Darren Wilkinson's research blog

Statistics, computing, data science, Bayes, stochastic modelling, systems biology and bioinformatics

MCMC programming in R, Python, Java and C

EDIT 16/7/11 – this post has now been largely superceded by a new post!

MCMC

Markov chain Monte Carlo (MCMC) is a powerful simulation technique for exploring high-dimensional probability distributions. It is particularly useful for exploring posterior probability distributions that arise in Bayesian statistics. Although there are some generic tools (such as WinBugs and JAGS) for doing MCMC, for non-standard problems it is often desirable to code up MCMC algorithms from scratch. It is then natural to wonder what programming languages might be good for this purpose. There are hundreds of programming languages one could in principle use for MCMC programming, but it is necessary to use a language with a good scientific library, including good random number generation routines. I have used a large number of programming languages over the years, but these days I mostly program in R, Python, Java or C. I find each of these languages interesting and useful, with different strengths and weaknesses, and I find that no one of these languages dominates any of the others in all situations.

Example

For the purposes of this post we will focus on Gibbs sampling for a simple bivariate distribution defined on the half-plane $x>0$.

$f(x,y) = k\, x^2 \exp\{-xy^2-y^2+2y-4x\}$

The statistical motivation is not important for this post, but this is the kind of distribution which arises naturally in Bayesian inference for the mean and variance of a normal random sample. Gibbs sampling is a simple MCMC scheme which samples in turn from the full-conditional distributions. In this case, the full-conditional for x is Gamma(3,y^2+4) and the full-conditional for y is N(1/(x+1),1/(x+1)). The resulting Gibbs sampling algorithm is very simple and works very efficiently, but for illustrative purposes, we will consider generating 20,000 iterations with a "thin" of 500 iterations.

Implementations

R

Although R has many flaws, it is well suited to programming with data, and has a huge array of statistical libraries associated with it. Like many statisticians, I probably use R more than any other language in my day-to-day work. It is the language I always use for the analysis of MCMC output. It is therefore natural to think about using R for coding up MCMC algorithms. Below is a simple function to implement a Gibbs sampler for this problem.

```
gibbs=function(N,thin)
{
    mat=matrix(0,ncol=3,nrow=N)
    mat[,1]=1:N
    x=0
    y=0
    for (i in 1:N) {
        for (j in 1:thin) {
            x=rgamma(1,3,y*y+4)
            y=rnorm(1,1/(x+1),1/sqrt(x+1))
        }
        mat[i,2:3]=c(x,y)
    }
    mat=data.frame(mat)
    names(mat)=c("Iter","x","y")
    mat
}
```

This function stores the output in a data frame. It can be called and the data written out to a file as follows:

```
writegibbs=function(N=20000,thin=500)
{
    mat=gibbs(N,thin)
    write.table(mat,"data.tab",row.names=FALSE)
}
tim=system.time(writegibbs())
print(tim)
```

This also times how long the function takes to run. We will return to timings later. As already mentioned, I always analyse MCMC output in R. Below is some R code to compare the contours of the true distribution with the contours of the empirical distribution obtained by a 2d kernel smoother.

```
fun=function(x,y)
{
    x*x*exp(-x*y*y-y*y+2*y-4*x)
}

mat=read.table("data.tab",header=TRUE)
op=par(mfrow=c(2,1))
x=seq(0,4,0.1)
y=seq(-2,4,0.1)
z=outer(x,y,fun)
contour(x,y,z,main="Contours of actual distribution")
require(KernSmooth)
fit=bkde2D(as.matrix(mat[,2:3]),c(0.1,0.1))
contour(fit$x1,fit$x2,fit$fhat,main="Contours of empirical distribution")
par(op)
```

Python

Python is a great general purpose programming language. In an ideal world, I would probably use Python for all of my programming needs. These days it is also a reasonable platform for scientific computing, due to the scipy project. It is certainly possible to develop MCMC algorithms in Python. A simple Python script for this problem can be written as follows.

```
import random,math

def gibbs(N=20000,thin=500):
    x=0
    y=0
    print "Iter  x  y"
    for i in range(N):
        for j in range(thin):
            x=random.gammavariate(3,1.0/(y*y+4))
            y=random.gauss(1.0/(x+1),1.0/math.sqrt(x+1))
        print i,x,y

gibbs()
```

This can be run with a command like `python gibbs.py > data.tab`. Note that python uses a different parametrisation of the gamma distribution to R.

