Simulation of a stationary Gaussian time-series

Robert B Davies

4 June, 2001

We are given the auto-covariances for a stationary Gaussian time-series observed at equal time intervals and wish to simulate this series. A method for doing this was described in Davies & Harte (1987).

I am often asked for more explanation. So here is some additional information and a C++ program using my libraries for implementing it.

The method has been rediscovered several times – our publication may have been second. For an earlier reference see Ripley (1987, page 110) and Davis et al (1981)[1]. For a later reference see Wood and Chan (1994) – but they give a generalisation to multiple dimensions.

Suppose we want to generate a stationary Gaussian time-series of length n+1 with mean 0 and auto-covariances $c_0, c_1, \ldots, c_n$. The method is as follows:

Find the finite Fourier transform of the sequence

$$\{c_0, c_1, \ldots, c_{n-1}, c_n, c_{n-1}, \ldots, c_1\}. \tag{1}$$

That is

$$g_k = \sum_{j=0}^{n-1} c_j \exp\left(\frac{2\pi ijk}{2n}\right) + \sum_{j=n}^{2n-1} c_{2n-j} \exp\left(\frac{2\pi ijk}{2n}\right) \tag{2}$$

for $k = 0,1,\ldots,2n-1$. You can use the discrete cosine transform for carrying out transform (2). The series $\{g_k\}$ will be real, but for the method to work it must also be non-negative. If any of the g_k are negative, the method fails. I haven't found this to be a problem. However, if any of the g_k are negative it may be possible to solve the problem by choosing a larger value of n and possibly adjusting the values of the auto-covariances at the centre of the sequence (1), for example c_{n-1}, c_n, c_{n-1}, to make the sequence smoother.

---

[1] Neither of these papers has the transform quite correct and neither notes that the method is exact for a process on a line (end not connected to the beginning) if you use only half of the *X*s. So perhaps we really were first. In any case, use the version given here rather than the versions given in those papers.

Generate a sequence of independent complex normal random numbers $\{Z_0, Z_1, \ldots, Z_{2n-1}\}$ where Z_0 and Z_n are real with variance 2; $\{Z_k : k = 1, \ldots, n-1\}$ have independent real and imaginary parts, each with variance 1 and $Z_k = \bar{Z}_{2n-k}$ for $n < k < 2n$.

Then the time-series

$$X_j = \frac{1}{2\sqrt{n}} \sum_{k=0}^{2n-1} Z_k \sqrt{g_k} \exp\left(\frac{2\pi ijk}{2n}\right) \tag{3}$$

for $0 \le j \le n$ has the required distribution. You can compute this using the inverse of the finite Fourier transform of a real sequence.

To prove that the method works first invert the Fourier transform (2):

$$c_j = \frac{1}{2n} \sum_{k=0}^{2n-1} g_k \exp\left(-\frac{2\pi ijk}{2n}\right) = \frac{1}{2n} \sum_{k=0}^{2n-1} g_k \exp\left(\frac{2\pi ijk}{2n}\right) \tag{4}$$

for $j = 0, 1, \ldots, n$.

The $\{X_j\}$ sequence (3) is real because the imaginary terms in its definition cancel. Hence

$$\operatorname{cov}(X_p, X_q) = E(X_p X_q) = E(X_p \bar{X}_q)$$

$$= \frac{1}{4n} \sum_{k=0}^{2n-1} \sum_{l=0}^{2n-1} E(Z_k \bar{Z}_l) \sqrt{g_k g_l} \exp\left(\frac{2\pi i(pk - ql)}{2n}\right). \tag{5}$$

Now $E(Z_k \bar{Z}_l) = 0$ if $k \ne l$ (by independence if $l \ne 2n - k$ and because $E(Z_k \bar{Z}_{2n-k}) = E(Z_k^2) = 0$). Also $E(Z_k \bar{Z}_k) = 2$. Hence

$$\operatorname{cov}(X_p, X_q) = \frac{1}{2n} \sum_{k=0}^{2n-1} g_k \exp\left(\frac{2\pi i(p-q)k}{2n}\right) = c_{p-q} \tag{6}$$

for $0 \le p - q \le n$ which is what we wanted.

Beran (1994) gives S-plus code for using this method for generating various long memory processes.

Here is a C++ program for generating fractional Gaussian noise using my matrix and random number programs.

