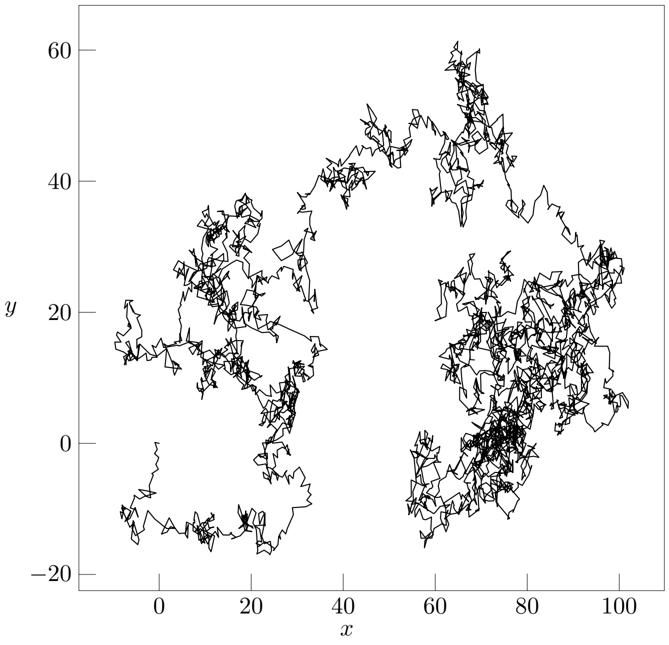
December 16, 2011 at 21:32

1. Introduction.

Wiener

A computer program for the generation of numerical data simulating a Wiener process.
(Version 1.0 of November 11, 2011)

Written by Fredrik Jonsson



This CWEB[†] computer program computes a series of floating-point numbers corresponding to a Wiener process in D dimensions. It relies on the random number generator as proposed by Donald Knuth in *The Art of Computer Programming, Volume 1 – Fundamental Algorithms*, 3rd edition (Addison-Wesley, Boston, 1998), generating numbers which are fed into the Box–Muller transform to generate the normal distribution associated with the Wiener process. Besides providing a simulator of the Wiener process, the WIENER program can also be used in a "lock-in" mode with zero expectation value for each data point, providing a pretty good random number generator for large series of stochastic data, not relying on the (rather poor) generators available in standard C libraries.

The WIENER program does not solve any problem *per se*, but is merely to be considered as a generator of statistical data to be used by other applications in modeling of physical, chemical or financial processes.

[†] For information on the CWEB programming language by Donald E. Knuth, as well as samples of CWEB programs, see `http://www-cs-faculty.stanford.edu/~knuth/cweb.html`. For general information on literate programming, see `http://www.literateprogramming.com`.

2. The Wiener process. "In mathematics, the Wiener process is a continuous-time stochastic process named in honor of Norbert Wiener. It is often called standard Brownian motion, after Robert Brown. It is one of the best known Lévy processes (càdlàg stochastic processes with stationary independent increments) and occurs frequently in pure and applied mathematics, economics and physics.

The Wiener process plays an important role both in pure and applied mathematics. In pure mathematics, the Wiener process gave rise to the study of continuous time martingales. It is a key process in terms of which more complicated stochastic processes can be described. As such, it plays a vital role in stochastic calculus, diffusion processes and even potential theory. It is the driving process of Schramm–Loewner evolution. In applied mathematics, the Wiener process is used to represent the integral of a Gaussian white noise process, and so is useful as a model of noise in electronics engineering, instrument errors in filtering theory and unknown forces in control theory.

The Wiener process has applications throughout the mathematical sciences. In physics it is used to study Brownian motion, the diffusion of minute particles suspended in fluid, and other types of diffusion via the Fokker–Planck and Langevin equations. It also forms the basis for the rigorous path integral formulation of quantum mechanics (by the Feynman–Kac formula, a solution to the Schrödinger equation can be represented in terms of the Wiener process) and the study of eternal inflation in physical cosmology. It is also prominent in the mathematical theory of finance, in particular the Black–Scholes option pricing model."

—Wikipedia, "Wiener process" (2011)

3. What the WIENER program does (and doesn't). The present CWEB program does not solve any problems related to any of the processes described by models involving the Wiener process, but is merely an attempt to produce an as-good-as-possible result when simulating the Wiener process as such. In the WIENER program, special attention has been paid to the generation of random numbers, as this is a crucial and rather tricky problem when it comes to generating large non-recurring series of data. In the present program, the random number generator proposed by Donald Knuth† has been employed, generating uniformly distributed numbers which are fed into the Box–Muller transform to generate the normal distribution associated with the Wiener process.

Apart from being a pretty good and reliable generator of statistical data, to be used by other applications in modeling of physical, chemical or financial processes, the WIENER program does not solve any problems *per se*.

† Donald E. Knuth, *The Art of Computer Programming, Volume 1 – Fundamental Algorithms*, 3rd edition (Addison-Wesley, Boston, 1998).

4. The CWEB programming language. For the reader who might not be familiar with the concept of the CWEB programming language, the following citations hopefully will be useful. For further information, as well as freeware compilers for compiling CWEB source code, see `http://www.literateprogramming.com`.

I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: 'Literate Programming.'

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.

– Donald Knuth, *The CWEB System of Structured Documentation* (Addison-Wesley, Massachusetts, 1994)

The philosophy behind CWEB is that an experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like TEX for formatting, and a language like C for programming. Neither type of language can provide the best documentation by itself; but when both are appropriately combined, we obtain a system that is much more useful than either language separately.

The structure of a software program may be thought of as a 'WEB' that is made up of many interconnected pieces. To document such a program we want to explain each individual part of the web and how it relates to its neighbors. The typographic tools provided by TEX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by languages like C make it possible for us to specify the algorithms formally and unambiguously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation.

Besides providing a documentation tool, CWEB enhances the C language by providing the ability to permute pieces of the program text, so that a large system can be understood entirely in terms of small sections and their local interrelationships. The CTANGLE program is so named because it takes a given web and moves the sections from their web structure into the order required by C; the advantage of programming in CWEB is that the algorithms can be expressed in "untangled" form, with each section explained separately. The CWEAVE program is so named because it takes a given web and intertwines the TEX and C portions contained in each section, then it knits the whole fabric into a structured document.

– Donald Knuth, "Literate Programming", in *Literate Programming* (CSLI Lecture Notes, Stanford, 1992)

5. Revision history of the program.

2011-11-11 [v.1.0] `<http://jonsson.eu/programs/cweb/>`
First properly working version of the WIENER program, written in CWEB and (ANSI-conformant) C.