C

The language I have used most for the development of MCMC algorithms is C (usually strict ANSI C). C is fast and efficient, and gives the programmer lots of control over how computations are carried out. The GSL is an excellent scientific library for C, which includes good random number generation facilities. A program for this problem is given below.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
```

```c
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

void main()
{
  int N=20000;
  int thin=500;
  int i,j;
  gsl_rng *r = gsl_rng_alloc(gsl_rng_mt19937);
  double x=0;
  double y=0;
  printf("Iter x y\n");
  for (i=0;i<N;i++) {
    for (j=0;j<thin;j++) {
      x=gsl_ran_gamma(r,3.0,1.0/(y*y+4));
      y=1.0/(x+1)+gsl_ran_gaussian(r,1.0/sqrt(x+1));
    }
    printf("%d %f %f\n",i,x,y);
  }
}
```

This can be compiled and run (on Linux) with commands like:

```
gcc -O2 -lgsl -lgslcblas gibbs.c -o gibbs
time ./gibbs > data.tab
```

Java

Java is slightly nicer to code in than C, and has certain advantages over C in web application and cloud computing environments. It also has a few reasonable scientific libraries. I mainly use Parallel COLT, which (despite the name) is a general purpose scientific library, and not just for parallel applications. Using this library, code for this problem is given below.

```java
import java.util.*;

class Gibbs
{

    public static void main(String[] arg)
    {
    int N=20000;
    int thin=500;
    cern.jet.random.tdouble.engine.DoubleRandomEngine rngEngine=
        new cern.jet.random.tdouble.engine.DoubleMersenneTwister(new Date());
    cern.jet.random.tdouble.Normal rngN=
        new cern.jet.random.tdouble.Normal(0.0,1.0,rngEngine);
    cern.jet.random.tdouble.Gamma rngG=
        new cern.jet.random.tdouble.Gamma(1.0,1.0,rngEngine);
    double x=0;
    double y=0;
    System.out.println("Iter x y");
    for (int i=0;i<N;i++) {
        for (int j=0;j<thin;j++) {
        x=rngG.nextDouble(3.0,y*y+4);
        y=rngN.nextDouble(1.0/(x+1),1.0/Math.sqrt(x+1));
        }
        System.out.println(i+" "+x+" "+y);
    }
    }

}
```

Assuming that Parallel COLT is in your classpath, this can be compiled and run with commands like.

