CS4340: Probabilistic Programming Seminar

Lecture 2

All course practicalities

https://sebdumancic.github.io/courses/1_prob_prog/

Course practicalities

This is a seminar course

I expect you to come prepared

I expect you to talk more than me

I expect you to do more than just learn the material

There is no textbook, we will use research papers

Course components

Paper reviews (0%)

Participation (10%)

Presentation (25%)

Research report (65%)

Course components: Participation

Ask questions about the papers (before or during the class)

Answer each other’s questions

Course components: Report

Design a research project without executing it

Four components:

- Topic description
- Relation to other topics in the course
- Analysis of the state of the art
- Experimental analysis
- Research design(s). (What, How, Why, Wrong, Experiments)

Feedback time

Course components: Report

I’m not a researcher, how am I supposed to do this?

I’m not expecting you to do a PhD.
Discussion in class helps with this aspect
Future work sections in papers

Course components: Presentation

Goal: present the main ideas as clear as possible

You have to choose what to present

You can use any material on the Web (and share it!)

Last remarks

Choose your papers by September 12 (put your name in the sheet by Sep 8!)

The course is suitable both for 1st and 2nd year of MSc

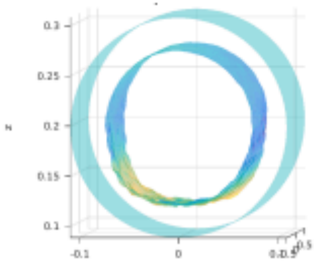
The official PPL for the class if Gen.jl

How to explore the literature?

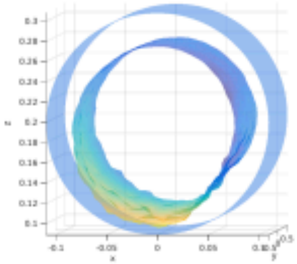
Thinking generatively

Oil Pipeline Inspection

?

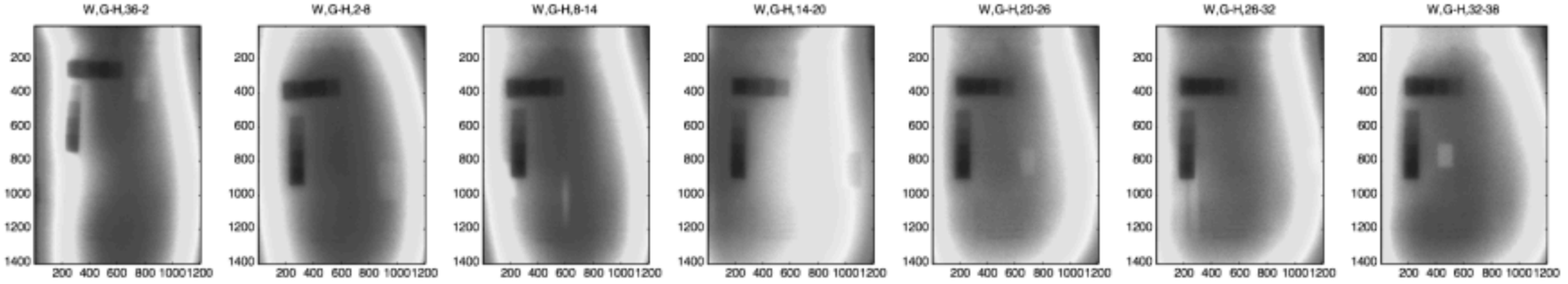


Pipe sample 1

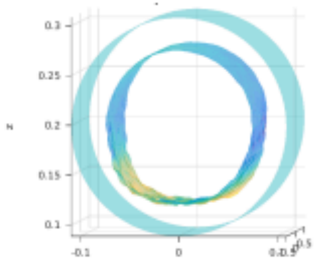


Pipe Sample 2

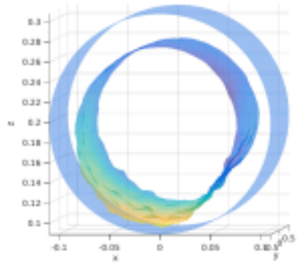
Can you write a program to do this?



