `Prob (RealNum -> RealNum)` contains random functions (e.g. `randlinear`)

- The dataflow property holds.

Other options:

- Domain-theoretic models; Goubault-Larrecq/Jia/Théron; Jia/Lindenhovius/Mislove/Zamdzhiev LICS2021
- Linear-logic based models; e.g. Ehrhard/Pagani/Tasson 2018 Dahlqvist/Kozen POPL 2020
- Topological-domain-based models... e.g. Huang/Morrisett/Spitters



For now: quasi-Borel spaces

Heunen, Kammar, Staton, Yang, LICS 2017

Inspired by:

- Logical relations
- Quasi-topological spaces, diffeological spaces, sequential spaces... see also Matache, Moss, Staton, LICS 2022

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Lemma. One uniform distribution is sufficient to generate all probability measures*.

```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

$$\mathbb{R} \xrightarrow{\alpha} X \xrightarrow{f} Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Types : quasi-Borel spaces.

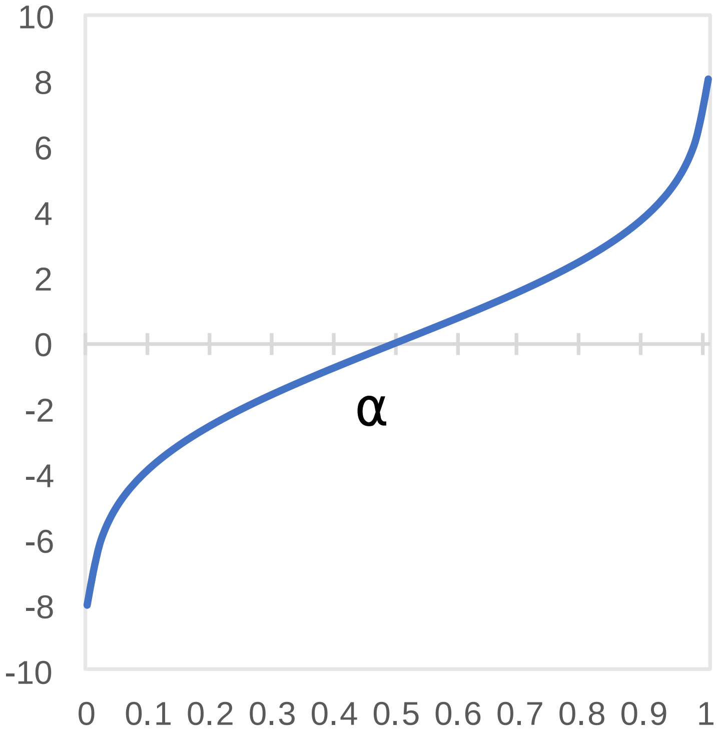
Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

```
do { r <- uniform ; return (α r) }
```

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.



Quasi-Borel spaces

Heunen, Kammar, Staton, Yang, LICS 2017

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

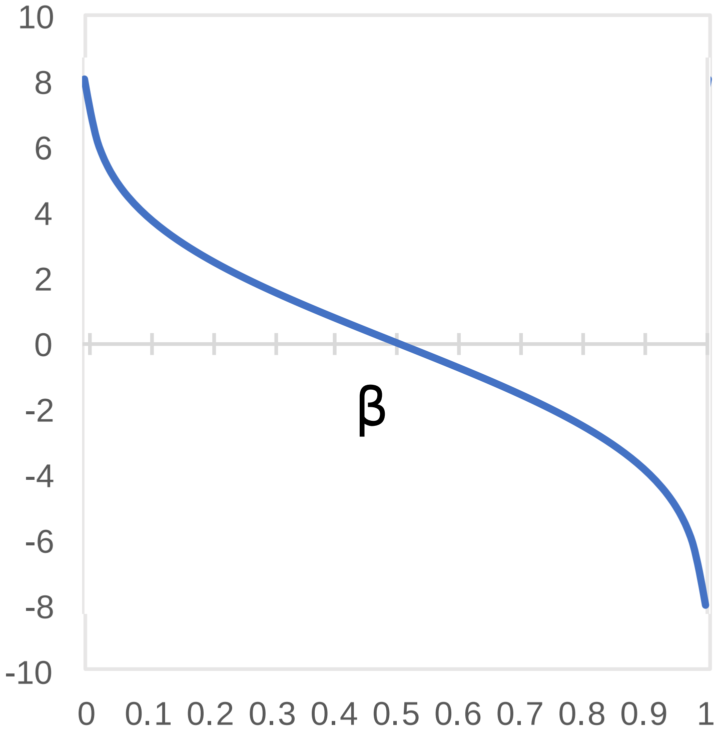
Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.



```
do { r <- uniform ; return (α r) }
```

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Types : quasi-Borel spaces.

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Lemma. One uniform distribution is sufficient to generate all probability measures*.

```
do { r <- uniform ; return (α r) }
```

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

The qBs of reals $(\mathbb{R}, M_\mathbb{R})$ has $M_\mathbb{R} \subseteq [\mathbb{R} \to \mathbb{R}]$ as the Borel functions.

Heunen, Kammar, Staton, Yang, LICS 2017

Quasi-Borel spaces

Defn. A *quasi-Borel space* is a set X equipped with a set of random elements, $M \subseteq [\mathbb{R} \to X]$ such that…

Types : quasi-Borel spaces.

$$[\![\, \texttt{RealNum} \,]\!] = \mathbb{R}$$

$$[\![\, \texttt{Prob a} \,]\!] = Pr([\![\, \texttt{a} \,]\!])$$

Programs : morphisms, i.e. functions $f : X \to Y$ such that

$$f \circ M_X \subseteq M_Y$$

Defn. A *probability measure on a qBs* (X, M_X) is a function in M_X modulo $\sim$.

The qBs of reals $(\mathbb{R}, M_{\mathbb{R}})$ has $M_{\mathbb{R}} \subseteq [\mathbb{R} \to \mathbb{R}]$ as the Borel functions.

Desiderata for a theory of Prob

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- norma
     b <- norma
     let f x =
     return f
```

Theorem. The quasi-Borel space model satisfies the dataflow property.

Heunen, Kammar, Staton, Yang, LICS 2017

```
do c <- normal 0 3
        normal 0 3
        normal 0 3
         x = a*x + b
        n f
```

$$\int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}b\,\mathrm{d}a \qquad \int\int k(\lambda x\,.\,ax+b)\boxed{\mathrm{d}a\,\mathrm{d}b}$$

$$\int\int\int k(\lambda x\,.\,ax+b)\,\mathrm{d}a\,\mathrm{d}b\boxed{\mathrm{d}c}$$

Dataflow property:

*Program lines can be **reordered** and **discarded** if dataflow is preserved.*

Related to Fubini’s theorem.

Desiderata for a theory of Prob

Theorem. The quasi-Borel space model satisfies the dataflow property.