```
javac Gibbs.java
time java Gibbs > data.tab
```

Timings

Complex MCMC algorithms can be *really* slow to run (weeks or months), so execution speed really does matter. So, how do these languages compare for this particular simple problem? R is the slowest, so let's think about the speed relative to R (on my laptop). The Python code ran 2.4 times faster than R. To me, that is not really worth worrying too much about. Differences in coding style and speed-up tricks can make much bigger differences than this. So I wouldn't choose between these two on the basis of speed differential alone. Python is likely to be nicer to develop in, but the convenience of doing the computation in R, the standard platform for statistical analysis, could nevertheless tip things in favour of R. C was the fastest, and this is the reason that most of my MCMC code development has been traditionally done in C. It was around 60 times faster than R. This difference *is* worth worrying about, and can make the difference between an MCMC code being practical to run or not. So how about Java? Many people assume that Java must be much slower than C, as it is byte-code compiled, and runs in a virtual machine. In this case it is around 40 times faster than R. This is within a factor of 2 of C. This is quite typical in my experience, and I have seen codes which run *faster* in Java. Again, I wouldn't choose between C and Java purely on the basis of speed, but rather on speed of development and ease of deployment. Java can have advantages over C on both counts, which is why I've been experimenting more with Java for MCMC codes recently.

Summary

R and Python are slow, on account of their dynamic type systems, but are quick and easy to develop in. Java and C are much faster. The speed advantages of Java and C can be important in the context of complex MCMC algorithms. One possibility that is often put forward is to prototype in a dynamic language like R or Python and then port (the slow parts) to a statically typed language later. This could be time efficient, as the debugging and re-factoring can take place in the dynamic language where it is easier, then just re-coded fairly directly into the statically typed language once the code is working well. Actually, I haven't found this to be time efficient for the development of MCMC codes. That could be just me, but it could be a feature of MCMC algorithms in that the dynamic code runs too slowly to be amenable to quick and simple debugging.

RELATED:

Catalogue of my first 25 blog posts
With 1 comment

Gibbs sampler in various languages (re-visited)
In "byte-code"

Index to first 50 posts
In "50"

PUBLISHED BY

darrenjw

I am Professor of Stochastic Modelling within the School of Mathematics & Statistics at Newcastle University, UK. I am also a computational systems biologist. View all posts by darrenjw →

28/04/2010 darrenjw MCMC programming R Python C Java GSL scipy COLT Monte Carlo

33 thoughts on “MCMC programming in R, Python, Java and C”

elong
30/05/2010 at 10:34

Glad to see it.

crchrdsn
07/08/2010 at 21:31

Hello Darren,

I am glad I found your blog. I recently graduated with my MS in Statistics and I am interested in learning more about statistical computing. I have programmed a Gibbs sampler for Gaussian mixture- models in R for a final project in a machine learning course. This was lot of fun to do, although R is definitely slow for this type of work (even with vectorization). I am interested in doing more of this type of stuff with compiled languages like C/C++. I have dabbled in C a bit, I wrote an empirical distribution function in C that is callable from R (I know R already has the “ecdf” function). I was wondering what books or other information sources you might recommend for someone like me to become more proficient in statistical computing? My background is in Stats. I did take a programming course in CS using C++ though.

Thanks!

darrenjw
11/08/2010 at 18:43

I don’t know of much dealing specifically with statistical programming in C/C++. But there are quite a few decent books on scientific programming and algorithms more generally, and many of these are very relevant. Although I’m not a big fan of “Numerical recipes”, it’s not too bad a place to start. The “Matrix” book by Golub and van Loan covers linear algebra algorithms well. The source code for the GSL can give good ideas for how to approach things. After that, R source code (including, but not only, the C source) can be a good “source” of inspiration for statistical algorithms.

Pingback: Metropolis-Hastings MCMC algorithms « Darren Wilkinson's research blog

crchrdsn
17/08/2010 at 04:55

Thanks for these suggestions. The Golub and van Loan book does look excellent. There is a new book due out next May from CRC Press called Statistical Computing in C++ and R. I will most definitely get this as well.

Pingback: Getting started with parallel MCMC « Darren Wilkinson's research blog

Pingback: Calling C code from R « Darren Wilkinson's research blog

Pingback: Современные методы статистики | Boris Demeshev

olani debelo
16/05/2011 at 15:05

It is a good reading material

Carlos Rodriguez
27/05/2011 at 15:12

Hi Darren,

I am an R user as well and yes I agree for complex MCMC computations C is the best option. However, in C, when working with reversible jump MCMC methods the things get even harder because the dimension of parameters change. I am dealing with this right now and I can't find a work around. I am using malloc, realloc etc but it does not work. Do you have any suggestion to deal with this problems.

I already have the code in R but it is too slow.

Cheers,
Carlos

darrenjw
27/05/2011 at 15:35

I'm not sure that I said C was the _best_ option – just that it was fast. Manual management of memory in C is one of the things which makes it difficult, but also one of the things which makes it fast. You can use malloc, realloc, etc. to solve the reversible jump problem – it does work! ☺ The trick is to try to keep memory allocation to the absolute minimum. eg. when you exceed your max stack size, double it. This will ensure that you quickly get to a stage where you (almost) never need to allocate more memory. This will help to keep your code fast.