Oil Pipeline Inspection

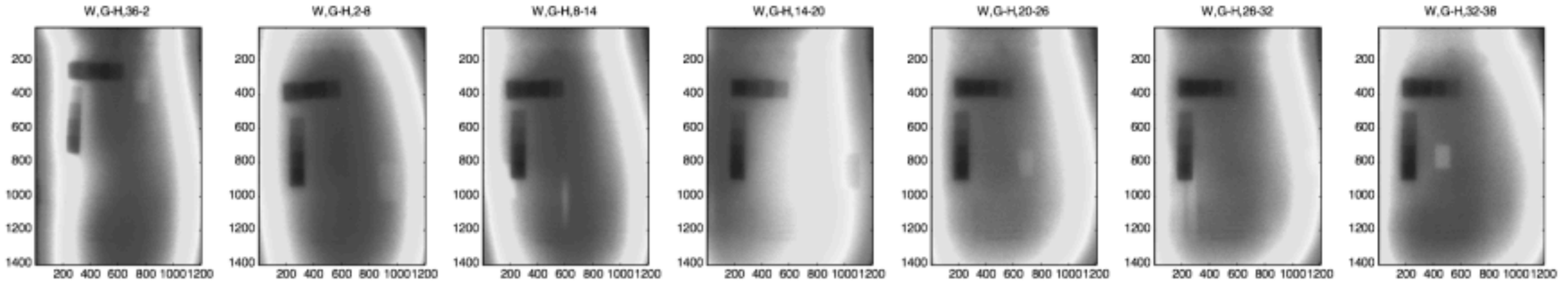


Pipe sample 1

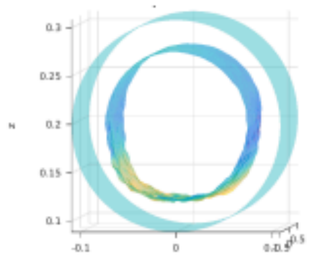


Pipe Sample 2

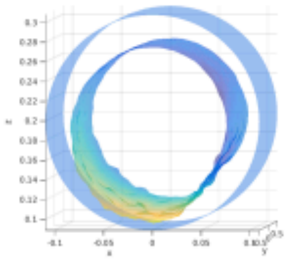
Can you write a program to do this?