- The probability monad is commutative and affine. cf Kock TAC 2012
- The parameterized distributions form a monoidal category cf Fritz Adv Math 2020, Cho & Jacobs MSCS 2019 Stein & Staton LICS 2021

Dataflow property:

Program lines can be ***reordered*** *and* ***discarded*** *if dataflow is preserved.*

Related to Fubini's theorem.

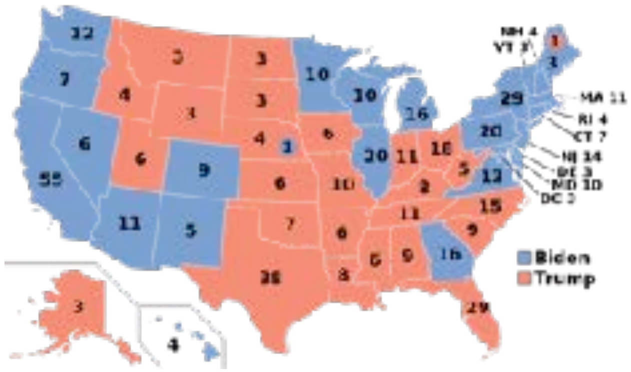
repeat in quasi-Borel spaces

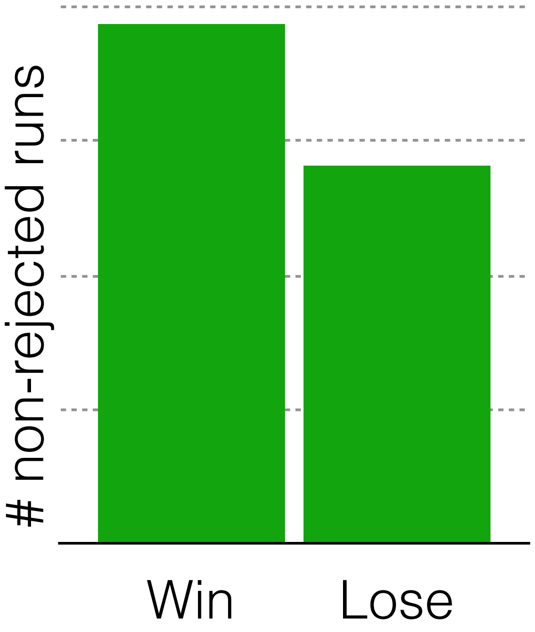
repeat in quasi-Borel spaces

A very simple model deducing chance of win from poll.

```
model :: Prob ([Bool] , Bool)
model = do
   voteShare <- uniform 0 1
   votes <- repeat (bernoulli voteShare)
   return (take 100 votes , (voteShare > 0.5))
```

Simon Walker / HM Treasury & Simon Dawson / No10 Downing Street
[Open Government Licence v3.0](https://www.nationalarchives.gov.uk/doc/open-government-licence/version/3/)





`repeat` in quasi-Borel spaces

```
model :: Prob ([Bool] , Bool)
model = do
  voteShare <- uniform 0 1
  votes <- repeat (bernoulli voteShare)
  return (take 100 votes , (voteShare > 0.5))
```

```
repeat :: Prob a -> Prob [a]
```

Repeatedly draws from a distribution, forever.

Observation.
In measure theoretic probability, `repeat` is defined by *Kolmogorov extension*.

repeat in quasi-Borel spaces

```
model :: Prob ([Bool] , Bool)
model = do
  voteShare <- uniform 0 1
  votes <- repeat (bernoulli vo
  return (take 100 votes , (vot
```

Theorem (summer 2022). `repeat` can be defined for any quasi-Borel space `a`.

```
repeat :: Prob a -> Prob [a]
```

Repeatedly draws from a distribution, forever.

Observation.
In measure theoretic probability, `repeat` is defined by *Kolmogorov extension*.

Programming language foundations for statistics

1. Quick look at
   probabilistic programming for statistics

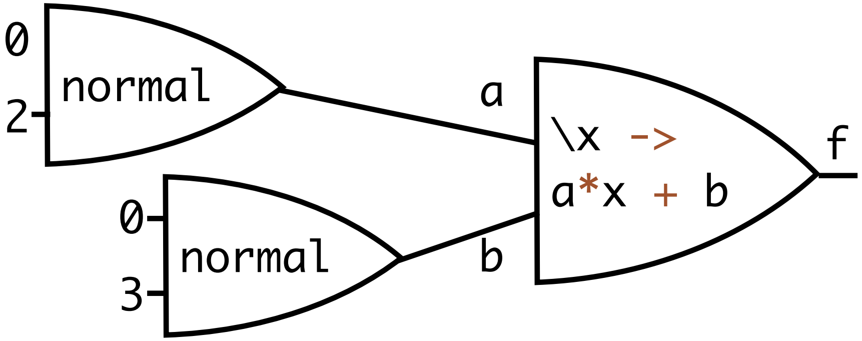
2. Function spaces ...

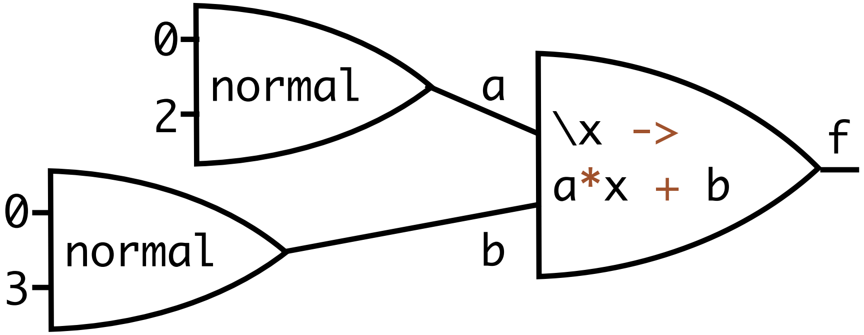
3. ... and understanding them.

4. **Symmetries and names**

Dataflow symmetries

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 2         do b <- normal 0 3
     b <- normal 0 3            a <- normal 0 2
     let f x = a*x + b          let f x = a*x + b
     return f                   return f
```





Dataflow property:

Program lines can be ***reordered*** *and* ***discarded*** *if dataflow is preserved.*

Dataflow symmetries