Carlos Rodriguez
27/05/2011 at 21:22

OK I misunderstood, you did not say that C was the best option just the fastest. I do not know if to try Java maybe I rather try FORTRAN. Yes C has that little drawback of the memory management. But after have waited 10 hours for my code to finish in R and just a few seconds or minutes in C: for me right now the priority is speed.

Cheers,
Carlos

Pingback: Java math libraries and Monte Carlo simulation codes « Darren Wilkinson's research blog

Chris L-M
05/06/2011 at 06:30

On the python front, there has been a lot of progress on pypy, the python interpreter with just-in-time compilation. Using it (pypy version 1.5-alpha0) with your python example above I get raw times of 9.75-9.83seconds where the C version runs in almost exactly 2.45s so the difference is speed is only about 4x faster for C.

I write most of my analyses in python and occasional dip down to C, fortran, or C++ code. As the C interfaces and other numerical library modules mature for pypy, I think it's going to be an even more amazing tool.

Kolmogorov
24/06/2011 at 04:58

I often use Groovy as a way to get scripting convenience (dynamic typing, no-compilation runs, etc.) for code that uses Java libraries. In this example, the Groovy code, using the same colt library, took 1.7 times as long as the Java program. Often I find that good enough and just stop there, but when I need that extra 1.7x speedup, it's a simple exercise to go through Groovy code, adding types and replacing Groovy-specific features with more verbose Java features. If you write things in Java, it's worth your while to experiment with Groovy as a rapid-prototyping solution.

Derek
24/06/2011 at 08:43

I'd echo Chris' comments… it would really great to have an update on this blog, with the inclusion of actual times, also addressing "compiled Python" with pypy.

In this case, the conclusion might be more like:

"R and dynamic Python are slow, on account of their dynamic type systems, but are quick and easy to develop in. Compiled Python is both fast and easy to develop in. Java and C are much faster than dynamic Python. However Java is only about 2.6 times faster than the compiled Python and that is not really worth worrying too much about, especially given the cost and effort of switching to another langauge."

darrenjw
24/06/2011 at 17:39

So, I'm quite busy right now, but for a while I have been considering doing a new version of this post with additional languages and more detailed timing information. I'll try and do it over the summer some time. I'll be running on a 64 bit Ubuntu system. If you comment with code examples for languages you would like me to include (and/or instructions for compilation), I'll do my best to include them.

darrenjw
25/06/2011 at 19:44

Thanks to Sanjog Misra, who has just emailed me to point out that the normal full conditional used in all of the codes isn't quite right – it should have a variance of 1/[2(1+x)], rather than 1/(1+x). It won't change any of the conclusions, but it's something that I'll fix when I get around to re-writing the post.

Derek
29/06/2011 at 10:49

Thanks Darren… we will wait!

Pingback: Gibbs sampler in various languages (revisited) « Darren Wilkinson's research blog

darrenjw
16/07/2011 at 16:11

OK, so there is now a new version of this post, which includes detailed timings, PyPy, and Scala:

https://darrenjw.wordpress.com/2011/07/16/gibbs-sampler-in-various-languages-revisited/

Please see the new post for further details, and try to comment on the new post in preference to this one. Thanks!

Pingback: Catalogue of my first 25 blog posts « Darren Wilkinson's research blog

Pingback: Statistics Made Accessible – R « betamoore

stayfnf

08/03/2012 at 12:36

You should be a part of a contest for one of the highest quality blogs on the net. I am going to highly recommend this blog!

Pingback: Olhando o MCMC mais de perto… « Recologia

Piggy
21/03/2013 at 13:27

How do you get the conditionals for x and y?
What if I want to sample from a distribution of which I have no analytical form?

darrenjw
22/03/2013 at 08:09

You get the full conditional from the joint by regarding the joint as a function of just the variable you are interested in, with everything else getting factored into the constant of proportionality, and hopefully recognising it as being of standard form. If it is not of standard form, you may find my posts on Metropolis-Hastings sampling helpful.

Piggy
23/03/2013 at 17:31

I think I got it. Once I have the posterior, the full conditionals are basically derived from that by keeping only the parameters I'm interested in and dropping the rest (since it would be constant wrt the parameter).
I'm gonna write a post on my blog to keep this in mind

Thanks very much indeed. Time to read your MH posts.
ahhh I love this blog

(oo)

lindonslog
29/06/2013 at 00:28

Hi Darren,

What made you opt for the GSL libraries for random number generation as opposed to other libraries, say, the R standalone library? Is there much of a speedup?

darrenjw
29/06/2013 at 08:46

There is a significant speed-up using the GSL code over the R library code, but the main reason is that the GSL is a good full featured, widely used, well-designed, all-round scientific library.

duyptnk
18/04/2015 at 17:03

Reblogged this on Statistics and Other Things.

Ayad

15/01/2017 at 20:55

Sorry Is there someone help me to writing the R code to draw a sample of the posterior distribution using Markov chains and gibbs sampler

Ayad

15/01/2017 at 21:20

My research proposal about nonlinear time series specifically about threshold autoregressive (TAR) model by using Bayes method .
Please,can you to help me (or sand my code) to estimate all parameters of threshold autoregressive (TAR) model.