Appendix: Software References and Tools

By Simone Sbaraglia

Here we will include references on how to implement the models and some examples.

The precision of Monte Carlo simulation results mainly depends on the number of simulations performed.

Thus, the implementation of high-level software is not the best solution, as it results in a much slower execution, and therefore in wider intervals of confidence in its results. Instead, programming in a low-level language allows for faster runs and thus the ability to perform a higher number of iterations, which yields higher-quality results.

In the following, we present a C language implementation of the models, together with the description needed to execute and possibly modify and improve the software.

Let us describe here a sample implementation of the algorithms outlined in the book. The high-level structure of the source code is an implementation of a Monte Carlo algorithm, whose simulation accuracy highly depends on the number of iterations performed. Therefore, it would be inefficient to write the code in a high-level interpreted programming language. The samples are therefore provided in the C programming language.

The structure of the source code is as follows:

main function {

1) interpret command line args
2) read input files

Banking Systems Simulation: Theory, Practice, and Application of Modeling Shocks, Losses, and Contagion, First Edition. Stefano Zedda.

3) initialize data structures
4) allocate data structures
5) compute correlation matrix
6) Monte Carlo external loop, proceeds until the number of defaults equals nTotDefaults. At each iteration of the loop:
   6.1 generate random bank losses
   6.2 simulate interbank contagion
7) at the end of Monte Carlo loop dump all output files

}

1) **Interpret command line args**

The command line options provided are the following:

-nointerbank	do not simulate interbank contagion
-dumpbankloss	output the matrix of nbanks columns by ntotdefaults rows containing the excess losses for each bank in each simulation with at least one default (full mapping)
-dumpecontrib	output a line vector with the risk contribution for each bank above of the "large loss limit" set
-dumpecloss	output a line vector with the excess losses for each simulation
-randomseq	the random numbers generation is different in every set of simulations

Furthermore, the software expects one input file with the following contents: assets PD, deposits, capital, total assets, interbank debts, interbank credits, and correlation.

This version of the software includes the distinction of the losses above a preset threshold.

When launching the execution, the command line must contain the input file name and the "large loss limit" value. Launching the simulation process without reference to any threshold (so all losses) is as simple as setting the "large loss limit" value to zero.