```
randlinear :: Prob (RealNum -> RealNum)
randlinear =
  do a <- normal 0 2      do b <- normal 0 3
     b <- normal 0 3         a <- normal 0 2
     let f x = a*x + b       let f x = a*x + b
     return f                return f
```

also Jacobs, Staton. CMCS 2020

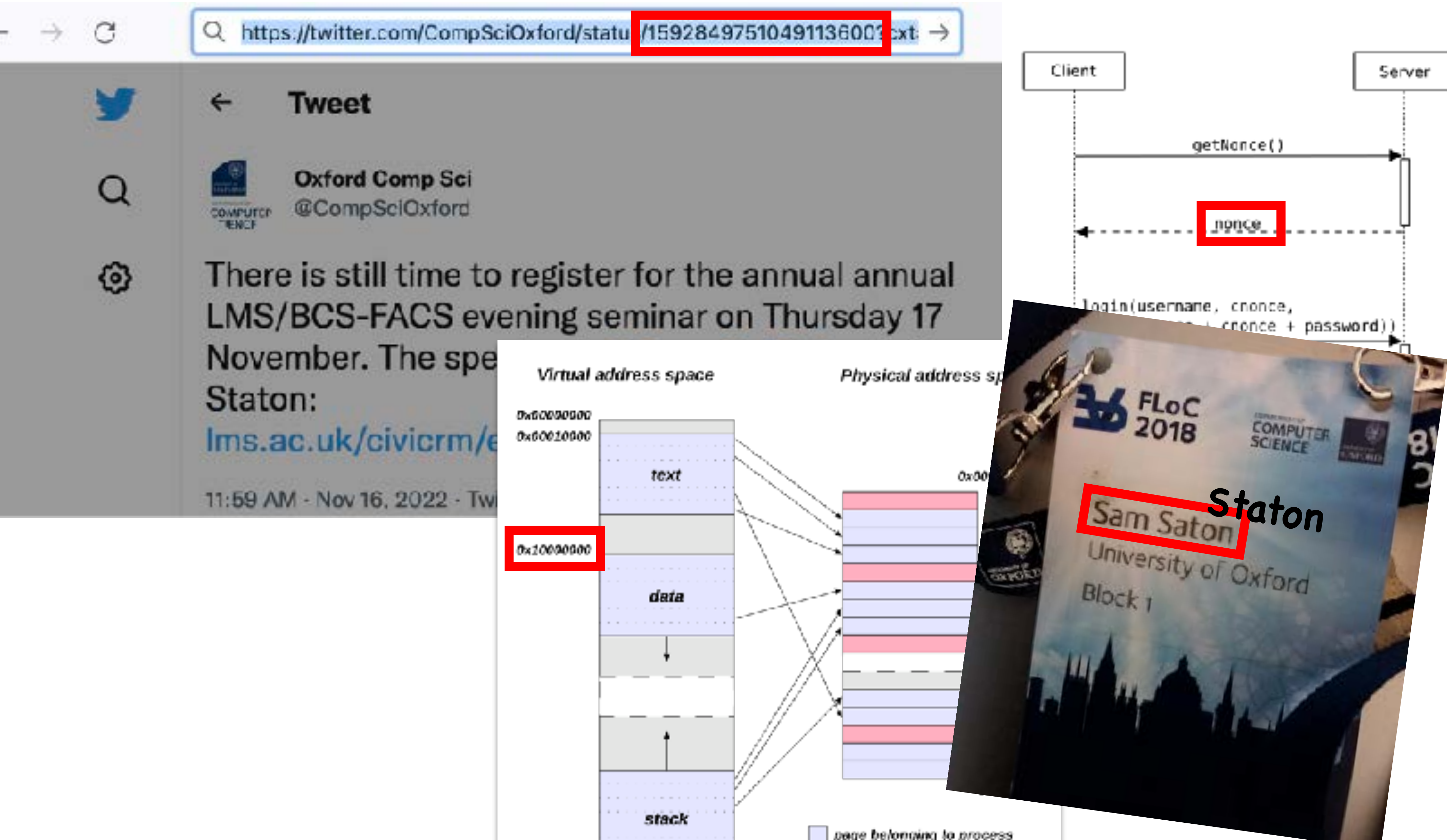
de Finetti (1931):

Independence can be analyzed in terms of reordering ('exchangeability')

Dataflow property:

Program lines can be ***reordered*** *and* ***discarded*** *if dataflow is preserved.*