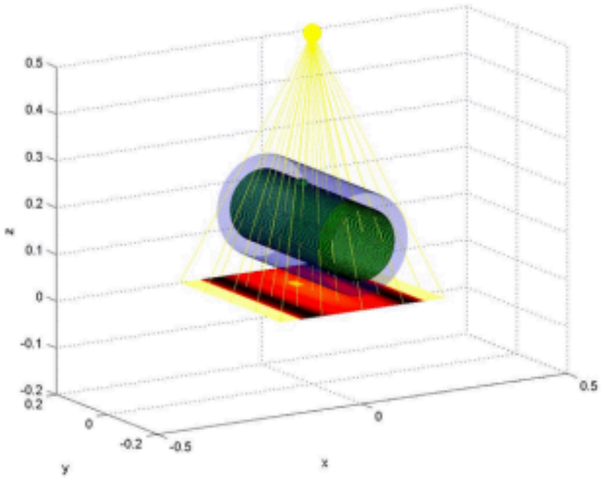
Oil Pipeline Inspection



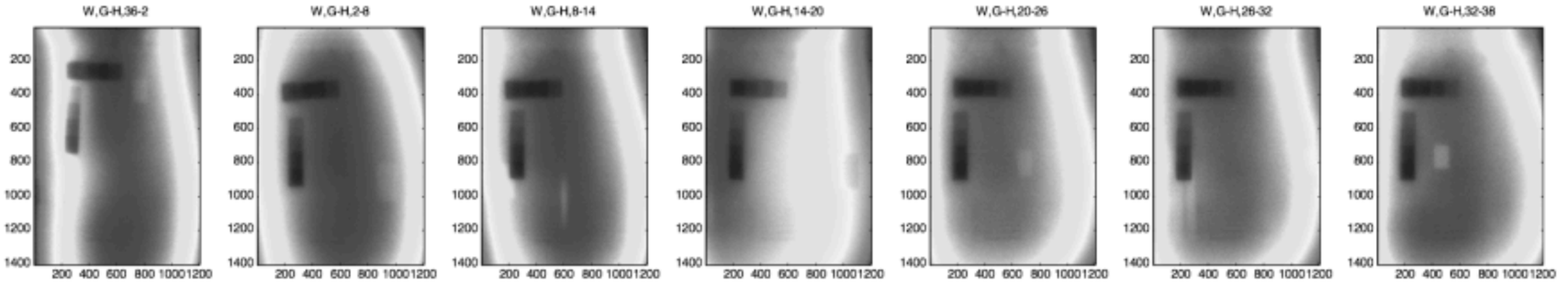
Pipe sample 1



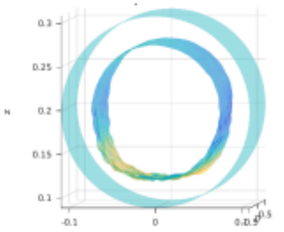
Pipe Sample 2



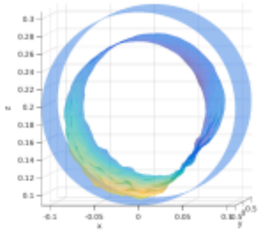
Can you write a program to do this?



Oil Pipeline Inspection



Pipe sample 1



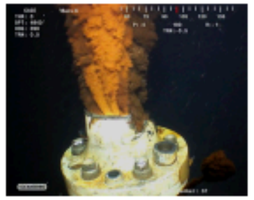
Pipe Sample 2

...

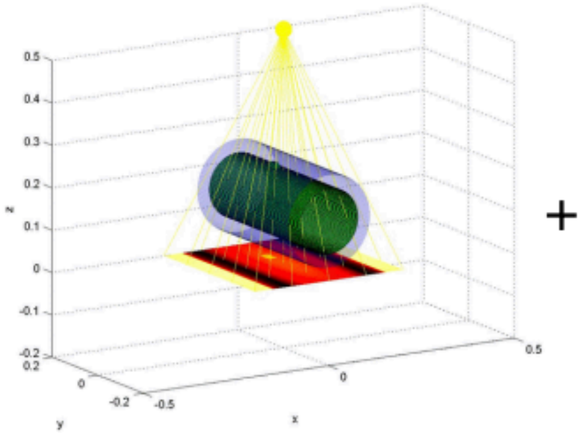
Inference

Generative Model

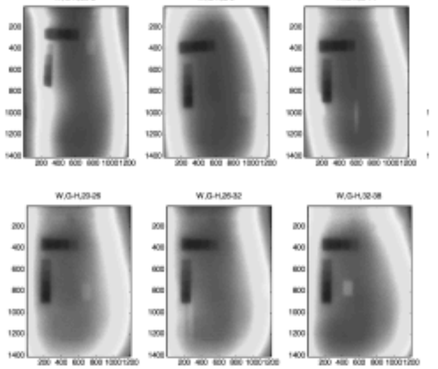
$$p(X|Y) \propto p(X)\, p(Y|X)$$



?



+



[**Rainforth**, Mahendran, Osborne, Vedaldi and Wood. Industry sponsored project 2017]

Captcha

aG8?PY

Can you write a program to do this?

Captcha

aG8?PY

Can you write a program to do this?