6. Compiling the source code. The program is written in CWEB, generating ANSI C (ISO C90) conforming source code and documentation as plain TEX-source, and is to be compiled using the sequences as outlined in the Makefile listed below.

```
#
# Makefile designed for use with ctangle, cweave, gcc, and plain TeX.
#
# Copyright (C) 2002-2011, Fredrik Jonsson <http://jonsson.eu>
#
# The CTANGLE program converts a CWEB source document into a C program
# which may be compiled in the usual way.  The output file includes #line
# specifications so that debugging can be done in terms of the CWEB source
# file.
#
# The CWEAVE program converts the same CWEB file into a TeX file that may
# be formatted and printed in the usual way.  It takes appropriate care of
# typographic details like page layout and the use of indentation, italics,
# boldface, etc., and it supplies extensive cross-index information that it
# gathers automatically.
#
# CWEB allows you to prepare a single document containing all the informa-
# tion that is needed both to produce a compilable C program and to produce
# a well-formatted document describing the program in as much detail as the
# writer may desire.    The user of CWEB ought to be familiar with TeX as well
# as C.
#
PROJECT  = wiener
FIGURES  = fig1.eps fig2.eps fig3.eps

CTANGLE  = ctangle
CWEAVE   = cweave
CC       = gcc
CCOPTS   = -O2 -Wall -ansi -std=iso9899:1990 -pedantic
LNOPTS   = -lm
TEX      = tex
DVIPS    = dvips
DVIPSOPT = -ta4 -D1200
PS2PDF   = ps2pdf
METAPOST = mpost

all:  $(PROJECT) $(FIGURES) $(PROJECT).pdf

$(PROJECT): $(PROJECT).o
	$(CC) $(CCOPTS) -o $(PROJECT) $(PROJECT).o $(LNOPTS)

$(PROJECT).o:  $(PROJECT).c
	$(CC) $(CCOPTS) -c $(PROJECT).c

$(PROJECT).c:  $(PROJECT).w
	$(CTANGLE) $(PROJECT)
```

```
$(PROJECT).pdf:  $(PROJECT).ps
        $(PS2PDF) $(PROJECT).ps $(PROJECT).pdf

$(PROJECT).ps:  $(PROJECT).dvi
        $(DVIPS) $(DVIPSOPT) $(PROJECT).dvi -o $(PROJECT).ps

$(PROJECT).dvi:  $(PROJECT).tex
        $(TEX) $(PROJECT).tex

$(PROJECT).tex:  $(PROJECT).w
        $(CWEAVE) $(PROJECT)

#
# Generate the Encapsulated PostScript image fig1.eps for the documentation.
# This is a 2D scatter plot of the uniformly distributed pseudo-random numbers
# prior to having been fed into the Box-Muller transform.
#
fig1.eps:  Makefile $(PROJECT).w
        wiener --uniform -D 2 -M 10000 > fig1.dat;
        cho "input graph;\
                def mpdot = btex\
                   \vrule height 0.5pt width 1.0pt depth 0.5pt\
                etex enddef;\
                beginfig(1);\
                draw begingraph(86mm,86mm);\
                setrange(0,0,1,1);\
                pickup pencircle scaled .5pt;\
                gdraw \"fig1.dat\" plot mpdot;\
                pickup pencircle scaled .25pt;\
                autogrid(itick bot,itick lft);\
                glabel.bot(btex $$ x$$ etex,OUT);\
                glabel.lft(btex $$ y$$ etex,OUT);\
                endgraph; endfig; end" > fig1.mp
                $(METAPOST) fig1.mp
        $(TEX) -jobname=fig1 "\input epsf\nopagenumbers\
            \centerline\epsfboxfig1.1\bye"
        $(DVIPS) -D1200 -E fig1.dvi -o fig1.eps

#
# Generate the Encapsulated PostScript image fig2.eps for the documentation.
# This is a 2D scatter plot of the normally distributed pseudo-random numbers
# resulting from the Box-Muller transform.
#
```

```
fig2.eps:  Makefile $(PROJECT).w
         wiener --normal -D 2 -M 10000 > fig2.dat;
         cho "input graph;\
                  def mpdot = btex\
                     \vrule height 0.5pt width 1.0pt depth 0.5pt\
                  etex enddef;\
                  beginfig(1);\
                  draw begingraph(86mm,86mm);\
                  setrange(whatever,whatever,whatever,whatever);\
                  pickup pencircle scaled .5pt;\
                  gdraw \"fig2.dat\" plot mpdot;\
                  pickup pencircle scaled .25pt;\
                  autogrid(itick bot,itick lft);\
                  glabel.bot(btex $$ x$$ etex,OUT);\
                  glabel.lft(btex $$ y$$ etex,OUT);\
                  endgraph; endfig; end" > fig2.mp
                  $(METAPOST) fig2.mp
         $(TEX) -jobname=fig2 "\input epsf\nopagenumbers\
             \centerline\epsfboxfig2.1\bye"
         $(DVIPS) -D1200 -E fig2.dvi -o fig2.eps

#
# Generate the Encapsulated PostScript image fig3.eps for the documentation.
# This is a 2D graph showing the resulting simulated Wiener process.
#
fig3.eps:  Makefile $(PROJECT).w
         wiener -D 2 -M 10000 > fig3.dat;
         cho "input graph;\
                  beginfig(1);\
                  draw begingraph(86mm,86mm);\
                  setrange(whatever,whatever,whatever,whatever);\
                  pickup pencircle scaled .5pt;\
                  gdraw \"fig3.dat\";\
                  pickup pencircle scaled .25pt;\
                  autogrid(itick bot,itick lft);\
                  glabel.bot(btex $$ x$$ etex,OUT);\
                  glabel.lft(btex $$ y$$ etex,OUT);\
                  endgraph; endfig; end" > fig3.mp
                  $(METAPOST) fig3.mp
         $(TEX) -jobname=fig3 "\input epsf\nopagenumbers\
             \centerline\epsfboxfig3.1\bye"
         $(DVIPS) -D1200 -E fig3.dvi -o fig3.eps

clean:
         -rm -Rf $(PROJECT) *  *.c *.o *.exe *.dat *.pdf *.mp *.trj *.mpx
         -rm -Rf *.tex *.aux *.log *.toc *.idx *.scn *.dvi *.ps *.1 *.eps

archive:
         make -ik clean
         tar --gzip --directory=../ -cf ../$(PROJECT).tgz $(PROJECT)
```

This `Makefile` essentially executes two major calls. First, the CTANGLE program parses the CWEB source document `wiener.w` to extract a C source file `wiener.c` which may be compiled in the usual way using any ANSI C conformant compiler. The output source file includes `#line` specifications so that any debugging can be done conveniently in terms of the original CWEB source file. Second, the CWEAVE program parses the same CWEB source file to extract a plain TEX source file `wiener.tex` which may be compiled in the usual way. It takes appropriate care of typographic details like page layout and the use of indentation, italics, boldface, and so on, and it supplies extensive cross-index information that it gathers automatically.