Names



https://twitter.com/CompSciOxford/statu/1592849751049113600?cxt

Tweet

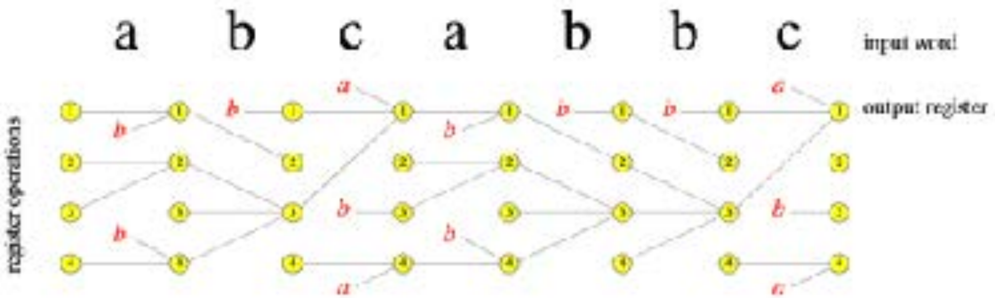
Oxford Comp Sci
@CompSciOxford

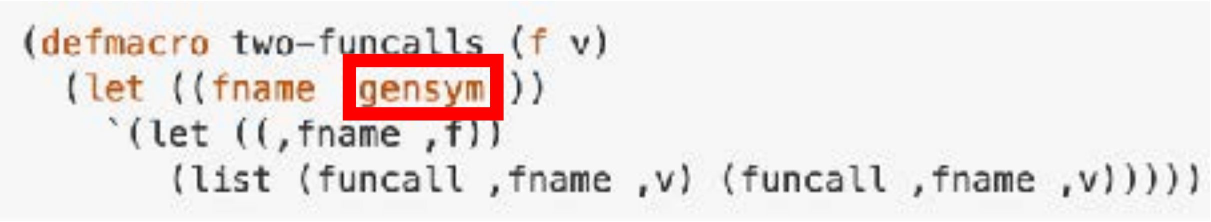
There is still time to register for the annual annual LMS/BCS-FACS evening seminar on Thursday 17 November. The spe
Staton:
lms.ac.uk/civicrm/e

11:59 AM - Nov 16, 2022 · Twi

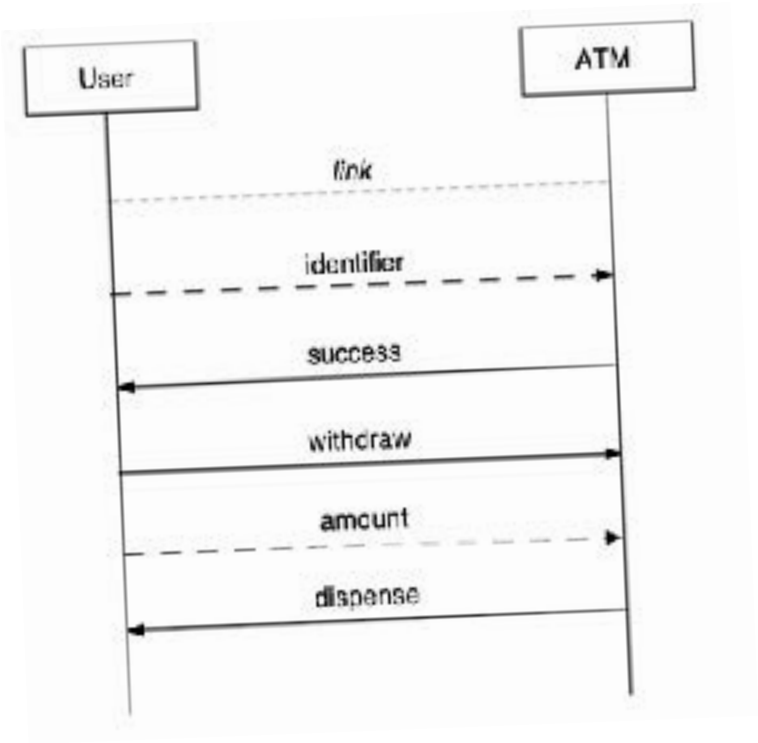
Names

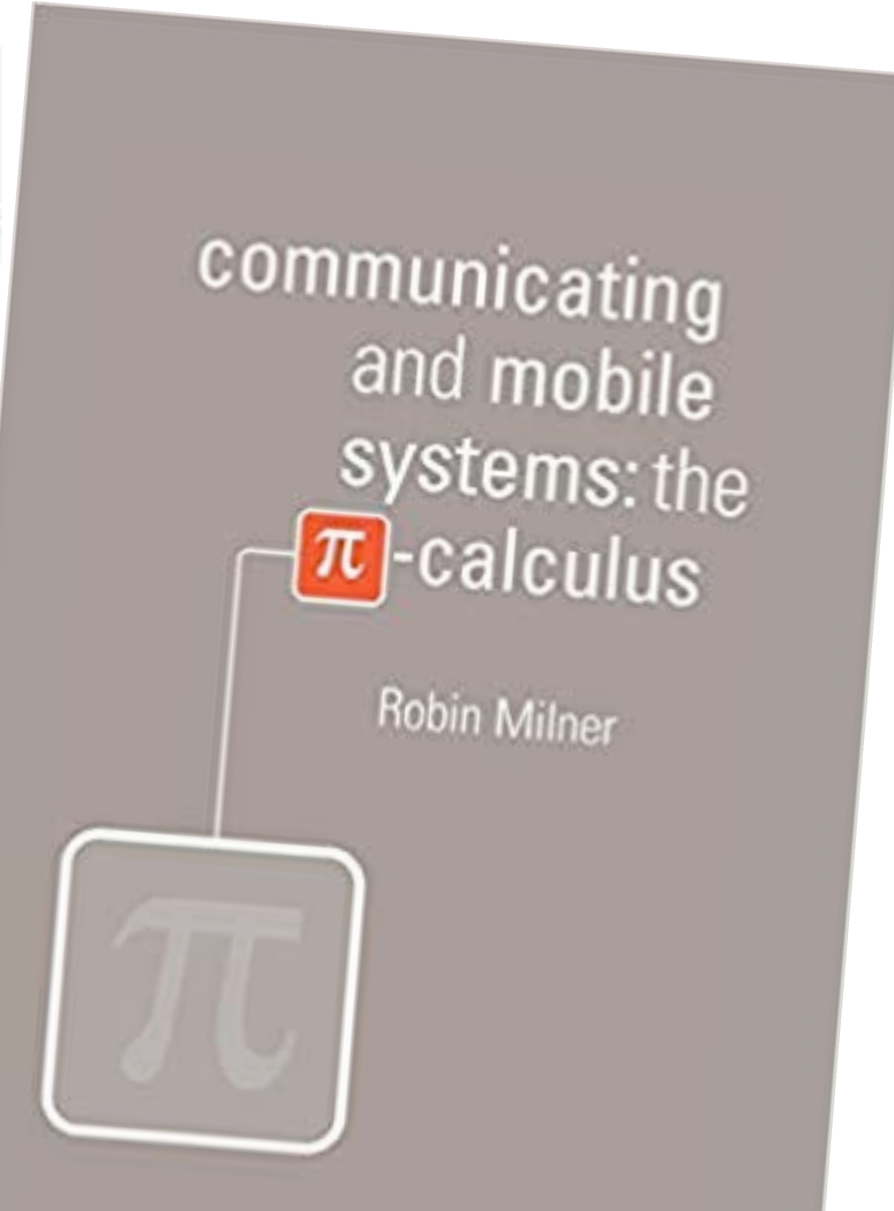
```
let x = fresh-name() in ...
```





