2. Simulating simple Gaussian Processes

Simon Vaughan *

July 8, 2016

Details for generating Gaussian variables.

How does a function like `rmvnorm()` actually produce the Gaussian vector output with the right correlation structure?

We begin with $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ where all the x_i are independently distributed with a Gaussian distribution, mean $\mu = 0$ and variance $\sigma^2 = 1$ for all i. We can think of $\mathbf{x}$ as random vector with a Gaussian distribution, with mean vector $\boldsymbol{\mu} = 0$, and a simple covariance matrix $\Sigma = I_n$ where I_n is the $n \times n$ identity matrix.

$$cov(\mathbf{x}) = E[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T] = E[\mathbf{x}\mathbf{x}^T] = I_n \tag{1}$$

We wish to produce a vector $\mathbf{y} = (y_1, y_2, \ldots, y_n)^T$ which has the right covariance structure. To do this we left multiply the original random vector $\mathbf{x}$ by a matrix L such that the output, $\mathbf{y}$, has the right covariance matrix.

$$\mathbf{y} = L\mathbf{x} \tag{2}$$

The right matrix to chose is the Cholesky decomposition (the 'matrix square root') of S, the covariance matrix.

$$S = LL^T \tag{3}$$

L is an $n \times n$ lower triangular matrix. Now, we can check the covariance of $\mathbf{y}$ is what we want

$$cov(\mathbf{y}) = E[\mathbf{y}\mathbf{y}^T] \tag{4}$$

substituting $\mathbf{y} = L\mathbf{x}$ we get

$$= E[(L\mathbf{x})(L\mathbf{x})^T] \tag{5}$$

using $(L\mathbf{x})^T = \mathbf{x}^T L^T$ we get

$$= E[L\mathbf{x}\mathbf{x}^T L^T] \tag{6}$$

The matrix L is constant, so is just a constant factor on the expectation $E[\ldots]$

$$= LE[\mathbf{x}\mathbf{x}^T]L^T \tag{7}$$

*Email: sav2@le.ac.uk

Now we notice that $E[\mathbf{x}\mathbf{x}^T] = I_n$, the covariance matrix of the original vector $\mathbf{x}$.

$$= LI_nL^T \tag{8}$$
$$= LL^T \tag{9}$$
$$= S. \tag{10}$$

So to make $\mathbf{y}$ with the right covariance we need to

1. Define the covariance matrix S
2. Find its (lower) Cholesky decomposition L
3. Generate independent Gaussian variables $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$
4. Multiply by L to get $\mathbf{y} = L\mathbf{x}$.

First we define the times, and generate the (easy) random vector $\mathbf{x}$.

```
# define vector x
n <- 20
t <- seq(0, 10, length = n)
x <- rnorm(n, mean = 0, sd = 1)
```

Then we define the ACF

```
# define covariance function
acv <- function(tau, A, l) {
  acov <- A * exp(-0.5 * (tau / l)^2)
  return(acov)
}
```

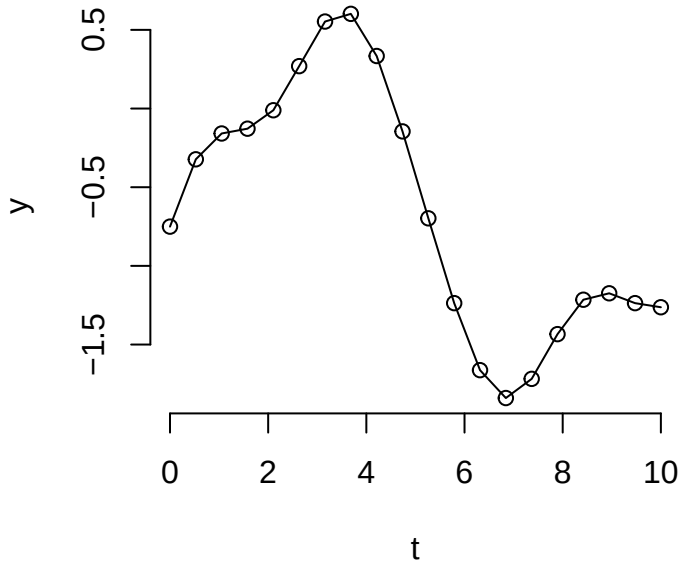
Then we define the mean $\boldsymbol{\mu} = 0$ and populate the elements of the desired covariance matrix S

```
mu <- array(0, dim = n)     # set all means to zero
tau <- outer(t, t, "-")     # compute t_j - t_i
tau <- abs(tau)             # compute |t_j - t_i|
S <- acv(tau, 1.0, 1.5)     # acf(tau)
```

