18.337/6.338 **Fall 2016**

Final Project: RandomizeThenOptimize.jl

Student: Zheng Wang

1 Introduction

1.1 Bayesian inference

Inverse problems often appear in science and engineering. It involves quantifying our knowledge of an unknown input of a math/computational model, based on measurements of its output. Mainly, if we have a model f, and noisy measurements y such that

$$y = f(\theta) + \text{noise}$$

what can we say about the input θ?

Bayesian inference is one possible approach to solve inverse problems. Using Bayesian inference, we model our *belief* on the values the input can take using a distribution. Its recipe prescribes the **posterior** distribution, $p(\theta|y)$, to our knowledge of θ after observing y.

1.2 Final project

For our final project, we implemented Randomize-then-Optimize (RTO) [1], one of many sampling algorithms that can be used to numerically explore the Bayesian posterior. RTO is particularly attractive since its most expensive steps are embarrassingly parallel.

We compared our implementation of RTO, to two other sampling algorithms available from the Julia package `Mamba`: Metropolis-Adjusted Langevin Algorithm (MALA) [3], and Hamiltonian Monte-Carlo (HMC) [2]. We found that RTO is comparable in efficiency to the other two samplers and scales reasonably well in parallel.

2 A simple motivating example

We begin our narrative with a simple motivating example. Consider the following parameterized model with parameters θ:

$$g(x; \theta) = \theta_1 + \theta_2 e^{\theta_3 x}$$

where we are given two noisy measurements given as $\{x, y\}$ pairs.

Figure 1 depicts the data and models generated by randomly drawing θ from a standard multivariate normal. Although the data does not uniquely identify any single parameter, it is clear that not all values of θ match the data. Using Bayesian inference, we can describe the likely values of θ through the posterior.

As seen in Figure 2, the posterior describes an interesting 3D relationship between the parameters. In addition, the models in the posterior more closely match the data.

Remark: In fact, the true posterior contains a second "mode". Similar to how optimization algorithms can get stuck in a local minimum, sampling algorithms can (and often do) get stuck sampling from only one of

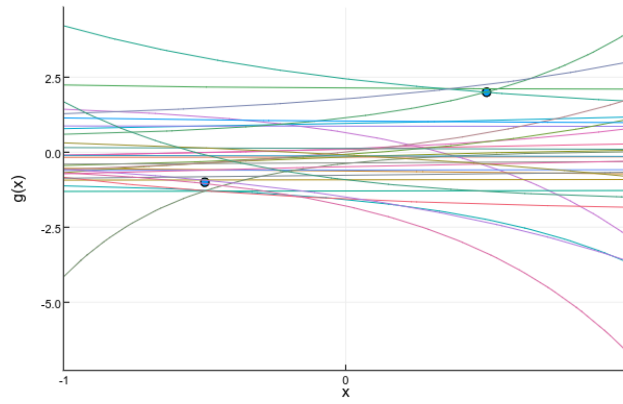
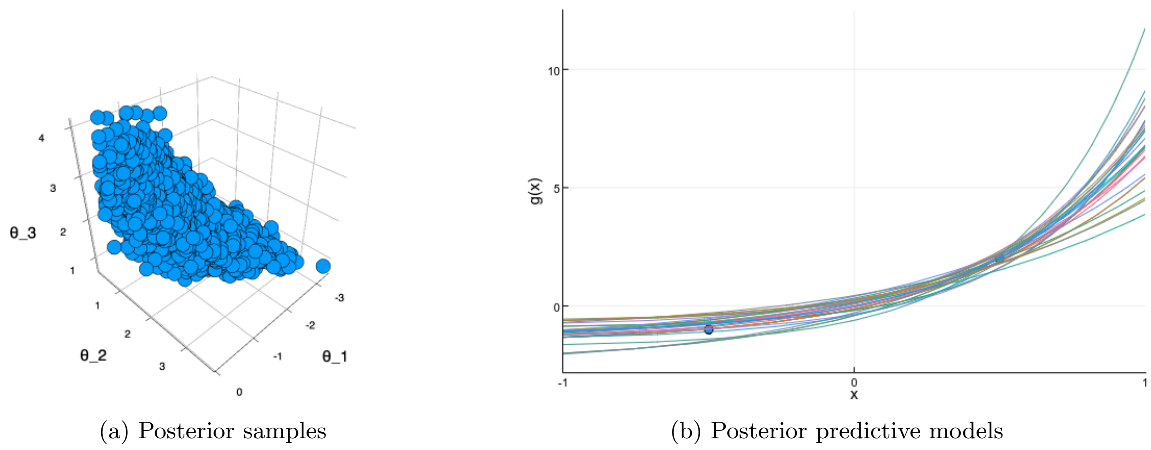


Figure 1: Data and models generated from $\theta \sim N(0, I)$.