```
(defmacro two-funcalls (f v)
  (let ((fname (gensym)))
    `(let ((,fname ,f))
       (list (funcall ,fname ,v) (funcall ,fname ,v)))))
```

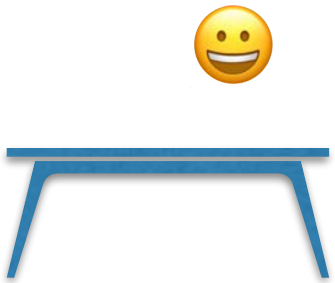




communicating and mobile systems: the π-calculus

Robin Milner

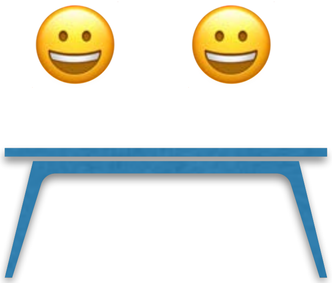
Chinese restaurant process



Each new customer either sits at a random table or a new table.

Chance depends on popularity of tables.

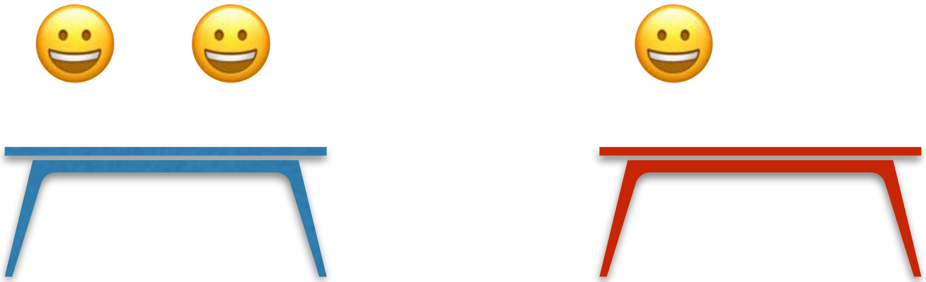
Chinese restaurant process



Each new customer either sits at a random table or a new table.

Chance depends on popularity of tables.

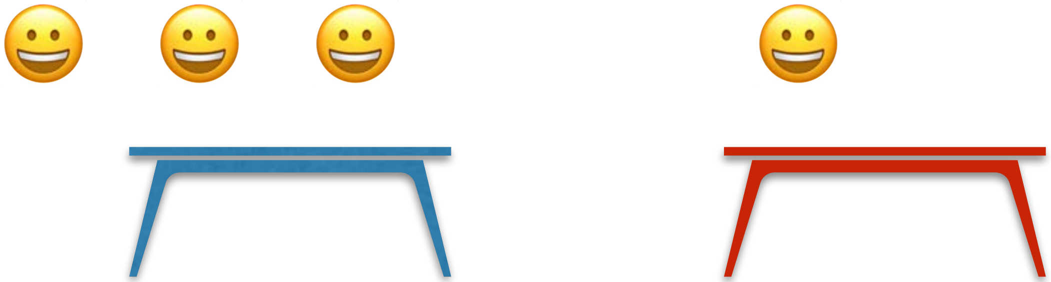
Chinese restaurant process



Each new customer either sits at a random table or a new table.

Chance depends on popularity of tables.

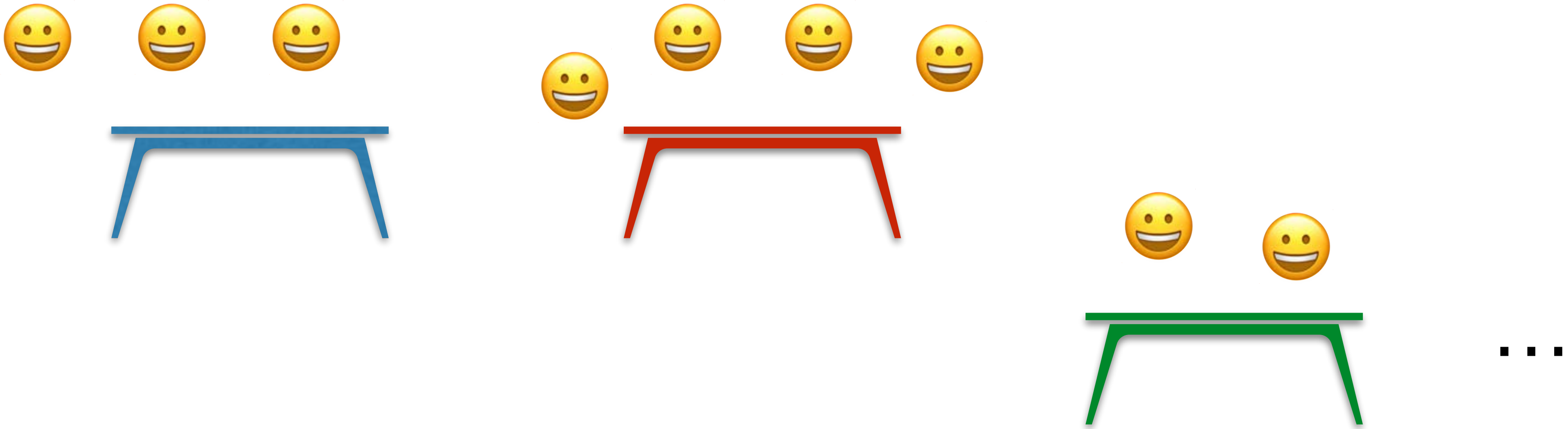
Chinese restaurant process



Each new customer either sits at a random table or a new table.

Chance depends on popularity of tables.

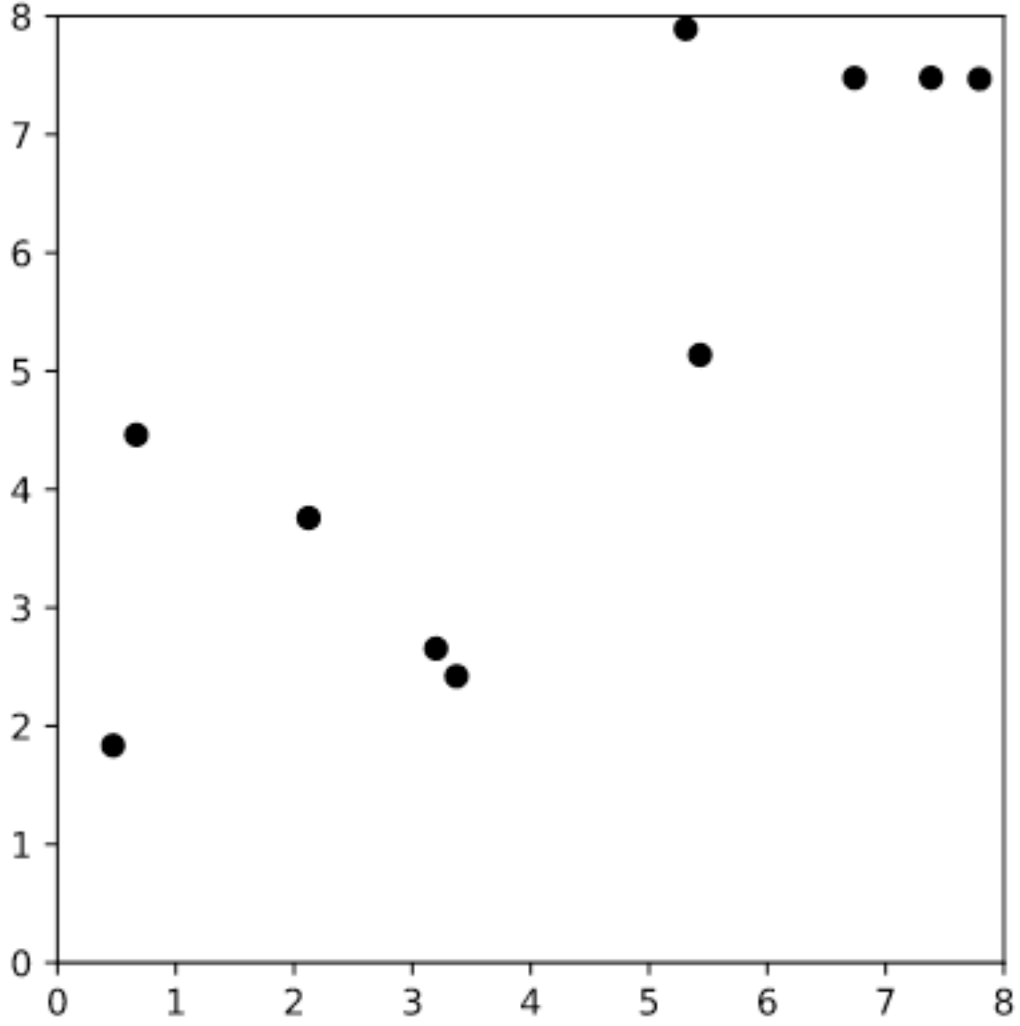
Chinese restaurant process



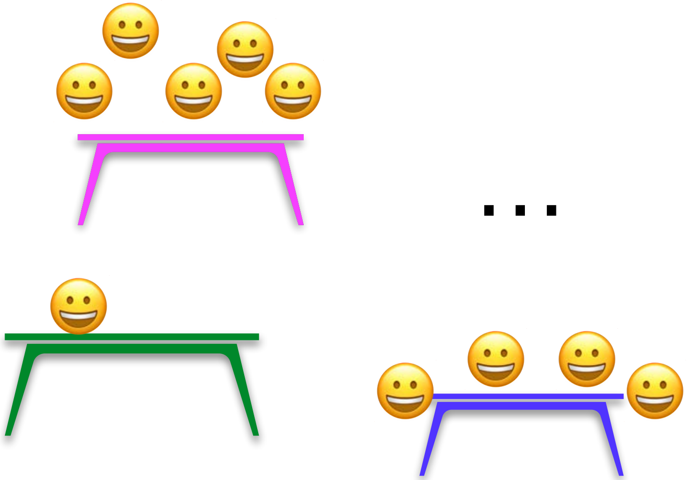
Each new customer either sits at a random table or a new table.

Chance depends on popularity of tables.

Example: Non-parametric clustering

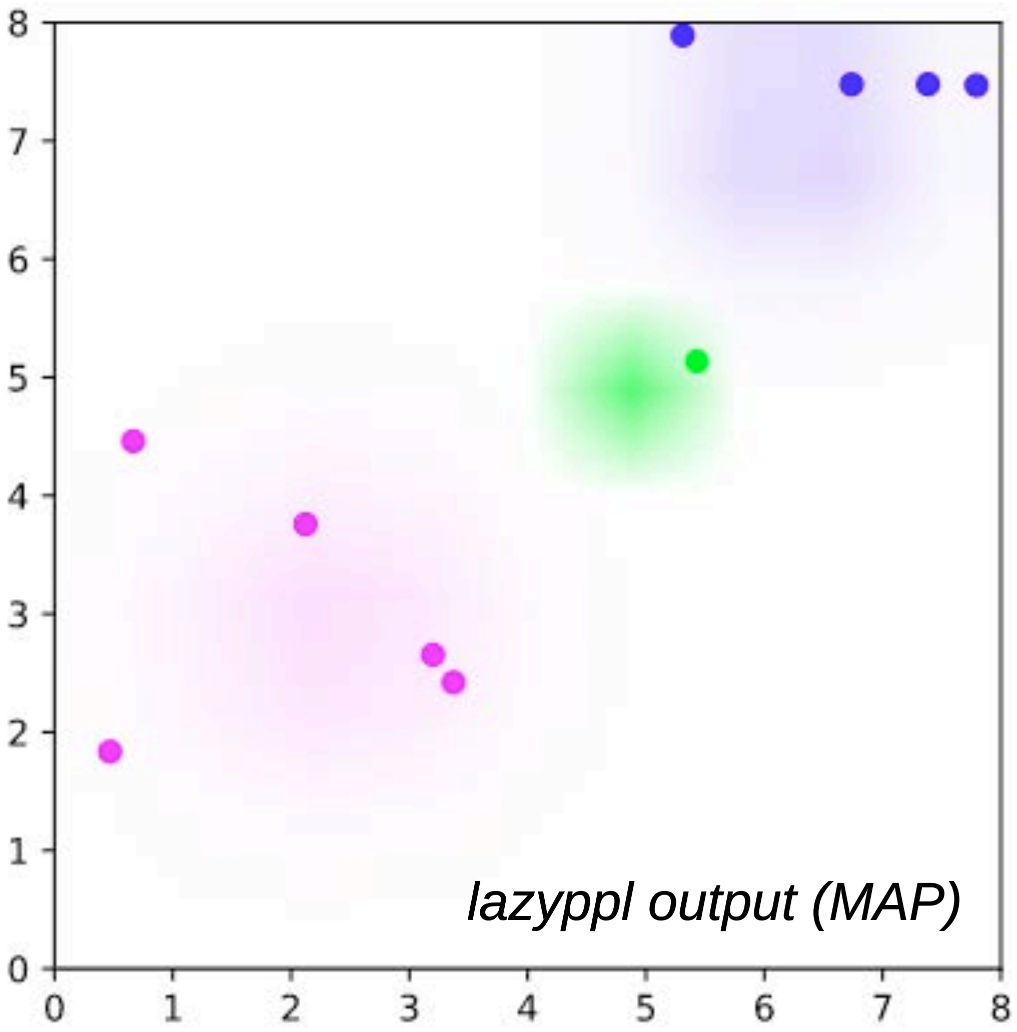


Restaurant metaphor:
Each point is a customer, the clusters are the tables.