Now we can compute the (lower) Cholesky decomposition and perform the multiplication with $\mathbf{x}$.

```
U <- chol(S)                # (upper) Cholesky decomposition
L <- t(U)                   # Flip to lower matrix
y <- L %*% x                # multiplication y = Lx

# make a plot
plot(t, y, type = "o", bty = "n")
```



Notice that the `chol()` produces an upper-right matrix. But as $U = L^T$ we transpose this using the transpose function `t()`. Then we use `%*%` for matrix multiplication.

Using R functions

The `rmvnorm()` function (from the `mvtnorm` package) creates random vectors with a single line. In fact it has three different methods to do this. The default uses the eigen-decomposition of the covariance matrix. But by setting `method = "chol"` it will use the Cholesky approach explained above.

There is another function, called `mvrnorm()` (in the `MASS` package[1]), which will produce random Gaussian vectors from a mean vector and covariance matrix. This uses the eigen-decomposition of the covariance matrix.

Let's compare these three by making $m = 200$ draws of an $n = 1000$ vector.

```
n <- 1000
m <- 200
t <- 1:n
tau <- outer(t, t, "-")  # compute t_j - t_i
tau <- abs(tau)          # compute |t_j - t_i|
S <- acv(tau, 1.0, 50)   # acf(tau)

# add a small 'epsilon' value on diagonal
diag(S) <- diag(S) + 0.0001
mu <- rep(0, 1000)       # mean values
```

[1] This is a 'base' R package, meaning it is already installed

```
# compute the randon values, record the compute time
system.time(
  y <- mvtnorm::rmvnorm(m, mean = mu, sigma = S)
  )

##    user  system elapsed
##    2.29    0.04    2.42
```

This is the `mvtnorm::rmvnorm` function, which by default uses the eigen-decomposition method. The output `y` is an $m \times n$ vector, each row is one of the n-dimensional vectors.

We can also ask it to use the Cholesky method:

```
system.time(
  y <- mvtnorm::rmvnorm(m, mean = mu, sigma = S, method = "chol")
  )

##    user  system elapsed
##    0.39    0.05    0.43
```

Notice how this is much faster. There is a third method, using SVD (singular value decomposition).

```
system.time(
  y <- mvtnorm::rmvnorm(m, mean = mu, sigma = S, method = "svd")
  )

##    user  system elapsed
##    3.81    0.06    3.89
```

This is slower. How about the `MASS::mvrnorm` function?

```
system.time(
  y <- MASS::mvrnorm(m, mu = mu, Sigma = S)
  )

##    user  system elapsed
##    2.21    0.03    2.25
```

Perhaps not surprisingly, this is similar to `mvtnorm::rmvnorm` using the (default) eigen-decomposition method. But what about doing the Cholesky method ourselves?

```
start <- proc.time()
# Cholesky decomposition of covariance matrix (n*n)
  L <- t(chol(S))
# generate n * m random numbers with Normal(0,1) distribution
  X <- rnorm(n*m, mean = 0, sd = 1)
# re-shape these into an n * m matrix
  dim(X) <- c(n, m)
# multiply n*n matrix L by n * m matrix X
```

```
  y <- L %*% X
# result is an n * m matrix.
finish <- proc.time()
print(finish-start)

##    user  system elapsed
##    0.39    0.01    0.41
```

This is comparable to `mvtnorm::rmvnorm` with the Cholesky method, and faster than the eigen-decomposition method. Also, the result is an $n \times m$ matrix, with each column is an n-dimensional vector (it's the transpose of the way that the other routines output the result).

If you look in the code for the `MASS::mvrnorm` and `mvtnorm::rmvnorm` functions – both of which are written in R – you will see there are many lines that 'check' the input data.

```
MASS::mvrnorm
mvtnorm::rmvnorm
```

These lines first check that the input mean and covariance matrix have the right dimensions (n and $n \times n$), and that the covariance matrix is symmetric and positive semi-definite. These additional checks may slow down the R functions a little, but make them 'safer' to use.

There is some advice available on the internet (e.g. Stack Exchange) suggesting that when the covariance matrix in nearly singular the Cholesky method is not very stable, and that the eigen-decomposition method can be better, but the (slower) SVD method is most stable. So – like with many numerical methods – there is a trade-off between speed and stability.