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

```
#include <unistd.h>
#include <math.h>
#include <string.h>

#include <gsl_randist.h>
#include <gsl_cdf.h>
#include "common.h"

#include <gsl_errno.h>
#include <gsl_matrix.h>
#include <gsl_rng.h>

#define NARGS 2
#define MAXSTRLEN 256
#define NCOLS 7

void Usage(char *s) {
   fprintf(stderr, "Usage: %s [-nointerbank | -
dumpbankloss | -dumpecontrib |-dumpecloss | -
randomseq] inputfile LargeLossLimit\n", s);
   exit(-1);
}

double R(double x) {
   return(0.12*(1-exp(-50*x))/(1-exp(-50))+0.24*
(1-(1-exp(-50*x))/(1-exp(-50))));
}

double ba(double x) {
   return(pow(0.11852-0.05478*log(x), 2));
}

double regcap(double x, double y, double k) {
   double a;

   a = (k*gsl_cdf_ugaussian_P(
              pow(1-R(x), -0.5)*gsl_cdf_ugaussian_Pinv(x)+
              pow(R(x)/(1-R(x)), 0.5)*y) - x*k)*
        pow(1-1.5*ba(x), -1)*1.06;
   return(a);
}
```

```
void printDoubleArray(char *s, int n, double *p) {
   int i;

   if (s!=NULL) {
      printf("-->ARRAY %s\n", s);
   }
   for (i=0; i<n; i++) {
      printf("%f\n", p[i]);
   }
}

int main(int argc, char *argv[]) {
   double rho=0;
   double LGD=0.45;
   double BankLossCap = 0;
   char *filename = NULL;
   char *outfnameprefix1 = "Bankloss";
   char *outfnameprefix2 = "DIScharged";
   char *outfnameprefix3 = "Econtrib";
   char *outfnameprefix4 = "EcLoss";
   char *outfname1;
   char *outfname2;
   char *cont = "n";
   FILE *fp = NULL;
   FILE *ofp1 = NULL;
   FILE *ofp2 = NULL;
   FILE *ofp3 = NULL;
   FILE *ofp4 = NULL;
   char buffer[MAXSTRLEN];
   double **data = NULL;
   double **BankLoss = NULL;
   double *RContrib = NULL;
   double *EcLoss = NULL;
   double *DIScharged = NULL;
   double *ECcharged = NULL;
   double *market = NULL;
   double *corr = NULL;
   int *BankDefaulted = NULL;
   int *BankDefaultedOld = NULL;
   double InterbankUnitaryLoss = 0;
   double Baseloss = 0;
   int i, b, j, nDefaults;
```

```
int nbanks = -1;
int niter = -1;
int hadDefault;
int hadDefaultInterbank = 0;
int numDefaultedBanks = 0;
int allDefaulted;
double InterBankLoss, InterBankDefault;
int argnum=1;
int intsize1 = 0;
int intsize2 = 0;
int intsize3 = 0;
int intsize4 = 0;
int interbanksimulation = 1;
int dumpbankloss = 0;
int dumpecontrib = 0;
int dumpecloss = 0;
int nTotDefaults = 100000;
int LargeLossLimit = 0;
double tmp=0;
const gsl_rng_type * Ra;
gsl_rng * r;
gsl_rng_env_setup();
Ra = gsl_rng_default;
r = gsl_rng_alloc (Ra);

/* checking number of args */
if (argc < NARGS + 1) {
   Usage(argv[0]);
   exit(-1);
}

while (argnum < argc-NARGS) {
  if (!strcmp(argv[argnum], "-nointerbank")) {
     argnum++;
     interbanksimulation = 0;
     fprintf(stderr, "Interbank Simulation Disabled\n");
  } else if (!strcmp(argv[argnum], "-dumpbankloss")) {
     argnum++;
     dumpbankloss = 1;
     fprintf(stderr, "Dumping BankLoss Enabled\n");
  } else if (!strcmp(argv[argnum], "-dumpecloss")) {
```

```
        argnum++;
        dumpecloss = 1;
        fprintf(stderr, "Dumping Economic Losses
Enabled\n");
     } else if (!strcmp(argv[argnum], "-dumpecontrib")) {
        argnum++;
        dumpecontrib = 1;
        fprintf(stderr, "Dumping Economic Risk
Contributions Enabled\n");
     } else if (!strcmp(argv[argnum], "-randomseq")) {
        argnum++;
        srandom(time(NULL));
        fprintf(stderr, "Init random seed\n");
     } else {
        Usage(argv[0]);
     }
  }

   /* reading command line */
   filename = argv[argnum];
   if ((fp = fopen(filename, "r")) == NULL) {
      fprintf(stderr, "could not open file %s for reading
\n", filename);
      exit(-1);
   }

   LargeLossLimit = atof(argv[argnum+1]);
   if ((LargeLossLimit < 0)) {
      fprintf(stderr, "LargeLossLimit cannot be
negative: %s\n", LargeLossLimit);
      exit(-1);
   }

   intsize1 = 10;
   intsize2 = floor(log10(nTotDefaults)) + 1;
   intsize3 = 15;

   outfnameprefix2 = filename;
```

2) Read input files

The input files specified on the command line are read in this phase. The software automatically computes the number of banks based on the dimension of the data in the first input file and allocates the corresponding data structures accordingly.

```
/* count number of banks (nrows in data matrix) */
nbanks = 0;
while (fgets(buffer, MAXSTRLEN, fp)) {
   nbanks++;
}
rewind(fp);

gsl_matrix * m = gsl_matrix_alloc (1, nbanks);
gsl_matrix * c = gsl_matrix_alloc (nbanks, nbanks);
gsl_matrix * out = gsl_matrix_alloc (1, nbanks);

/* allocate data matrix */
if ((data = (double **)malloc(nbanks*sizeof
(double *))) == NULL) {
   fprintf(stderr, "could not allocate data\n");
   exit(-1);
}
for (i=0; i<nbanks; i++) {
   if ((data[i] = (double *)malloc(NCOLS*sizeof
(double))) == NULL) {
   fprintf(stderr, "could not allocate data[%d]\n", i);
   exit(-1);
  }
  memset(data[i], 0, NCOLS*sizeof(double));
}
```