After having executed `make` in the same catalogue where the files `wiener.w` and `Makefile` are located, one is left with an executable file `wiener`, being the ready-to-use compiled program, and a PostScript file `wiener.ps` which contains the full documentation of the program, that is to say the document you currently are reading. On platforms running any operating system by Microsoft, the executable file will instead automatically be called `wiener.exe`. This convention also applies to programs compiled under the UNIX-like environment CYGWIN.

7. Running the program. The program is entirely controlled by the command line options supplied when invoking the program. The syntax for executing the program is

```
wiener [options]
```

where `options` include the following, given in their long ('`--`') as well as their short ('`-`') forms:

`--help`, `-h`
: Display a brief help message and exit clean.

`--verbose`, `-v`
: Toggle verbose mode. Default: off.

`--num_samples`, `-M` M
: Generate M samples of data. Here M should always be an even number, greater than the long lag `KK`. If an odd number is specified, the program will automatically adjust this to the next higher integer. Default: $M =$ `KK` $= 100$.

`--dimension`, `-D` D
: Specifies the dimension D of the Wiener process, that is to say generating a set of D numbers for each of the M data points in the seqence. Default: $D = 1$.

`--seed`, `-s` S
: Define a custom seed number S for the initialization of the random number generator. Default: $S =$ `DEFAULT_SEED` $= 310952$.

`--uniform`, `-u`
: Instead of generating a sequence of data corresponding to a Wiener process, lock the program to simply generate a uniform distribution of D-dimensional points, with each element distributed over the interval $[0, 1]$.

`--normal`, `-n`
: Instead of generating a sequence of data corresponding to a Wiener process, lock the program to simply generate a normal distribution of D-dimensional points, with each element distributed with zero expectation value and unit variance.

One may look upon the two last options as verification options, generating data suitable for spectral tests on the quality of the generator of pseudo-random numbers.

8. Plotting the results using GNUPLOT. The data sets generated by WIENER may be displayed and saved as Encapsulated PostScript images using, say, GNUPLOT or METAPOST. While I personally prefer MetaPost, mainly due to the possibility of incorporating the same typygraphic elements as in TEX, many people consider GNUPLOT to be easier in operation.

In order to save a scatter graph as Encapsulated PostScript using GNUPLOT, you may include the following calls in, say, a `Makefile` or a shell script:

```
wiener -D 2 -M 10000 > figure.dat;
echo "set term postscript eps;\
       set output ”figure.eps”;\
       set size square;\
       set nolabel;\
       plot ”figure.dat” with dots notitle;\
       quit" | gnuplot
```

9. Plotting the results using METAPOST. Another choice is to go for the METAPOST way. This is illustrated with the following blocks, taken directly from the enclosed Makefile and generating the figures which can be seen in the section relating to the generation of normally distributed variables (routine *normdist*):

```
PROJECT  = wiener
TEX      = tex
DVIPS    = dvips
METAPOST = mpost

#
# Generate the Encapsulated PostScript image fig1.eps for the documentation.
# This is a 2D scatter plot of the uniformly distributed pseudo-random numbers
# prior to having been fed into the Box-Muller transform.
#
fig1.eps:  Makefile $(PROJECT).w
         wiener --uniform -D 2 -M 10000 > fig1.dat;
         cho "input graph;\
                  def mpdot = btex\
                     \vrule height 0.5pt width 1.0pt depth 0.5pt\
                  etex enddef;\
                  beginfig(1);\
                  draw begingraph(86mm,86mm);\
                  setrange(0,0,1,1);\
                  pickup pencircle scaled .5pt;\
                  gdraw \"fig1.dat\" plot mpdot;\
                  pickup pencircle scaled .25pt;\
                  autogrid(itick bot,itick lft);\
                  glabel.bot(btex $$ x$$ etex,OUT);\
                  glabel.lft(btex $$ y$$ etex,OUT);\
                  endgraph; endfig; end" > fig1.mp
                  $(METAPOST) fig1.mp
         $(TEX) -jobname=fig1 "\input epsf\nopagenumbers\
             \centerline\epsfboxfig1.1\bye"
         $(DVIPS) -D1200 -E fig1.dvi -o fig1.eps

#
# Generate the Encapsulated PostScript image fig2.eps for the documentation.
# This is a 2D scatter plot of the normally distributed pseudo-random numbers
# resulting from the Box-Muller transform.
#
```

```
fig2.eps:  Makefile $(PROJECT).w
         wiener --normal -D 2 -M 10000 > fig2.dat;
         cho "input graph;\
                  def mpdot = btex\
                     \vrule height 0.5pt width 1.0pt depth 0.5pt\
                  etex enddef;\
                  beginfig(1);\
                  draw begingraph(86mm,86mm);\
                  setrange(whatever,whatever,whatever,whatever);\
                  pickup pencircle scaled .5pt;\
                  gdraw \"fig2.dat\" plot mpdot;\
                  pickup pencircle scaled .25pt;\
                  autogrid(itick bot,itick lft);\
                  glabel.bot(btex $$ x$$ etex,OUT);\
                  glabel.lft(btex $$ y$$ etex,OUT);\
                  endgraph; endfig; end" > fig2.mp
                  $(METAPOST) fig2.mp
         $(TEX) -jobname=fig2 "\input epsf\nopagenumbers\
             \centerline\epsfboxfig2.1\bye"
         $(DVIPS) -D1200 -E fig2.dvi -o fig2.eps

#
# Generate the Encapsulated PostScript image fig3.eps for the documentation.
# This is a 2D graph showing the resulting simulated Wiener process.
#
fig3.eps:  Makefile $(PROJECT).w
         wiener -D 2 -M 10000 > fig3.dat;
         cho "input graph;\
                  beginfig(1);\
                  draw begingraph(86mm,86mm);\
                  setrange(whatever,whatever,whatever,whatever);\
                  pickup pencircle scaled .5pt;\
                  gdraw \"fig3.dat\";\
                  pickup pencircle scaled .25pt;\
                  autogrid(itick bot,itick lft);\
                  glabel.bot(btex $$ x$$ etex,OUT);\
                  glabel.lft(btex $$ y$$ etex,OUT);\
                  endgraph; endfig; end" > fig3.mp
                  $(METAPOST) fig3.mp
         $(TEX) -jobname=fig3 "\input epsf\nopagenumbers\
             \centerline\epsfboxfig3.1\bye"
         $(DVIPS) -D1200 -E fig3.dvi -o fig3.eps
```