See http://www.robertnz.net/ol_doc.htm for details of my libraries.

```
// Simulation of stationary Gaussian process

#define WANT_STREAM
#define WANT_MATH

#include "newmatap.h"       // newmat applications
#include "newmatio.h"       // newmat output package
#include "newran.h"         // random number library

// Stationary Gaussian process simulation class

class SGS
{
   ColumnVector G;          // to hold G(k)
   Normal normal;           // normal random number generator
   int N1;                  // n+1
public:
   SGS(const ColumnVector& AC);
   void Simulate(int n, ColumnVector& X);
};

// AC contains the auto-covariances - length must be odd
// AC(1) contains the variance
// AC(2) contains the lag 1 auto-covariance
// AC(3) contains the lag 2 auto-covariance etc

SGS::SGS(const ColumnVector& AC)
{
   N1 = AC.Nrows();
   if (!(N1 & 1)) Throw(Runtime_error("SGS: length of AC must be odd"));
   DCT(AC, G);                    // Cosine transform
   for (int k = 1; k <= N1; ++k)
   {
      double gk = G(k);
      if (gk < 0.0) Throw(Runtime_error("SGS: negative gk"));
      G(k) = sqrt(2.0 * gk);
   }
}

// n is the length of the sequence to be returned - must be no more than the
// length of AC
// The sequence is returned to X.

void SGS::Simulate(int n, ColumnVector& X)
{
   if (n > N1) Throw(Runtime_error("SGS: too many observations requested"));
   double Sqrt2 = sqrt(2.0);
   ColumnVector U(N1), V(N1);      // real and imaginary parts of Z array
   for (int k = 2; k < N1; ++k)
      { U(k) = G(k) * normal.Next(); V(k) = G(k) * normal.Next(); }
   U(1)  = Sqrt2 * G(1)  * normal.Next(); V(1)  = 0.0;
   U(N1) = Sqrt2 * G(N1) * normal.Next(); V(N1) = 0.0;
   RealFFTI(U, V, X);              // inverse of real FFT
   U.CleanUp(); V.CleanUp();       // release memory used by U and V
   X = X.Rows(1,n) * sqrt(N1-1);   // select first n values & rescale
}


int main()
{
   Try
   {
      Random::Set(0.3445821955);      // initialise RNG

      // Generate 100001 numbers from fractional Gaussian noise process
      int N1 = 100001;
      double H = 0.75;                // the value of H we are going to use
      double H2 = 2.0 * H;
      ColumnVector AC(N1);            // for the auto-correlations
      AC(1) = 1.0;
      for (int i = 1; i < N1; ++i)
      {
         // calculate auto-covariances - needs some more work
         if (i < 10000)
```

```
            AC(i+1) = 0.5 * (pow(i+1,H2) + pow(i-1,H2)) - pow(i, H2);
         else
            AC(i+1) = pow(i, H2-2) * H * (H2 - 1);
      }
      SGS Hurst(AC);                  // set up simulation structure
      ColumnVector X;                 // for the results
      Hurst.Simulate(N1, X);          // do simulation

      // print out the variance and first few auto-covariances
      cout << "variance   = " << X.SumSquare() / N1 << endl;
      cout << "cov: lag 1 = "
         << DotProduct(X.Rows(1,N1-1),X.Rows(2,N1)) / (N1-1) << endl;
      cout << "cov: lag 2 = "
         << DotProduct(X.Rows(1,N1-2),X.Rows(3,N1)) / (N1-2) << endl;
      cout << "cov: lag 3 = "
         << DotProduct(X.Rows(1,N1-3),X.Rows(4,N1)) / (N1-3) << endl;
      cout << "cov: lag 4 = "
         << DotProduct(X.Rows(1,N1-4),X.Rows(5,N1)) / (N1-4) << endl;
      cout << "cov: lag 5 = "
         << DotProduct(X.Rows(1,N1-5),X.Rows(6,N1)) / (N1-5) << endl;
      cout << "Theoretical values" << endl;
      cout << setw(11) << setprecision(6) << AC.Rows(1,6).t() << endl;
      return 0;
   }
   CatchAll
   {
      cout << "Simulation fails" << endl;
      cout << Exception::what() << endl;
      exit(1);
   }
}
```

References

Beran, Jan (1994). *Statistics for long-memory processes*. Chapman & Hall, New York.

Davies, R.B. & Harte, D.S. (1987). Tests for Hurst effect. *Biometrika* **74**, 95-101.

Davis, B.M., Hagan, R. & Borgman, L.E. (1981). A program for the finite Fourier transform simulation of realizations from a one-dimensional random function with known covariance. *Computers & Geosciences* **7**, 199-206.

Ripley, B.D. (1987). *Stochastic Simulation*, Wiley, New York.

Wood, A.T.A. & Chan, Grace (1994). Simulation of Stationary Gaussian Processes in $[0,1]^d$. *Journal of Computational and Graphical Statistics* **3**, 409-432.