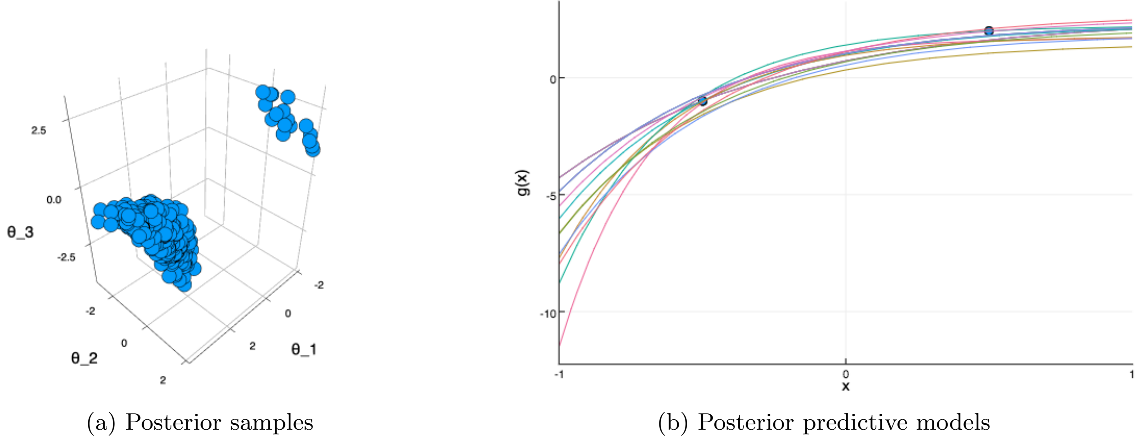


(a) Posterior samples

(b) Posterior predictive models

Figure 2: Posterior samples of θ in the first mode.

many modes of a multi-modal posterior. For this motivating example, we can sample from the second mode by choosing a different starting point. Samples from the second posterior mode are depicted in Figure 3.



(a) Posterior samples

(b) Posterior predictive models

Figure 3: Posterior samples of θ in the second mode.

3 Julia package for Randomize-then-Optimize

For our final project, we implemented RTO — an optimization-based sampling algorithm for Bayesian inference. The Julia code is publicly available at `https://github.com/wang-zheng/RandomizeThenOptimize.jl` along with a brief tutorial on how to use it.

The most computationally expensive step of RTO is to repeatedly solve an optimization problem, each with random forcing in the objective function. For these optimizations, we use the gradient based algorithms in `NLopt`. Since the objective functions do not depend on any previous solution, the optimization problems are embarrassingly parallel. Our implementation takes advantage of this fact and will use any available additional worker processors.

4 Numerical results

We conducted three numerical experiments, each with a different **forward model** — the function from parameters θ to data y. The forward models are:

1. Parameter estimation, $f : \mathbb{R}^3 \to \mathbb{R}^2$
$$f(\theta) = \begin{bmatrix} \theta_1 + \theta_2 \, e^{\theta_3 x_1} \\ \theta_1 + \theta_2 \, e^{\theta_3 x_2} \end{bmatrix}$$
This experiment corresponds to the motivating parameter estimation problem in Section 2.

2. Sinusoidal, $f : \mathbb{R}^2 \to \mathbb{R}$
$$f(\theta) = a \sin(b \, \theta_1) - c \, \theta_2$$
This experiment produces a 2D sinusoidal-looking posterior distribution.

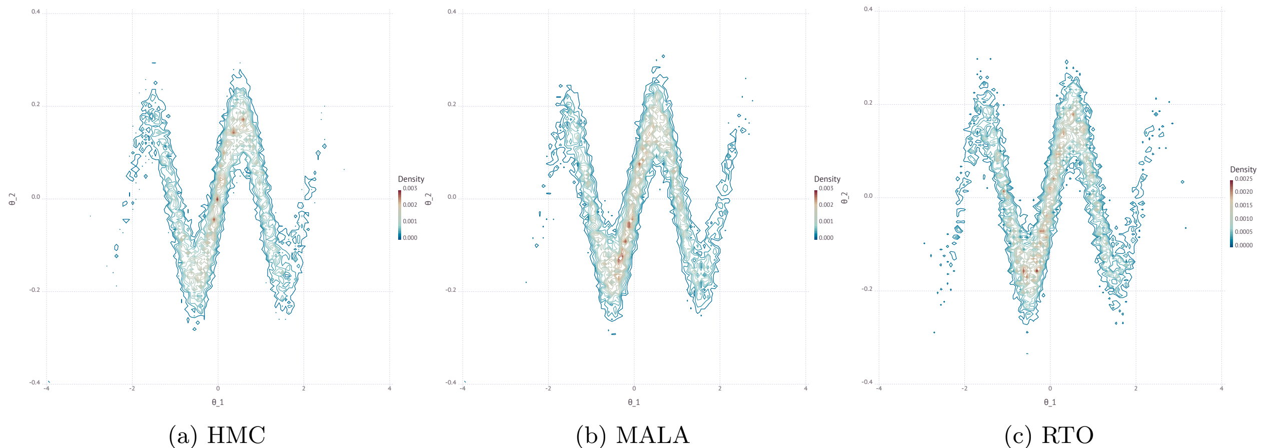
3. Matrix multiply, $f : \mathbb{R}^{30} \to \mathbb{R}^{20}$

$$f(\theta) = A\,\theta$$

This experiment tests the sampling algorithms a large (30-dimensional) parameter space.