Captcha Generative Model

```
(defm sample-char []
 {:symbol (sample (uniform ascii))
  :x-pos (sample (uniform-cont 0.0 1.0))
  :y-pos (sample (uniform-cont 0.0 1.0))
  :size (sample (beta 1 2))
  :style (sample (uniform-dis styles))
  …})
```

```
(defm sample-captcha []
 (let [n-chars (sample (poisson 4))
       chars (repeatedly n-chars
                         sample-char)
    noise (sample salt-pepper)
    …]
  gen-image))
```

Thinking generatively about supervised learning

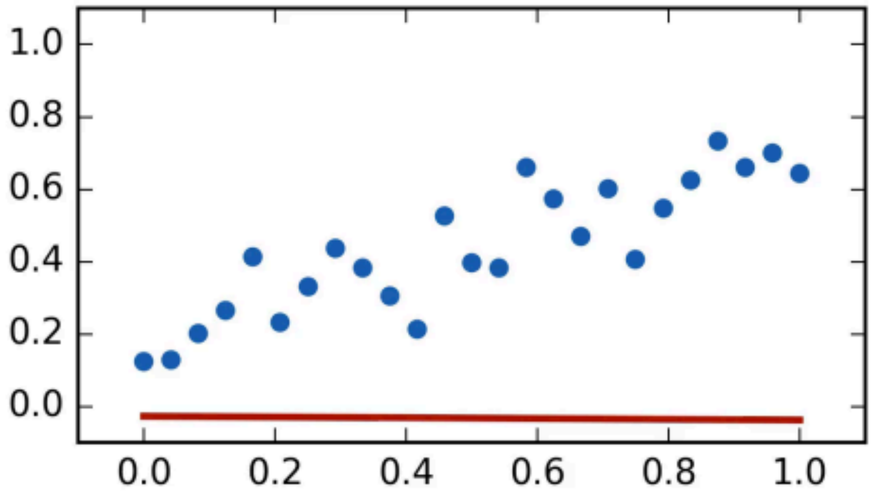
Instead of fitting parameters, specify a prior over them

Generate parameters

“Fit” parameters with inference

Thinking generatively about supervised learning

```
(defquery lin-reg [x-vals y-vals]
 (let [m (sample (normal 0 1))
       c  (sample (normal 0 1))
       f (fn [x] (+ (* m x) c))]
  (map (fn [x y]
         (observe
           (normal (f x) 0.1) y))
       x-vals y-vals))
  [m c])
```




```
(doquery :ipmcmc lin-reg data options)
```

([0.58 -0.05] [0.49 0.1] [0.55 0.05] [0.53 0.04] ….

Symbolic reasoning via generative modelling

- One function denotation: $f(x) = w_0 + \sum_{i=1}^{D} w_i x^i$

- Another function denotation:

```
(defn factorial
  "computes n * (n-1) * ... * 1"
  [n]
  (if (= n 1)
    1
    (*' n (factorial (- n 1)))))
```

- *Generative model for source code*: actually fairly easy to write down

Generative model for arithmetic expressions

Grammar for functions of one variable **x**

- Primitive operations: $\mathbf{op} \in \{+, -, \times, \div\}$
- Terminal symbols: $\mathbf{sym} \in \{x, 0, \ldots, 9\}$
- Simple and compound expressions:

$$\mathbf{e} \rightarrow \mathbf{sym}$$
$$\mathbf{e} \rightarrow (\mathbf{op}\ \mathbf{e}\ \mathbf{e})$$