3) **Initialize data structures**

Data structures allocated in the previous step need to be initialized. Namely, the "data" matrix is initialized based on the data read from Inputfile:

```
/* reading input file and initializing data matrix */
  for (i=0; i<nbanks; i++) {
```

```
      if (fgets(buffer, MAXSTRLEN, fp) == NULL) {
         fprintf(stderr, "could not read line %d
in input file\n", i);
         exit(-1);
      }
      if (sscanf(buffer, "%lf %lf %lf %lf %lf %lf %lf
\n", &data[i][0], &data[i][1], &data[i][2], &data[i]
[3], &data[i][4], &data[i][5],  &data[i][6]) < NCOLS) {
 fprintf(stderr, "could not read the %d tokens expected
\n", NCOLS);
 exit(-1);
      }
   }
   fclose(fp);
```

Here the "data" matrix has nbanks rows and NCOLS (fixed to 7) columns and holds the values coming from the "input file."

4) **Allocate data structures**

Once the number of banks (nbanks) involved in the simulation is known, all relevant data structures that are required in the computation and to hold output values can be allocated and initialized to zero:

```
   /* allocating matrix BankLoss */
   if ((BankLoss = (double **)malloc
(nTotDefaults*sizeof(double *))) == NULL) {
      fprintf(stderr, "could not allocate BankLoss\n");
      exit(-1);
   }
   for (i=0; i<nTotDefaults; i++) {
      if ((BankLoss[i] = (double *)malloc
(nbanks*sizeof(double))) == NULL) {
       fprintf(stderr, "could not allocate BankLoss
[%d]\n", i);
       exit(-1);
      }
      memset(BankLoss[i], 0, nbanks*sizeof(double));
   }

  /* allocating RContrib */
```

```
   if ((RContrib = (double *)malloc(nbanks*sizeof
(double))) == NULL) {
      fprintf(stderr, "could not allocate RContrib\n");
      exit(-1);
   }
   memset(RContrib, 0, nbanks*sizeof(double));

  /* allocating EcLoss */
   if ((EcLoss = (double *)malloc(nTotDefaults*sizeof
(double))) == NULL) {
      fprintf(stderr, "could not allocate EcLoss\n");
      exit(-1);
   }
   memset(EcLoss, 0, nTotDefaults*sizeof(double));

  /* allocating DIScharged */
   if ((DIScharged = (double *)malloc
(nTotDefaults*sizeof(double))) == NULL) {
      fprintf(stderr, "could not allocate DIScharged\n");
      exit(-1);
   }
   memset(DIScharged, 0, nTotDefaults*sizeof(double));

  /* allocating ECcharged */
   if ((ECcharged = (double *)malloc
(nTotDefaults*sizeof(double))) == NULL) {
      fprintf(stderr, "could not allocate ECcharged\n");
      exit(-1);
   }
   memset(ECcharged, 0, nTotDefaults*sizeof(double));

   /* allocating market */
   if ((market = (double *)malloc(nbanks*sizeof
(double))) == NULL) {
      fprintf(stderr, "could not allocate market\n");
      exit(-1);
   }
   memset(market, 0, nbanks*sizeof(double));

   /* allocating corr */
   if ((corr = (double *)malloc(nbanks*sizeof
(double))) == NULL) {
```

```
      fprintf(stderr, "could not allocate corr\n");
      exit(-1);
   }
   memset(corr, 0, nbanks*sizeof(double));

   /* allocating BankDefaulted */
   if ((BankDefaulted = (int *)malloc(nbanks*sizeof
(int))) == NULL) {
      fprintf(stderr, "could not allocate BankDefaulted\n");
      exit(-1);
   }
   memset(BankDefaulted, 0, nbanks*sizeof(int));

   /* allocating BankDefaultedOld */
   if ((BankDefaultedOld = (int *)malloc(nbanks*sizeof
(int))) == NULL) {
      fprintf(stderr, "could not allocate BankDefaulted
Old\n");
      exit(-1);
   }
   memset(BankDefaultedOld, 0, nbanks*sizeof(int));
```