10. The main program. Here follows the general outline of the main program.

```
⟨ Library inclusions 11 ⟩
⟨ Global definitions 12 ⟩
⟨ Global variables 13 ⟩
⟨ Routines 14 ⟩
int main(int argc, char *argv[])
{
  ⟨ Declaration of local variables 24 ⟩
  ⟨ Parse command line for parameters 25 ⟩
  ⟨ Allocate memory for a vector containing M × D elements 26 ⟩
  ⟨ Fill vector with M number of D:tuples describing the Wiener process 27 ⟩
  ⟨ Print the generated vector at standard terminal output 28 ⟩
  ⟨ Deallocate the memory occupied by the vector of M × D elements 29 ⟩
  return (SUCCESS);
}
```

11. Library dependencies. The standard ANSI C libraries included in this program are:

`math.h`	For access to common mathematical functions.
`stdio.h`	For file access and any block involving *fprintf*.
`stdlib.h`	For access to the *exit* function.
`string.h`	For string manipulation, *strcpy*, *strcmp* etc.
`ctype.h`	For access to the *isalnum* function.

```
⟨ Library inclusions 11 ⟩ ≡
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

This code is used in section 10.

12. Global definitions. These are the global definitions present in the WIENER program:

`VERSION`	The current program revision number.
`COPYRIGHT`	The copyright banner.
`SUCCESS`	The return code for successful program termination.
`FAILURE`	The return code for unsuccessful program termination.
`QUALITY`	The recommended "quality level" for high-resolution use, according to Knuth. Used by the *ranf_array* routine.
`KK`	The "long lag" used by routines *ranf_array* and *ranf_start*.
`LL`	The "short lag" used by routines *ranf_array* and *ranf_start*.
`DEFAULT_SEED`	The default seed to use when initializing the random number generator, using the routine *ranf_start*. The seed can be hand-picked using the `--seed` command-line option.
`cmd_match(s,l,c)`	Check if the string s and/or l matches the string c. This short-hand macro is used when parsing the command line for options.

⟨ Global definitions 12 ⟩ ≡
```
#define VERSION "1.0"
#define COPYRIGHT "Copyright␣(C)␣2011,␣Fredrik␣Jonsson␣<http://jonsson.eu>"
#define SUCCESS (0)
#define FAILURE (1)
#define QUALITY (1009)
#define KK (100)
#define LL (37)
#define DEFAULT_SEED (310952)
#define cmd_match(s, l, c) ((¬strcmp((c), (s))) ∨ (¬strcmp((c), (l))))
#define MODE_WIENER_PROCESS (0)
#define MODE_LOCKED_UNIFORM_DISTRIBUTION (1)
#define MODE_LOCKED_NORMAL_DISTRIBUTION (2)
```
This code is used in section 10.

13. Declaration of global variables. Usually, the only global variables allowed in my programs are *optarg*, which is a pointer to the the string of characters that specified the call from the command line, and *progname*, which simply is a pointer to the string containing the name of the program, as it was invoked from the command line. However, as Donald Knuth has a *faiblesse* for global variables, I have for the sake of consistency with the routines related to the random number generator kept his definitions in a global scope.

⟨ Global variables 13 ⟩ ≡
```
  extern char *optarg;
  char *progname;
  double ranf_arr_buf[QUALITY];
  double ranf_arr_dummy = −1.0, ranf_arr_started = −1.0;
  double *ranf_arr_ptr = &ranf_arr_dummy;     /* the next random fraction, or -1 */
```
This code is used in section 10.

14. Declaration of routines. These routines exclusively concern the generation of random numbers and the generation of normally distributed data points in the Wiener process.

⟨ Routines 14 ⟩ ≡
  ⟨ Routine for generation of uniformly distributed random numbers 15 ⟩
  ⟨ Routine for initialization of the random number generator 16 ⟩
  ⟨ Routine for generation of normally distributed variables 17 ⟩
  ⟨ Routine for generation of numerical data describing a Wiener process 18 ⟩
  ⟨ Routine for memory allocation 19 ⟩
  ⟨ Routine for memory de-allocation 20 ⟩
  ⟨ Routine for displaying a help message at terminal output 21 ⟩
  ⟨ Routine for checking valid path characters 22 ⟩
  ⟨ Routine for stripping away path string 23 ⟩

This code is used in section 10.

15. Generation of uniformly distributed random numbers. We here avoid relying on the standard functions available in C,[†] but rather take resort to the algorithm devised by Donald Knuth in *The Art of Computer Programming, Volume 1 – Fundamental Algorithms*, 3rd edition, Section 3.6. (Addison–Wesley, Boston, 1998).[‡] The variables to the routine are as follows:

`double aa[]`	On return, this array contains n new random numbers, following the recurrence $X_j = (X_{j-100} - X_{j-37}) \bmod 2^{30}$. Not used on input.
`double n`	The number n of random numbers to be generated. This array length must be at least `KK` elements.

The *mod_sum*(x, y) macro is here merely a shorthand for "$(x + y) \bmod 1.0$."

⟨ Routine for generation of uniformly distributed random numbers 15 ⟩ ≡

```
#define mod_sum(x, y)  (((x) + (y)) − (int)((x) + (y)))
  double ran_u[KK];      /* the generator state */
  void ranf_array(double aa[], int n)      /* put n new random fractions in aa */
  {
    register int i, j;
    for (j = 0; j < KK; j++)  aa[j] = ran_u[j];
    for ( ; j < n; j++)  aa[j] = mod_sum(aa[j − KK], aa[j − LL]);
    for (i = 0; i < LL; i++, j++)  ran_u[i] = mod_sum(aa[j − KK], aa[j − LL]);
    for ( ; i < KK; i++, j++)  ran_u[i] = mod_sum(aa[j − KK], ran_u[i − LL]);
  }
  void ranf_matrix(double **aa, int mm, int dd)
  {
    register int i, j, col;
    for (col = 0; col < dd; col++) {
      for (j = 0; j < KK; j++)  aa[j][col] = ran_u[j];
      for ( ; j < mm; j++)  aa[j][col] = mod_sum(aa[j − KK][col], aa[j − LL][col]);
      for (i = 0; i < LL; i++, j++)  ran_u[i] = mod_sum(aa[j − KK][col], aa[j − LL][col]);
      for ( ; i < KK; i++, j++)  ran_u[i] = mod_sum(aa[j − KK][col], ran_u[i − LL]);
    }
  }
```

This code is used in section 14.

[†] Here only included as a reference, the (primitive) standard C way of generating random integer numbers, in this case initializing with a simple `srand(time(NULL))` and generating random numbers between 0 and `RAND_MAX`, is