We used three algorithms: MALA and HMC — from `Mamba`; and RTO — from our package; to sample the posterior distributions of all three experiments.

All three algorithms should, in theory, draw samples from the same distribution. The samples visually appear consistent from all samplers for all three experiments. Figure 4 depicts the sample posteriors for the sinusoidal experiment. This result gives us confidence that RTO is correctly implemented.



(a) HMC (b) MALA (c) RTO

Figure 4: Posterior samples for the sinusoidal example using all three samplers.

We then compare the sampling efficiency of the three samplers. The quality of a chain of correlated samples generated from these algorithms, is often measured using Effective Sample Size (ESS). ESS can be thought of as the number of independent samples that our chain of correlated samples contains[1]. We use ESS per computation time as a measure of efficiency. ESS is calculated by `Mamba` and computation time is found using `BenchmarkTools`. RTO slightly outperforms the other two algorithms for all three examples, as seen in Table 1. However, since the other algorithms have many tunable settings that affect their sampling efficiency, we can only conclude that these three samplers are comparable[2].

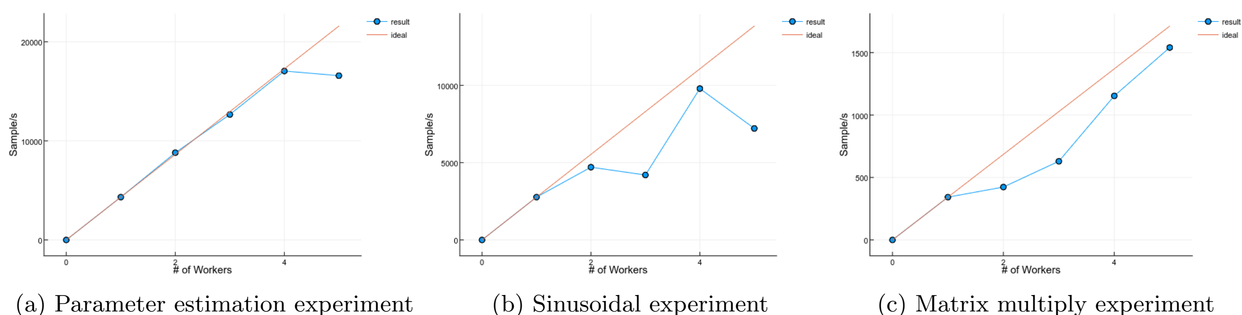
Table 1: ESS per second

	MALA	HMC	RTO
Experiment 1	900	525	1531
Experiment 2	452	199	1670
Experiment 3	230	212	241

Finally, we compare the parallel performance of RTO on the three examples. As seen in Figure 5, the algorithm does not achieve ideal parallel scaling but does speed up as we add more workers.

[1] Technically, it is defined as the number of samples required in an independent Monte-Carlo estimate to have the same variance as the Monte-Carlo estimate from our correlated chain. ESS is estimated component-wise using the autocorrelation.

[2] The efficiency of the three samplers are around the same order of magnitude.



(a) Parameter estimation experiment (b) Sinusoidal experiment (c) Matrix multiply experiment

Figure 5: Posterior samples for the sinusoidal example using all three samplers.

A Some further thoughts

- For simple forward models, the array initialization in the algorithm's inner most loop took most of the computation time. I toyed with three different matrix initializations:

  1. `A = view(B,1:n,1:m)`
  2. `A = Array(Float64,n,m)`
  3. `A = Array(eltype(B),n,m)`

  Here the matrix `B`, is later written over. The list is ordered from fastest to slowest. However, the first method requires that `AbstractVector` and `AbstractMatrix` are valid inputs for the user-specified function. In addition, the second method assumes that all the inputs have elements of type `Float64`. I ended up decided to go with the second method for a balance of speed and user-friendliness. Is there a better way? What would you suggest?

- In developing the package I found it useful to define a function that performs a commutative operation element-wise on two tuples. The specific operation I wanted was `hcat` and so I wrote a hidden function in the package to do so. For the future, is there a simpler way to do this?

- On GitHub, have found many Julia packages with a `LICENSE` file. Do I need one? If so, how can I get one?

References

[1] Johnathan M Bardsley, Antti Solonen, Heikki Haario, and Marko Laine. Randomize-then-optimize: A method for sampling from posterior distributions in nonlinear inverse problems. *SIAM Journal on Scientific Computing*, 36(4):A1895–A1910, 2014.

[2] Radford M Neal. MCMC using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2:113–162, 2011.

[3] Gareth O. Roberts and Richard L. Tweedie. Exponential convergence of Langevin distributions and their discrete approximations. *Bernoulli*, pages 341–363, 1996.