The meaning of the above data structures in the simulation code is as follows:

BankLoss
: Each row (iteration) holds the total loss for each bank. This matrix is nTotDefaults X nbanks in dimension.

RContrib
: Records the risk contribution for each bank above the preset "large loss limit" threshold. It is an array of dimension nbanks.

EcLoss
: Records the value of excess losses result of banks defaulted in the simulation. It is an array of dimension nbanks.

DISCharged
: Records the value of deposits to be covered as a result of banks defaulted in the simulation. It is an array of dimension nbanks.

ECcharged
: Records the excess losses of banks defaulted in the simulation. It is an array of dimension nbanks.

BankDefaulted
: Records which bank defaults in the simulation. It is an array of dimension nbanks.

5) **Compute correlation matrix**

During this step, the "corr" correlation matrix is initialized and afterward we compute its Cholesky decomposition, using the linear algebra GPL-licensed software GSL.

```
/* compute correlation matrix and its Cholesky decomp */
       for (i = 0; i < nbanks; i++){
                 corr[i]=pow((rho+data[i][6]), 0.5);
                 }
       for (i = 0; i < nbanks; i++){
                 for (j = 0; j < nbanks; j++){
                gsl_matrix_set (c, i, j, corr[i]*corr[j]);
                 if (i==j){gsl_matrix_set (c, i, j, 1);}
                 }
       }
gsl_linalg_cholesky_decomp(c);

     for (i = 0; i < nbanks; i++){
              for (j = 0; j < nbanks; j++){
                   if (i<j){gsl_matrix_set (c, i, j, 0);}
                   }
              }
```

6) **Monte Carlo loop**

The main computation phase starts with an outer Monte Carlo simulation, which is carried out until a certain number of scenarios have been computed (100.000 in the default value). After each iteration, if at least one default has occurred, the variable nDefaults is increased, and the iteration values are recorded. The Monte Carlo simulation therefore proceeds until nDefaults equal nTotDefaults (100.000). This means that the total number of iterations is not known a priori, as it depends on a bank's stability: the higher the

stability, the higher the number of iterations due for completing the simulations set. When no default has occurred, no values are recorded.

```
/* loop until number of defaults equals nTotDefaults */
nDefaults = 0;
niter = -1;
while (nDefaults < nTotDefaults) {
   niter++;

   hadDefault = 0;
   memset(BankDefaulted, 0, nbanks*sizeof(int));
```

BankDefaulted is an array of dimension nbanks that will record a 1 for the banks defaulted in the current iteration, and therefore is initialized to zero.

6.1 **Generate random bank losses**

A random number is generated for each bank:

```
for (b=0; b<nbanks; b++) {
gsl_matrix_set (m, 0, b, gsl_ran_gaussian(r,1));
}
```

Random values are then correlated among them:

```
for (j = 0; j < nbanks; j++){
         tmp=0;
         for (i = 0; i < nbanks; i++){
                  tmp+=gsl_matrix_get(m, 0, i)
*gsl_matrix_get(c, j, i);
                  }
         gsl_matrix_set (out, 0, j, tmp);
         }
```

The random correlated value is transformed into simulated unitary loss by means of the FIRB formula:

```
for (i=0; i<nbanks; i++) {
       market[i]= gsl_matrix_get (out, 0, i);
         }
```

```
for (b=0; b<nbanks; b++) {
 Baseloss = regcap(data[b][0], market[b], LGD);
```

And the random correlated value is then multiplied by the bank dimension (total assets):

```
BankLoss[nDefaults][b] = data[b][3]*Baseloss;
```