```
  int rand_stdc()
  srand(time(NULL)); /* Initialize random number generator */
  return(rand());
```

I have personally *not* (yet) checked this approach using the spectral tests recommended by Knuth.

[‡] The credit for this random number generator, which employs double floating-point precision rather than the original integer version, goes entirely to Donald Knuth and the persons who contributed. The original source code are available at `http://www-cs-faculty.stanford.edu/~knuth/programs/rng-double.c`. The current routine takes into account changes as explained in the errata to the 2nd edition, see `http://www-cs-faculty.stanford.edu/~knuth/taocp.html` in the changes to Volume 2 on pages 171 and following.

16. Initialization of the random number generator. To quote Knuth, “The tricky thing about using a recurrence like $[X_j = (X_{j-100} - X_{j-37}) \bmod 2^{30}]$ is, of course, to get everuthing started properly in the first place, by setting up suitable values of $X_0, \ldots, X_{99}$. The following routine *ran_start* initializes the generator nicely when given any seed number between 0 and $2^{30} - 3 = 1,073,741,821$ inclusive.” Here we rather employ the double precision variant of the initialization to match the data type of *ran_array*.

Again, the credits for the *ranf_start* routine goes entirely to Donald Knuth and the persons who contributed.

⟨ Routine for initialization of the random number generator 16 ⟩ ≡

```
#define TT 70      /* guaranteed separation between streams */
#define is_odd(s) ((s) & 1)
  void ranf_start(long seed)
  {
    register int t, s, j;
    double u[KK + KK − 1];
    double ulp = (1.0/(1L ≪ 30))/(1L ≪ 22);      /* 2 to the -52 */
    double ss = 2.0 ∗ ulp ∗ ((seed & #3fffffff) + 2);
    for (j = 0; j < KK; j++) {
      u[j] = ss;      /* bootstrap the buffer */
      ss += ss;
      if (ss ≥ 1.0) ss −= 1.0 − 2 ∗ ulp;      /* cyclic shift of 51 bits */
    }
    u[1] += ulp;      /* make u[1] (and only u[1]) ”odd” */
    for (s = seed & #3fffffff, t = TT − 1; t; ) {
      for (j = KK − 1; j > 0; j−−) u[j + j] = u[j], u[j + j − 1] = 0.0;      /* ”square” */
      for (j = KK + KK − 2; j ≥ KK; j−−) {
        u[j − (KK − LL)] = mod_sum(u[j − (KK − LL)], u[j]);
        u[j − KK] = mod_sum(u[j − KK], u[j]);
      }
      if (is_odd(s)) {      /* ”multiply by z” */
        for (j = KK; j > 0; j−−) u[j] = u[j − 1];
        u[0] = u[KK];      /* shift the buffer cyclically */
        u[LL] = mod_sum(u[LL], u[KK]);
      }
      if (s) s ≫= 1;
      else t−−;
    }
    for (j = 0; j < LL; j++) ran_u[j + KK − LL] = u[j];
    for ( ; j < KK; j++) ran_u[j − LL] = u[j];
    for (j = 0; j < 10; j++) ranf_array(u, KK + KK − 1);      /* warm things up */
    ranf_arr_ptr = &ranf_arr_started;
  }
#define ranf_arr_next() (∗ranf_arr_ptr ≥ 0 ? ∗ranf_arr_ptr++ : ranf_arr_cycle())
  double ranf_arr_cycle()
  {
    if (ranf_arr_ptr ≡ &ranf_arr_dummy) ranf_start(314159L);      /* the user forgot to initialize */
    ranf_array(ranf_arr_buf, QUALITY);
    ranf_arr_buf[KK] = −1;
    ranf_arr_ptr = ranf_arr_buf + 1;
    return ranf_arr_buf[0];
  }
```

This code is used in section 14.

17. Generation of normally distributed variables. Accepting uniformly distributed random numbers as input, the Box–Muller transform creates a set of normally distributed numbers. This transform originally appeared in G. E. P. Box and Mervin E. Muller, *A Note on the Generation of Random Normal Deviates*, Annals Math. Stat. **29**, 610–611 (1958).

In Donald Knuth's *The Art of Computer Programming, Volume 1 – Fundamental Algorithms*, 3rd edition (Addison–Wesley, Boston, 1998), the Box–Muller method is denoted the polar method and is described in detail in Section 3.4.1, Algorithm P, on page 122.

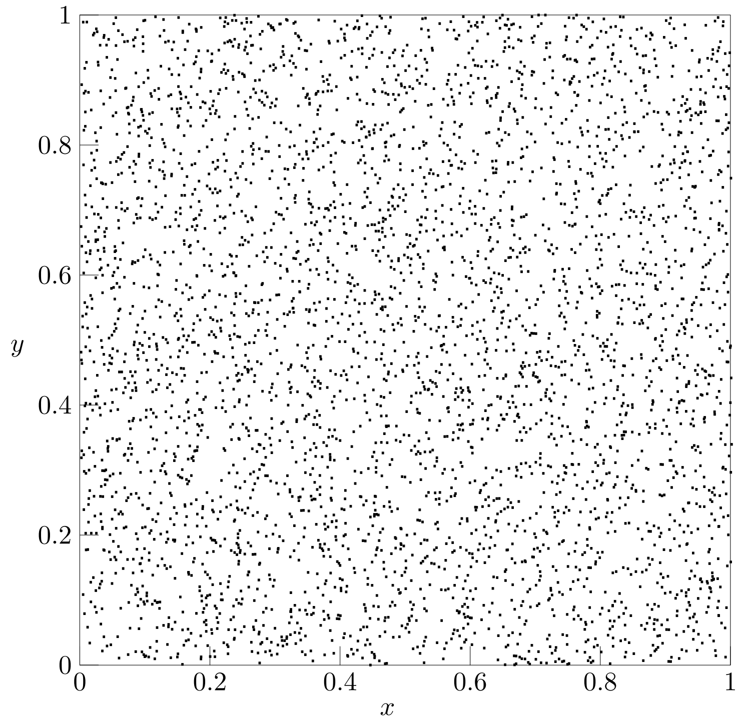


Figure 1. Sample two-dimensional output from the *ranf_matrix* routine, in this case 10 000 data points uniformly distributed over the domain $0 \leq \{x, y\} \leq 1$. The data for this graph was generated by WIENER using `wiener --uniform -D 2 -M 10000 > fig1.dat`. See the `fig1.eps` block in the enclosed `Makefile` for details on how METAPOST was used in the generation of the encapsulated PostScript image of the graph.

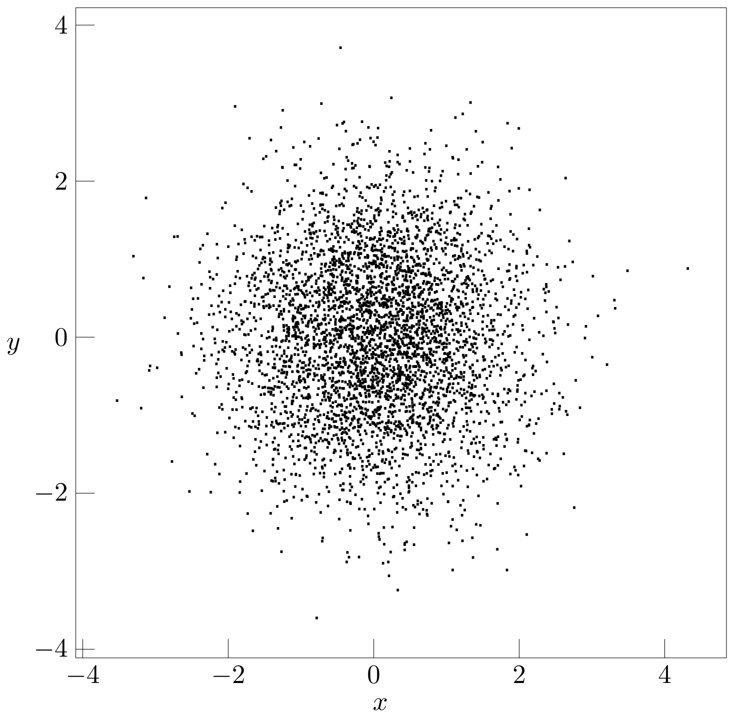


Figure 2. The same data points as in Fig. 1, but after having applied the Box–Muller transform to yield a normal distribution of pseudo-random numbers. The data for this graph was generated by WIENER using `wiener --normal -D 2 -M 10000 > fig2.dat`. See the `fig2.eps` block in the enclosed `Makefile` for details on how METAPOST was used in the generation of the encapsulated PostScript image of the graph.

⟨ Routine for generation of normally distributed variables 17 ⟩ ≡

