



PyQB

Monga

PyTensor

Monte-Carlo

Programming in Python¹

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

Academic year 2022/23, I semester

¹ © 2022 M. Monga. Creative Commons Attribution — Condividi allo stesso modo 4.0 Internazionale. <http://creativecommons.org/licenses/by-sa/4.0/deed.it>



PyQB

Monga

PyTensor

Monte-Carlo

Lecture XXIII: Behind pymc3



PyQB

Monga

PyTensor

Monte-Carlo

Behind PyMC

The probabilistic programming approach of PyMC is built on two “technologies”:

- ① A library that mixes numerical and symbolic computations (Theano, Aesara, currently a new implementation called PyTensor)
- ② Markov Chain Monte-Carlo (MCMC) algorithms to estimate posterior densities



PyQB

Monga

PyTensor

Monte-Carlo

PyTensor

It bounds numerical computations to its symbolic structure (“graph”)

```
import pytensor as pt
```

```
a = pt.tensor.dscalar('alpha')
b = pt.tensor.dscalar('beta')
```

```
c = a + b**2
```

```
f = pt.function([a,b], c)
```

```
assert f(1.5, 2) == 5.5
```

Symbolic manipulations



PyQB

Monga

PyTensor

Monte-Carlo

Variables can be used to compute values, but also symbolic manipulations.

```
d = pt.tensor.grad(c, b)
```

```
f_prime = pt.function([a, b], d)
```

```
assert f_prime(1.5, 2) == 4.
```

Note you still need to give an a because the symbolic structure needs it.

159

Markov Chain Monte-Carlo



PyQB

Monga

PyTensor

Monte-Carlo

It's way of estimating (relative) populations of "contiguous" states.

- It needs the capacity of evaluate the population/magnitude of any two close states (but a global knowledge of all the states *at the same time*)
- It's useful to estimate *posterior* distribution *without explicitly computing* $P(D)$: $P(M|D) = \frac{P(D|M) \cdot P(M)}{P(D)}$

160

Metropolis



PyQB

Monga

PyTensor

Monte-Carlo

The easiest MCMC approach is the so-called Metropolis algorithm (in fact appeared as Metropolis, N., **Rosenbluth, A., Rosenbluth, M.**, Teller, A., and Teller, E., 1953)

```
steps = 100000
positions = np.zeros(steps)
populations = [1,2,3,4,5,6,7,8,9,10]
current = 3
```

```
for i in range(steps):
    positions[i] = current
    proposal = (current + np.random.choice([-1,1])) %
    ↪ len(populations)
    prob_move = populations[proposal] /
    ↪ populations[current]
    if np.random.uniform(0, 1) < prob_move:
        current = proposal
```

161

Convergence

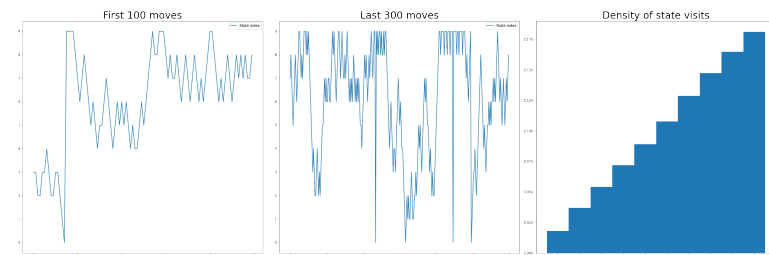


PyQB

Monga

PyTensor

Monte-Carlo



Eventual convergence is guaranteed, but it can be painful slow (and you don't know if you are there...). Many algorithms try to improve: Gibbs, Hamiltonian-MC, NUTS...

162

Putting them together



PyQB

Monga

PyTensor

Monte-Carlo

```
import pymc as pm

linear_regression = pm.Model()

with linear_regression:
    # PyTensor variables
    sigma = pm.Uniform('sigma_h', 0, 50)
    alpha = pm.Normal('alpha', 178, 20)
    beta = pm.Normal('beta', 0, 10)
    mu = alpha + beta*(adult_males['weight'] -
    ↪ adult_males['weight'].mean())
    # Observed!
    h = pm.Normal('height', mu, sigma,
    ↪ observed=adult_males['height'])

trace = pm.sample() # MCMC sampling
```