And the simulated loss is compared to capital for verifying if the bank has defaulted:

```
/* Check for defaults  */

if (BankLoss[nDefaults][b] > data[b][2]) {
             BankDefaulted[b] = 1;
             hadDefault = 1;
        }
      }
        for (j=0; j<nbanks; j++) {
              BankDefaultedOld[j]=BankDefaulted[j];
              }
```

6.2 **Interbank simulation**

After checking each of the banks independent of each other, it is of paramount importance to properly account for interbank lending and possible contagion. Unless the step is skipped by the user by means of the command-line option, the simulation is performed here. An inner loop propagates the loss to all other banks and checks if some other bank defaults as a result of contagion. The loss is then propagated to the system and the whole iteration continues until the system stabilizes (there are no more contagion defaults) or all banks default.

```
if (interbanksimulation) {
 cont = "c";
 allDefaulted = 1;
 for (j=0; j<nbanks; j++) {
    if (BankDefaultedOld[j] == 0) {
       allDefaulted = 0;
```

```
        break;
     }
  }

  hadDefaultInterbank = hadDefault;
  while ((allDefaulted==0) && (hadDefaultInterbank)) {
     hadDefaultInterbank = 0;
     InterBankDefault = 0;
     InterBankLoss = 0;
     for (j=0; j<nbanks; j++) {
        if (BankDefaulted[j]) {InterBankDefault += data
[j][4];}
InterBankLoss += data[j][5];
     }

     InterbankUnitaryLoss = (InterBankDefault/
InterBankLoss);
     if ((InterBankDefault/InterBankLoss)>1)
{InterbankUnitaryLoss = 1;}

for (j=0; j<nbanks; j++) {
      BankDefaulted[j] = 0;
      BankLoss[nDefaults][j] += (InterbankUnitary
Loss*data[j][5]);
       if ((BankLoss[nDefaults][j] > data[j][2]) &&
(BankDefaultedOld[j] == 0)) {
                BankDefaulted[j] = 1;
                hadDefaultInterbank = 1;
                BankDefaultedOld[j] = 1;
                }
        }
     allDefaulted = 1;
     for (j=0; j<nbanks; j++) {
        if (BankDefaultedOld[j] == 0) {
                allDefaulted = 0;
                break;
        }
        }
    }
  }
  /* end of interbank simulation */
```

At the end of the interbank contagion simulation, all data structures are updated for the next iteration of the outer Monte Carlo loop:

```
   /* update data structures before next iteration */

  for (j=0; j<nbanks; j++) {
     if (BankDefaultedOld[j]) {
            numDefaultedBanks++;
            DIScharged[nDefaults] += data[j][1];
            ECcharged[nDefaults] += (BankLoss
[nDefaults][j] - data[j][2]);
     }
  }

  if (hadDefault) {
     nDefaults++;
  }

  /* printf("Iterations=%d, nDefaults=%d/%d,
nIBDefaults=%d\n", niter+1, nDefaults,
numDefaultedBanks); */
    }

intsize4 = floor(log10(numDefaultedBanks+1)) + 1;
```

7) **Dump all output files**

Once the Monte Carlo simulation is over (the number of defaults equals nTotDefaults), the loop ends and all output values are dumped to disk.