```
#define PI (3.14159265358979323846264338)
  void normdist(double **aa, int mm, int dd)
  {
    register int j, k;
    register double f, z;
    for (j = 0; j < dd; j++) {
      for (k = 0; k < mm - 1; k += 2) {
        if (aa[k][j] > 0.0) {
          f = sqrt(-2 * log(aa[k][j]));
          z = 2.0 * PI * aa[k + 1][j];
          aa[k][j] = f * cos(z);
          aa[k + 1][j] = f * sin(z);
        }
        else {
          fprintf(stderr, "%s:␣Error:␣Zero␣element␣detected!\n", progname);
          fprintf(stderr, "%s:␣(row␣%d,␣column␣%d)\n", progname, k, j);
        }
      }
    }
    return;
  }
#undef PI
```

This code is used in section 14.

18. Routine for generation of numerical data describing a Wiener process. This is the core routine of the WIENER program. After having initialized the random number generator with the supplied seed value (calling *ranf_start*(*seed*)), the generation of a sequence of numbers describing a Wiener process starts with the generation of $M \times D$ random and uniformly distributed floating-point numbers ($M \times D/2$ pairs, assuming $M \times D$ to be an even number), from which $M \times D$ normally distributed floating-point numbers are computed using the Box–Muller transform†

The actual computation of the random numbers (at first corresponding to a uniform distribution in the interval $[0, 1]$) is done by the routine *ranf_matrix*(*aa*, *mm*, *dd*), which fills an $[M \times D]$ array.

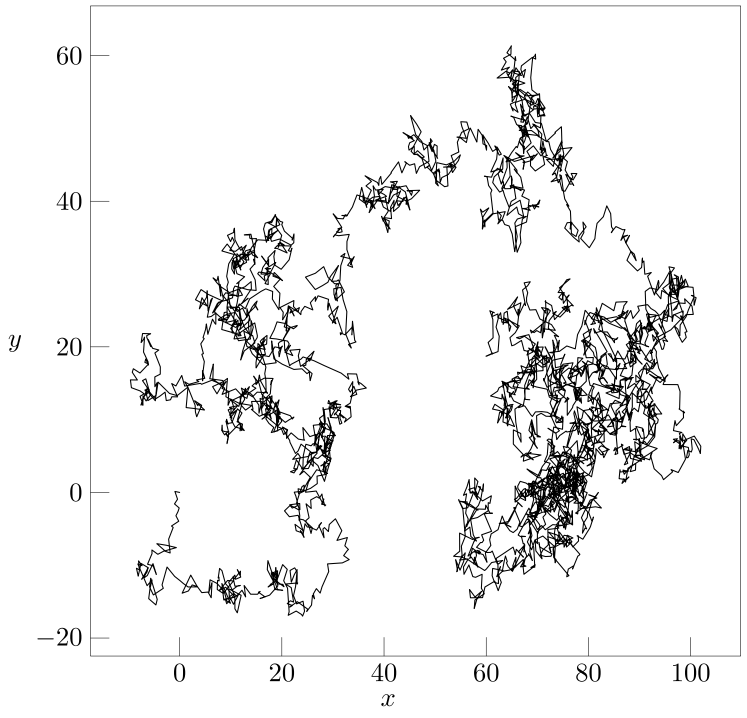


Figure 3. The same data points as in Fig. 2, but after having chained the normally distributed points to form the simulated Wiener process. The data for this graph was generated by WIENER using `wiener -D 2 -M 10000 > fig3.dat`. The trajectory starts with data point 1 at $(0, 0)$ and end up with data point 10 000 at approximately $(89.9, 12.6)$ See the `fig3.eps` block in the enclosed `Makefile` for details on how METAPOST was used in the generation of the encapsulated PostScript image of the graph.

The variables interfacing the *wiener* routine are as follows:

aa	[Output] The $M \times D$ matrix A, on return containing the M number of D-dimensional data points for the generated Wiener process.
mm	[Input] The M number of data points to generate. This equals to the number of rows in the *aa* array, and must be at least `KK` elements.
dd	[Input] The dimension of each generated data point.
seed	[Input] The seed to use when initializing the random number generator, using the routine *ranf_start*.

† For example, see `http://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform`.

mode	[Input] Determines if the sequence should be locked to simply generate a uniform distribution over the interval $[0,1]$ (`MODE_LOCKED_UNIFORM_DISTRIBUTION`) or a normal (Gaussian) distribution with expectation value zero and unit variance (`MODE_LOCKED_NORMAL_DISTRIBUTION`). Otherwise, the series of data will be generated to simulate a Wiener process, as is the main purpose of the WIENER program. One may look upon the two first modes as verification options, generating data suitable for spectral tests on the quality of the generator of pseudo-random numbers.

⟨ Routine for generation of numerical data describing a Wiener process 18 ⟩ ≡