- The function:

```
(fn [x] (op e e))
```

Samples of arithmetic functions

```clojure
(fn [x] (- (/ (- (* 7 0) 2) x) x))
(fn [x] (- x 8))
(fn [x] (* 5 8))
(fn [x] (+ 7 6))
(fn [x] (* x x))
(fn [x] (* 2 (+ 0 1)))
(fn [x] (/ 6 x))
(fn [x] (- 0 (+ 0 (+ x 5))))
(fn [x] (- x 6))
(fn [x] (* 3 x))
(fn [x]
 (+
  (+
   2
   (- (/ x x) (- x (/ (- (- 4 x) (* 5 4)) (* 6 x)))))
  x))
(fn [x] (- x (+ 7 (+ x 4))))
(fn [x] (+ (- (/ (+ x 3) x) x) x))
(fn [x] (- x (* (/ 8 (/ (+ x 5) x)) (- 0 1))))
(fn [x] (/ (/ x 7) 7))
(fn [x] (/ x 2))
(fn [x] (* 8 x))
(fn [x] (+ 3 (+ x 2)))
```

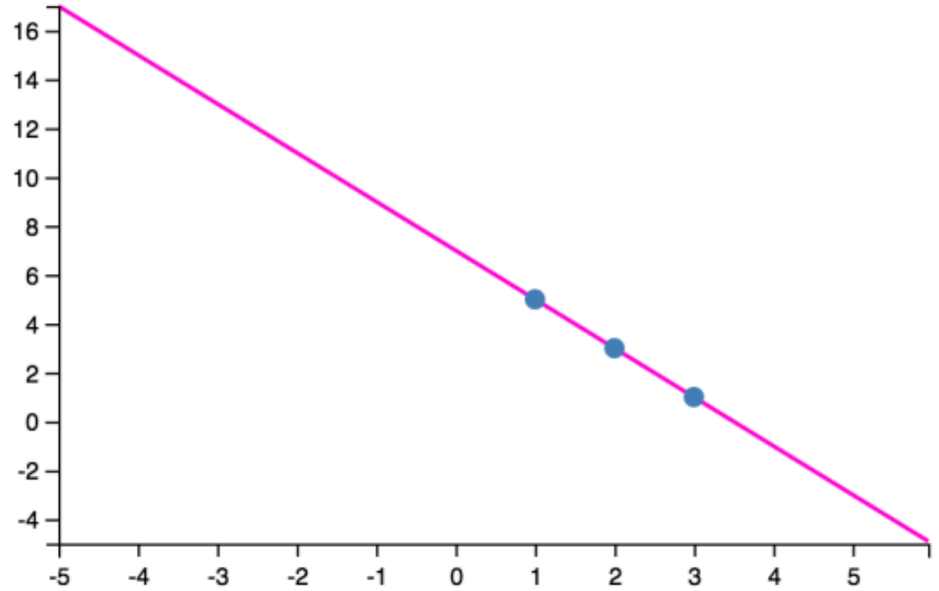
Inference in symbolic model of arithmetic functions

Observations

$f(1) = 5$ $f(2) = 3$ $f(3) = 1$

```
(fn [x] (+ 7 (- (- 0 x) x)))
(fn [x] (- (- 7 x) x))
(fn [x] (+ (- 0 x) (- 7 x)))
(fn [x] (- 7 (+ x x)))
```