The output values that are generated are as follows:

```
dump BankLoss:   nbanks
dump EContrib:   nbanks   > LargeLossLimit
dump EcLoss       nDefaults nbanks   > LargeLossLimit
dump DIScharged  nDefaults

   /* dump BankLoss */
   if (dumpbankloss) {
      outfname1 = malloc(strlen(outfnameprefix1)
 + strlen("_") + strlen(cont) + strlen("_") + strlen
```

```
(outfnameprefix2) + intsize1 + strlen("_") + intsize2 +
strlen("-")+ intsize4 + strlen(".dat") + 1);
      sprintf(outfname1, "%s_%s_%s_%d-%d-%d.dat",
outfnameprefix1, outfnameprefix2, cont, LargeLossLimit,
numDefaultedBanks, niter+1);
      if ((ofp1 = fopen(outfname1, "w")) == NULL) {
          fprintf(stderr, "could not open %s for
writing\n", outfname1);
           exit(-1);
      }
      for (i=0; i<nTotDefaults; i++) {
         fprintf(ofp1, "%1.3lf", BankLoss[i][0]);
         for (b=1; b<nbanks; b++) {
             fprintf(ofp1, " %1.3lf", BankLoss[i][b]);
         }
         fprintf(ofp1, "\n");
      }
      fclose(ofp1);
   }

   /* dump EContrib */
   if (dumpecontrib) {
      outfname1 = malloc(strlen(outfnameprefix3)
 + strlen("_") + strlen(outfnameprefix2) + strlen("_")
 + strlen(cont) + strlen("_") + intsize1 + strlen
("_") + intsize2 + strlen("-") + intsize4 + strlen
(".dat") + 1);
      sprintf(outfname1, "%s_%s_%s_%d-%d-%d.dat",
 outfnameprefix3, outfnameprefix2, cont, LargeLossLimit,
 numDefaultedBanks, niter+1);
      if ((ofp3 = fopen(outfname1, "w")) == NULL) {
 fprintf(stderr, "could not open %s for writing\n",
outfname1);
 exit(-1);
      }
      for (i=0; i<nDefaults; i++) {
                if (ECcharged[i] > LargeLossLimit) {
      for (b=0; b<nbanks; b++) {
           if (BankLoss[i][b]>data[b][2]) {
      RContrib[b]+= ((((ECcharged[i] -
 LargeLossLimit)/ ECcharged[i])*(BankLoss[i][b]-data
[b][2])) / (niter+1));
```

```
                                }

                            }

                    }
        }

      for (i=0; i<nbanks; i++) {
       fprintf(ofp3, "%f\n", RContrib[i]);
    }
    fclose(ofp3);
    }

    /* dump EcLoss */
    if (dumpecloss) {
       outfname1 = malloc(strlen(outfnameprefix4)
 + strlen("_") + strlen(outfnameprefix2) + strlen("_")
 + strlen(cont) + strlen("_") + intsize1 + strlen
("_") + intsize2 + strlen("-")+ intsize4 + strlen(".
dat") + 1);
       sprintf(outfname1, "%s_%s_%s_%d-%d-%d.dat",
 outfnameprefix4, outfnameprefix2, cont, LargeLossLimit,
 numDefaultedBanks, niter+1);
       if ((ofp4 = fopen(outfname1, "w")) == NULL) {
    fprintf(stderr, "could not open %s for writing
\n", outfname1);
       exit(-1);
       }
       for (i=0; i<nDefaults; i++) {
              if (ECcharged[i] > LargeLossLimit) {
                     for (b=0; b<nbanks; b++) {
                           BankLossCap=((BankLoss[i][b]-
 data[b][2]) < (data[b][1] + data[b][4]) ? (BankLoss
[i][b] - data[b][2]):(data[b][1] + data[b][4]));
if (BankLoss[i][b]>data[b][2]){EcLoss[i]+=
 BankLossCap;}
                            }

                     }
             }

      for (i=0; i<nDefaults; i++) {
```

```
      fprintf(ofp4, "%f\n", EcLoss[i]);
   }

   fclose(ofp4);
   }

   /* dump DIScharged */
   outfname2 = malloc(strlen(outfnameprefix2) + strlen
("_") + strlen(cont) + strlen("_")
 + intsize1 + strlen("_") + intsize2 + strlen("-")
 + intsize4 + strlen(".dat") + 1);
   sprintf(outfname2, "%s_%s_%d-%d-%d.dat",
 outfnameprefix2, cont, LargeLossLimit,
 numDefaultedBanks, niter+1);
   if ((ofp2 = fopen(outfname2, "w")) == NULL) {
      fprintf(stderr, "could not open %s for writing
\n", outfname2);
      exit(-1);
   }

   for (i=0; i<nTotDefaults; i++) {
      fprintf(ofp2, "%f\n", DIScharged[i]);
   }
   fclose(ofp2);

   return 0;
}
```