```
void wiener(double **aa, int mm, int dd, int seed, short mode)
{
  register int j, k;
  ranf_start(seed);
  ranf_matrix(aa, mm, dd);      /* Uniform distribution over [0, 1] */
  if (mode ≡ MODE_LOCKED_UNIFORM_DISTRIBUTION) return;
  normdist(aa, mm, dd);      /* Normal distribution of unit variance around zero */
  if (mode ≡ MODE_LOCKED_NORMAL_DISTRIBUTION) return;
  for (j = 0; j < dd; j++) {
    aa[0][j] = 0.0;
    for (k = 1; k < mm; k++) {
      aa[k][j] += aa[k − 1][j];
    }
  }
}
```

This code is used in section 14.

19. Memory allocation. The *dmatrix* routine allocates an array of double floating-point precision, with array row index ranging from *nrl* to *nrh* and column index ranging from *ncl* to *nch*.

⟨ Routine for memory allocation 19 ⟩ ≡

```
double **dmatrix(long nrl, long nrh, long ncl, long nch)
{
  long i, nrow = nrh − nrl + 1, ncol = nch − ncl + 1;
  double **m;
  m = (double **) malloc((size_t)((nrow + 1) * sizeof(double *)));
  if (¬m) {
    fprintf(stderr, "%s:␣Allocation␣failure␣1␣in␣dmatrix()\n", progname);
    exit(FAILURE);
  }
  m += 1;
  m −= nrl;
  m[nrl] = (double *) malloc((size_t)((nrow * ncol + 1) * sizeof(double)));
  if (¬m[nrl]) {
    fprintf(stderr, "%s:␣Allocation␣failure␣2␣in␣dmatrix()\n", progname);
    exit(FAILURE);
  }
  m[nrl] += 1;
  m[nrl] −= ncl;
  for (i = nrl + 1; i ≤ nrh; i++) m[i] = m[i − 1] + ncol;
  return m;
}
```

This code is used in section 14.

20. Memory de-allocation. The *free_dmatrix* routine *releases* the memory occupied by the double floating-point precision matrix $v[nl\,..\,nh]$, as allocated by *dmatrix*( ).

```
⟨ Routine for memory de-allocation 20 ⟩ ≡
  void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch)
  {
    free((char *)(m[nrl] + ncl − 1));
    free((char *)(m + nrl − 1));
  }
```

This code is used in section 14.

21. Displaying a help message at terminal output.

```
⟨ Routine for displaying a help message at terminal output 21 ⟩ ≡
  void display_help_message(void)
  {
    fprintf(stderr, "Usage:␣%s␣M␣[options]\n", progname);
    fprintf(stderr, "Options:\n");
    fprintf(stderr, "␣-h,␣--help\n");
    fprintf(stderr, "␣␣␣␣Display␣this␣help␣message␣and␣exit␣clean.\n");
    fprintf(stderr, "␣-v,␣--verbose\n");
    fprintf(stderr, "␣␣␣␣Toggle␣verbose␣mode.␣Default:␣off.\n");
    fprintf(stderr, "␣-M,␣--num_samples␣<M>\n");
    fprintf(stderr, "␣␣␣␣Generate␣M␣samples␣of␣data.␣Here␣M␣should␣always␣be\n");
    fprintf(stderr, "␣␣␣␣an␣even␣number,␣greater␣than␣the␣long␣lag␣KK=%d.\n", KK);
    fprintf(stderr, "␣␣␣␣If␣an␣odd␣number␣is␣specified,␣the␣program␣will\n");
    fprintf(stderr, "␣␣␣␣automatically␣adjust␣this␣to␣the␣next␣higher\n");
    fprintf(stderr, "␣␣␣␣integer.␣Default:␣M=%d.\n", KK);
    fprintf(stderr, "␣-D,␣--dimension␣<D>\n");
    fprintf(stderr, "␣␣␣␣Generate␣D-dimensional␣samples␣of␣data.␣Default:␣D=1.\n");
    fprintf(stderr, "␣-s,␣--seed␣<seed>\n");
    fprintf(stderr, "␣␣␣␣Define␣a␣custom␣seed␣number␣for␣the␣initialization\n");
    fprintf(stderr, "␣␣␣␣of␣the␣random␣number␣generator.␣Default:␣seed=%d.\n", DEFAULT_SEED);
    fprintf(stderr, "␣-u,␣--uniform\n");
    fprintf(stderr, "␣␣␣␣Instead␣of␣generating␣a␣sequence␣of␣D-dimensional\n");
    fprintf(stderr, "␣␣␣␣corresponding␣to␣a␣Wiener␣process,␣lock␣the␣program\n");
    fprintf(stderr, "␣␣␣␣to␣simply␣generate␣a␣uniform␣distribution␣of\n");
    fprintf(stderr, "␣␣␣␣D-dimensional␣points,␣with␣each␣element␣distributed\n");
    fprintf(stderr, "␣␣␣␣over␣the␣interval␣[0,1].\n");
    fprintf(stderr, "␣-n,␣--normal\n");
    fprintf(stderr, "␣␣␣␣Instead␣of␣generating␣a␣sequence␣of␣D-dimensional\n");
    fprintf(stderr, "␣␣␣␣corresponding␣to␣a␣Wiener␣process,␣lock␣the␣program\n");
    fprintf(stderr, "␣␣␣␣to␣simply␣generate␣a␣normal␣distribution␣of\n");
    fprintf(stderr, "␣␣␣␣D-dimensional␣points,␣with␣each␣element␣distributed\n");
    fprintf(stderr, "␣␣␣␣with␣zero␣expectation␣value␣and␣unit␣variance.\n");
  }
```

This code is used in section 14.

22. Checking for a valid path character. The *pathcharacter* routine takes one character *ch* as argument, and returns 1 ("true") if the character is valid character of a path string, otherwise 0 ("false") is returned.

⟨ Routine for checking valid path characters 22 ⟩ ≡

```
  short pathcharacter(int ch)
  {
    return (isalnum(ch) ∨ (ch ≡ '.') ∨ (ch ≡ '/') ∨ (ch ≡ '\\') ∨ (ch ≡ '_') ∨ (ch ≡ '-') ∨ (ch ≡ '+'));
  }
```

This code is used in section 14.

23. Stripping path string from a file name. The *strip_away_path* routine takes a character string *filename* as argument, and returns a pointer to the same string but without any preceding path segments. This routine is, for example, useful for removing paths from program names as parsed from the command line.

⟨ Routine for stripping away path string 23 ⟩ ≡