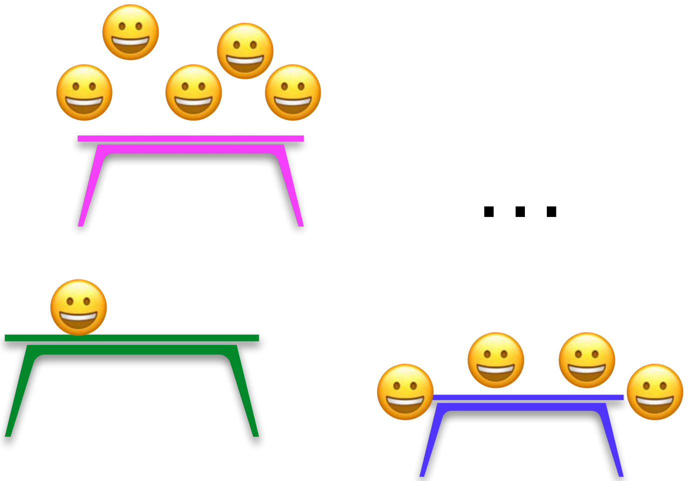
Non-parametric: we don’t know how many clusters.

Example: Non-parametric clustering



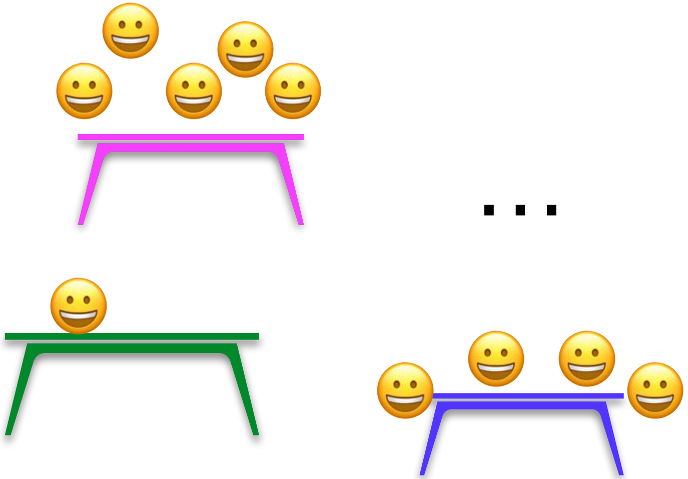
lazyppl output (MAP)

Restaurant metaphor:
Each point is a customer, the clusters are the tables.



Non-parametric: we don’t know how many clusters.

Translation down to traditional prob.



So: apply R to a nominal model to get a measure-theoretic realization.

Theorem:

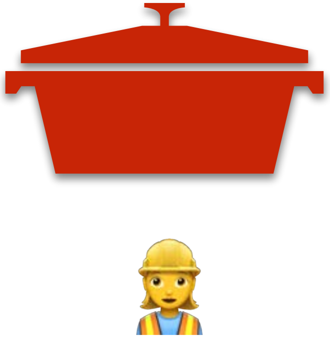
1. TFDAE: (a) a functor $R : \mathbf{NomSet} \to \mathbf{MeasSp}$ that preserves colimits and finite limits.

   (b) a measurable space w/ measurable diagonal.

2. A choice of atomless measure on the space $R(\mathbb{A})$ induces a symmetric monoidal functor extending R, $\mathrm{Kleisli}(\mathit{NameGeneration}) \to \mathrm{Kleisli}(\mathit{Giry})$

cf Sabok, Staton, Stein, Wolman, POPL 2021

Indian buffet process

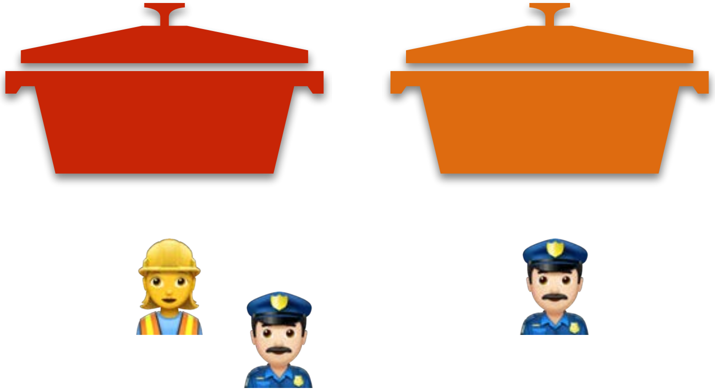


Each new customer takes a set of dishes.

Chance depends on popularity of dishes;
sometimes also take some new dishes.

Griffiths & Ghahramani, JMLR 2011

Indian buffet process

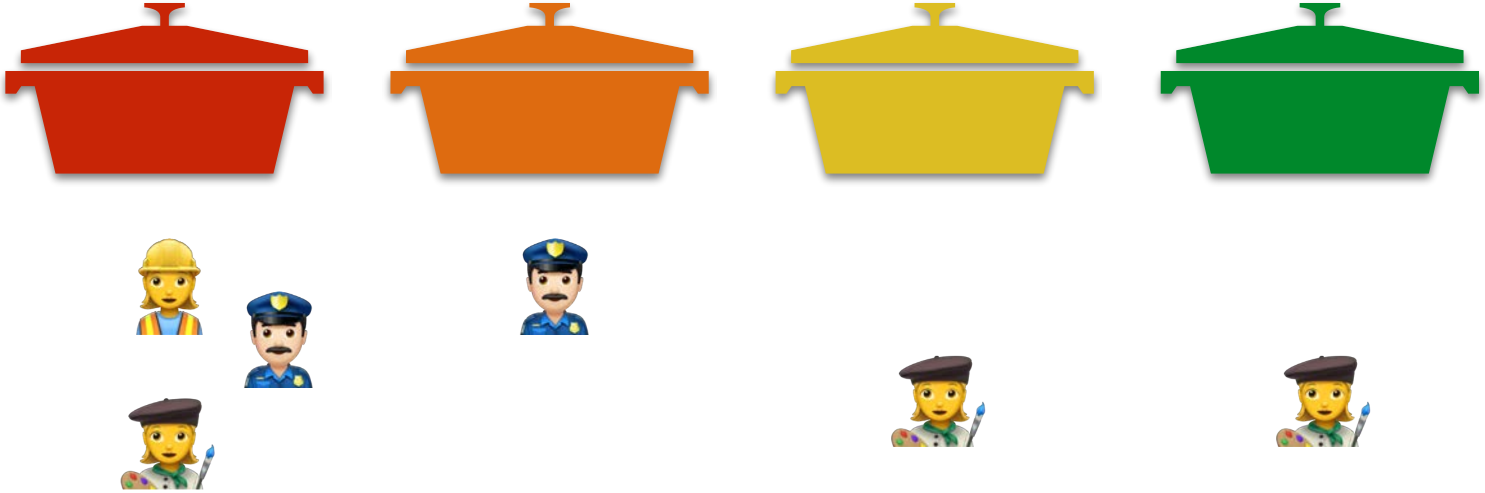


Each new customer takes a set of dishes.

Chance depends on popularity of dishes;
sometimes also take some new dishes.

Griffiths & Ghahramani, JMLR 2011