Posterior samples



$f(x) = 7 - 2x$

True data generating distribution

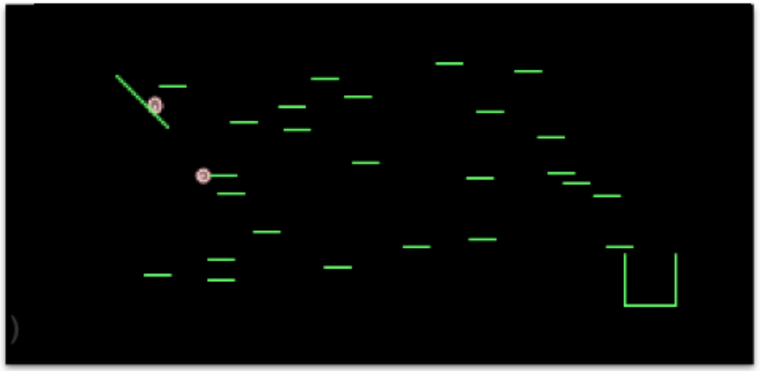
Playing games

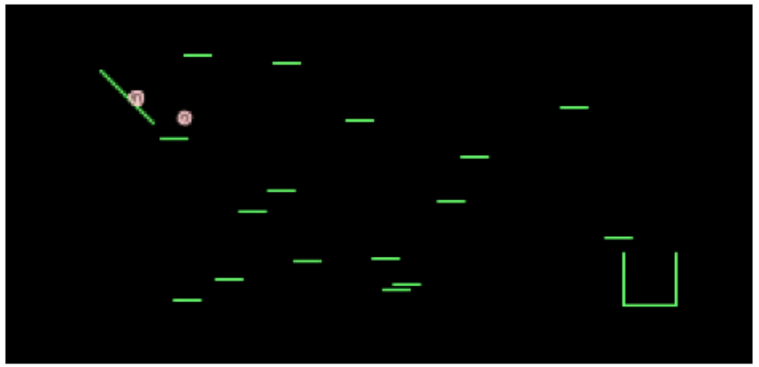
```
(defquery arrange-bumpers []
    (let [bumper-positions []

          ;; code to simulate the world
          world (create-world bumper-positions)
          end-world (simulate-world world)
          balls (:balls end-world)

          ;; how many balls entered the box?
          num-balls-in-box (balls-in-box end-world)]

      {:balls balls
       :num-balls-in-box num-balls-in-box
       :bumper-positions bumper-positions}))
```

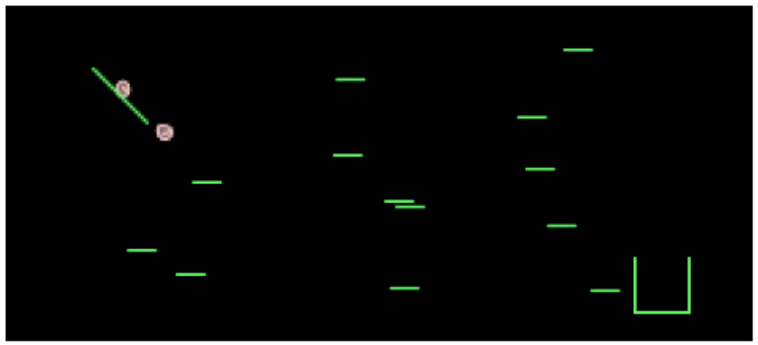
goal: “world” that puts ~20% of balls in box…

Playing games

```
(defquery arrange-bumpers []
    (let [number-of-bumpers (sample (poisson 20))
          bumpydist (uniform-continuous 0 10)
          bumpxdist (uniform-continuous -5 14)
          bumper-positions (repeatedly
                            number-of-bumpers
                            #(vector (sample bumpxdist)
                                     (sample bumpydist)))

          ;; code to simulate the world
          world (create-world bumper-positions)
          end-world (simulate-world world)
          balls (:balls end-world)

          ;; how many balls entered the box?
          num-balls-in-box (balls-in-box end-world)]

      {:balls balls
       :num-balls-in-box num-balls-in-box
       :bumper-positions bumper-positions}))
```







Playing games

```
(defquery arrange-bumpers []
    (let [number-of-bumpers (sample (poisson 20))
          bumpydist (uniform-continuous 0 10)
          bumpxdist (uniform-continuous -5 14)
          bumper-positions (repeatedly
                            number-of-bumpers
                            #(vector (sample bumpxdist)
                                     (sample bumpydist)))

          ;; code to simulate the world
          world (create-world bumper-positions)
          end-world (simulate-world world)
          balls (:balls end-world)

          ;; how many balls entered the box?
          num-balls-in-box (balls-in-box end-world)

          obs-dist (normal 4 0.1)]

      (observe obs-dist num-balls-in-box)

      {:balls balls
       :num-balls-in-box num-balls-in-box
       :bumper-positions bumper-positions}))
```