```
  char *strip_away_path(char filename[])
  {
    int j, k = 0;
    while (pathcharacter(filename[k])) k++;
    j = (−−k);     /* this is the uppermost index of the full path+file string */
    while (isalnum((int)(filename[j]))) j−−;
    j++;     /* this is the lowermost index of the stripped file name */
    return (&filename[j]);
  }
```

This code is used in section 14.

24. Declaration of local variables of the *main* program. In CWEB one has the option of adding variables along the program, for example by locally adding temporary variables related to a given sub-block of code. However, the philosophy in the WIENER program is to keep all variables of the *main* section collected together, so as to simplify tasks as, for example, tracking down a given variable type definition. Exceptions to this rule are dummy variables merely used for the iteration over loops, not participating in any other tasks. The local variables of the program are as follows:

aa	The $M \times D$ matrix A, containing the M number of D-dimensional data points for the generated Wiener process.
mm	The M number of data points to generate. This equals to the number of rows in the *aa* array (of dimension $[M \times D]$), and must be at least `KK` elements. The default initialization is $mm =$ `KK`; however this may change depending on supplied command-line parameters.
dd	The dimension of each generated data point. Default value: 1.
seed	The seed to use when initializing the random number generator, using the routine *ranf_start*. The seed can be hand-picked using the `--seed` command-line option. Default value: 310952.
mode	Determines if the sequence should be locked to simply generate a uniform distribution over the interval $[0, 1]$ (`MODE_LOCKED_UNIFORM_DISTRIBUTION`) or a normal (Gaussian) distribution with expectation value zero and unit variance (`MODE_LOCKED_NORMAL_DISTRIBUTION`). Otherwise, the series of data will be generated to simulate a Wiener process, as is the main purpose of the WIENER program. One may look upon the two first modes as verification options, generating data suitable for spectral tests on the quality of the generator of pseudo-random numbers.

⟨ Declaration of local variables 24 ⟩ ≡

```
double **aa;
unsigned long mm = KK;
unsigned dd = 1;
int seed = DEFAULT_SEED;
short mode = MODE_WIENER_PROCESS, verbose = 0;
int no_arg;
register int j, k;
```

This code is used in section 10.

25. Parse command line for parameters. We here use the possibility open in CWEB to add `getopt.h` to the inclusions of libraries, as this block is the only one making use of the definitions therein.

```
⟨ Parse command line for parameters 25 ⟩ ≡
  progname = strip_away_path(argv[0]);
  no_arg = argc;
  while (−− argc) {
    if (cmd_match("-h", "--help", argv[no_arg − argc])) {
      display_help_message( );
      exit(SUCCESS);
    }
    else if (cmd_match("-v", "--verbose", argv[no_arg − argc])) {
      verbose = (verbose ? 0 : 1);
    }
    else if (cmd_match("-M", "--num_samples", argv[no_arg − argc])) {
      −− argc;
      if (¬sscanf(argv[no_arg − argc], "%lu", &mm)) {
        fprintf(stderr, "%s:␣Error␣detected␣when␣parsing␣the␣number␣of␣" "samples.\n", progname);
        display_help_message( );
        exit(FAILURE);
      }
      if (mm < KK) {
        fprintf(stderr,
            "%s:␣The␣M␣number␣of␣data␣points␣must␣be␣at␣least␣" "the␣long␣lag␣of␣the\
            ␣generator,␣M␣>=␣KK␣=␣%d.\n", progname, KK);
        exit(FAILURE);
      }
      if (is_odd(mm)) mm++;     /* If odd, then make it even */
    }
    else if (cmd_match("-D", "--dimension", argv[no_arg − argc])) {
      −− argc;
      if (¬sscanf(argv[no_arg − argc], "%ud", &dd)) {
        fprintf(stderr, "%s:␣Error␣detected␣when␣parsing␣dimension.\n", progname);
        display_help_message( );
        exit(FAILURE);
      }
      if (dd < 1) {
        fprintf(stderr, "%s:␣Dimension␣D␣should␣be␣at␣least␣1.\n", progname);
        exit(FAILURE);
      }
    }
    else if (cmd_match("-s", "--seed", argv[no_arg − argc])) {
      −− argc;
      if (¬sscanf(argv[no_arg − argc], "%d", &seed)) {
        fprintf(stderr, "%s:␣Error␣detected␣when␣parsing␣the␣seed␣of␣the␣" "initializer.\n",
            progname);
        display_help_message( );
        exit(FAILURE);
      }
    }
    else if (cmd_match("-u", "--uniform", argv[no_arg − argc])) {
      mode = MODE_LOCKED_UNIFORM_DISTRIBUTION;
    }
    else if (cmd_match("-n", "--normal", argv[no_arg − argc])) {
      mode = MODE_LOCKED_NORMAL_DISTRIBUTION;
    }
    else {
      fprintf(stderr, "%s:␣Sorry,␣I␣do␣not␣recognize␣option␣'%s'.\n", progname,
          argv[no_arg − argc]);
      display_help_message();
      exit(FAILURE);
    }
  }
  if (verbose) fprintf(stdout, "This␣is␣%s␣v.%s.␣%s\n", progname, VERSION, COPYRIGHT);
```

This code is used in section 10.

26. Allocate memory for a vector containing $M \times D$ elements. We here make a call to the operating system for the allocation of a sufficcient amount of memory to accommodate M data points, each of dimension D. We here apply the common convention in C, starting the indexing of the allocated array at zero.

⟨ Allocate memory for a vector containing $M \times D$ elements 26 ⟩ ≡

```
  aa = dmatrix(0, mm − 1, 0, dd − 1);
```

This code is used in section 10.

27. Fill vector with M number of D:tuples describing the Wiener process.

⟨ Fill vector with M number of D:tuples describing the Wiener process 27 ⟩ ≡

```
  wiener(aa, mm, dd, seed, mode);
```

This code is used in section 10.

28. Print the generated vector at standard terminal output.

⟨ Print the generated vector at standard terminal output 28 ⟩ ≡

```
  for (k = 0; k < mm; k++) {
    for (j = 0; j < dd − 1; j++) printf("%.20f␣", aa[k][j]);
    printf("%.20f\n", aa[k][j]);
  }
```

This code is used in section 10.

29. Deallocate the memory occupied by the vector of $M \times D$ elements.

⟨ Deallocate the memory occupied by the vector of $M \times D$ elements 29 ⟩ ≡

```
  free_dmatrix(aa, 0, mm − 1, 0, dd − 1);
```

This code is used in section 10.

30. Index.

WIENER