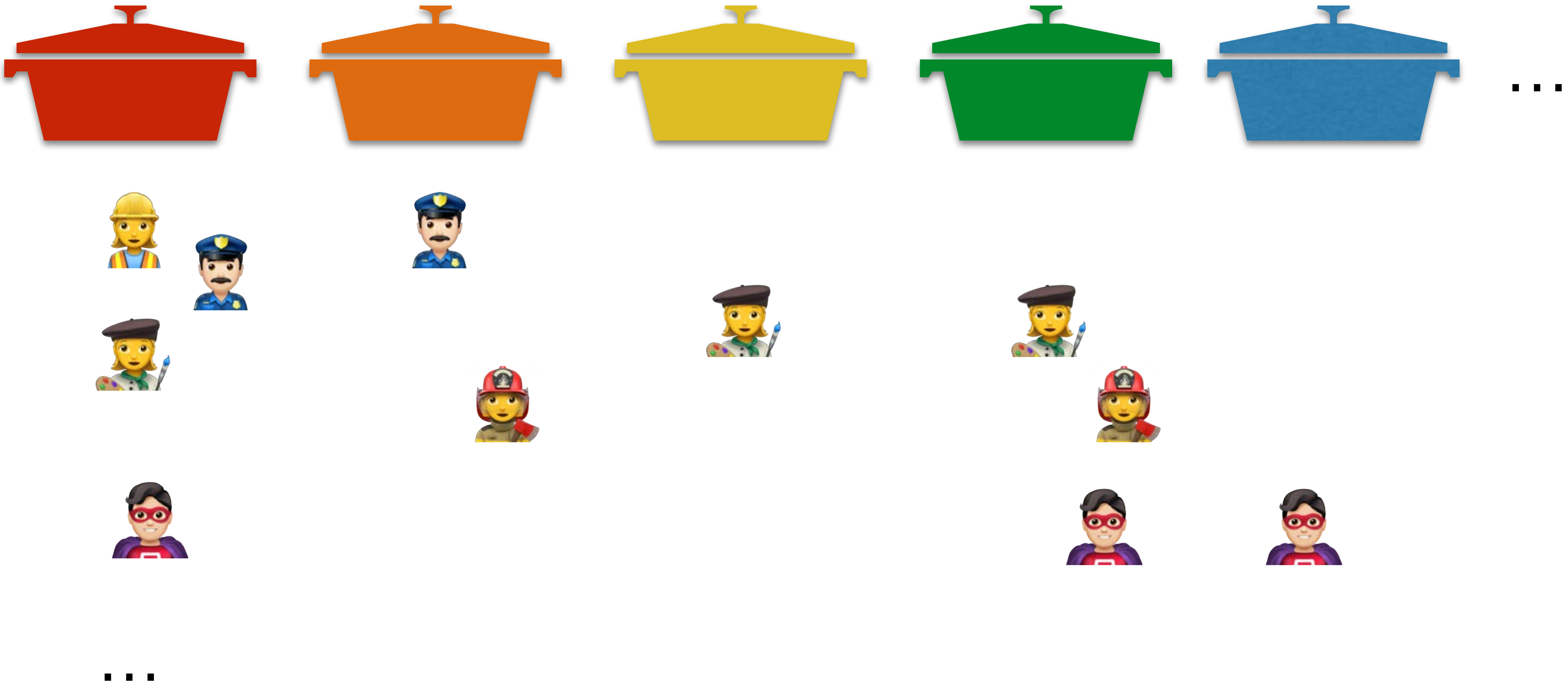
Indian buffet process



Each new customer takes a set of dishes.
Chance depends on popularity of dishes;
sometimes also take some new dishes.

Griffiths & Ghahramani, JMLR 2011

Indian buffet process



Each new customer takes a set of dishes.

Chance depends on popularity of dishes;
sometimes also take some new dishes.

Griffiths & Ghahramani, JMLR 2011

Indian buffets for feature extraction

Example: what are the different features of the countries of the world?

Indian buffets for feature extraction

Example: what are the different features of the countries of the world?

Restaurant metaphor:
Each country is a customer, the features are the dishes that they take.

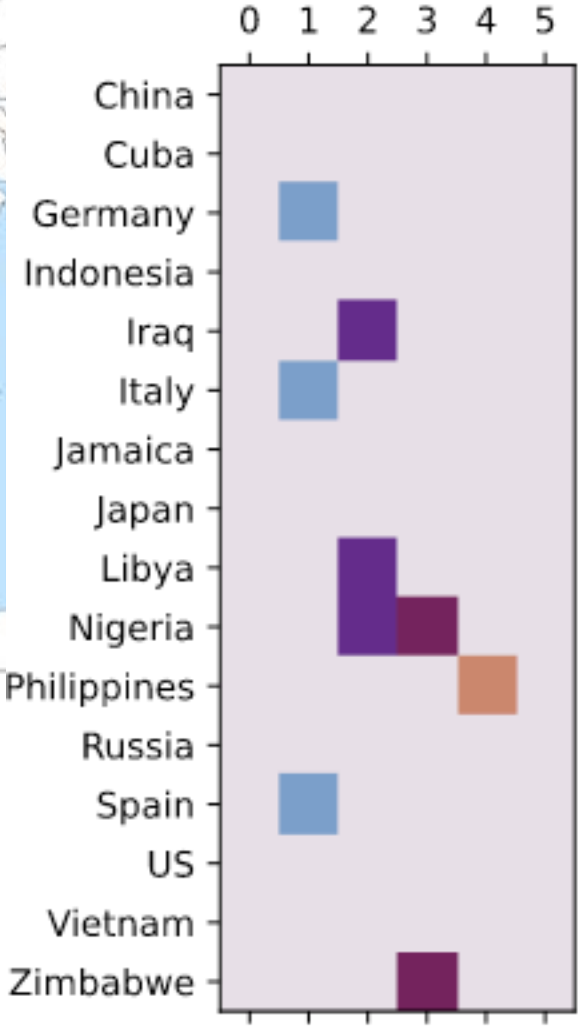
Given experimental data where people say which countries are similar, what are the features?

Navarro & Griffiths, NeurIPS 2006

Indian buffets for feature extraction

Example: what are the different features of the countries of the world?

Restaurant metaphor:
Each country is a customer, the features are the dishes that they take.



lazyppl output (MAP)

Given experimental data where people say which countries are similar, what are the features?

Navarro & Griffiths, NeurIPS 2006

Translation down to traditional prob.

So: apply R to a nominal model to get a measure-theoretic realization.

Theorem:

1. TFDAE: (a) a functor $R : \mathbf{NomSet} \to \mathbf{MeasSp}$ that preserves colimits and finite limits.

   (b) a measurable space w/ measurable diagonal.

2. A choice of atomless measure on the space $R(\mathbb{A})$ induces a symmetric monoidal functor extending R, $\mathrm{Kleisli}(\mathit{NameGeneration}) \to \mathrm{Kleisli}(\mathit{Giry})$

cf Sabok, Staton, Stein, Wolman, POPL 2021

Translation down to traditional prob.

So: apply R to a nominal model to get a
sure-theoretic
ation.

Challenge:
New symmetries,
new programs:
new statistical models

Theorem:

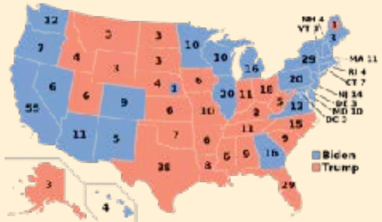
1. TFDAE: (a) a functor $R : \mathbf{NomSet} \to \mathbf{MeasSp}$
that preserves colimits and finite limits.
(b) a measurable space w/ measurable diagonal.

2. A choice of atomless measure on the space $R(\mathbb{A})$ induces a symmetric monoidal functor extending R,
$\mathrm{Kleisli}(\mathit{NameGeneration}) \to \mathrm{Kleisli}(\mathit{Giry})$

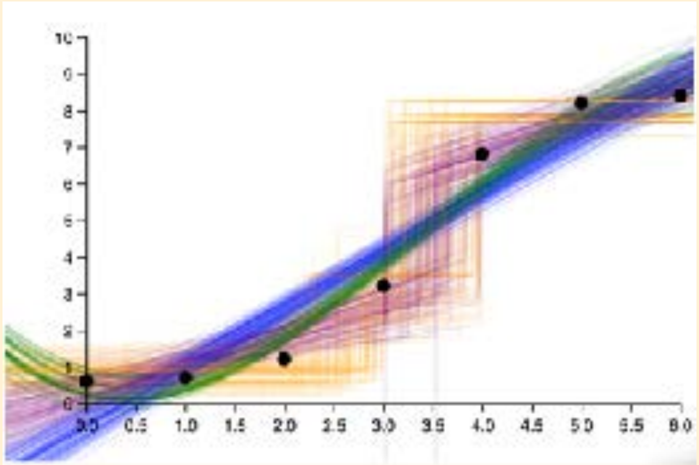
cf Sabok, Staton, Stein, Wolman, POPL 2021

Programming language foundations for statistics

	ML / stats apps	Foundational
High level	✓	✓
Low level	✓	✓



1. Quick look at probabilistic programming for statistics



2. Function spaces ...

3. ... and understanding them.

4. Symmetries and names