1. main.c

```
#include "header.h"
#include "rng.c"
#include "tools.c"

double region, regionH, cellWid;
double bendStiff, deltaT, p_nucle, maxBrownianForce, rCut, angLMMM;
double *r, *force, *totDisp, *Brown, *rPre;
int stepCount, monomers, cells, atomsPerEdge, *cellCubic;
int *NL, NLCur, *chains, cntFila, cntElong;

int main (void)
{
  Initialize();
  Polymerize();
  CloseJob ();
  return 0;
}

void Polymerize(void) {
  int n;
  double *ip;
  printf("Polymerization is begun!\n");
  while(monomers > 0) {
          stepCount ++;
          // set all force to be zero
          ip = force;
          while (ip < force + nAtom * NDIM) { *ip = 0.;   ip++; }
          // calculate forces
          MeasurePreviousDisplacement();
          ComputeRepulsiveForces();
          ComputeSpringForces();
          ComputeBendingForces();
          ComputeBrownianForces();
          // integrate positions according to calculated force
          for (n = 0; n < nAtom * NDIM; n++) {
                   r[n] += (force[n] + Brown[n]) * deltaT;
                   totDisp[n] += fabs(force[n] + Brown[n]) * deltaT;
          }
          // apply periodic boundary condition
          ApplyBoundaryCond();
          // record data
          if (stepCount % 100000 == 0) {
                   RecordProgress();
                   RecordEnergy();
                   RecordVMD();
          }
  }
  printf("Polymerization is finished!\n");
}

// Close a job
void CloseJob (void)
{
  RecordVMD();
  RecordFilamentLength();
}

// Initialize all
void Initialize(void) {
  int n, nX, nY, nZ, ind, seed;
  double rr;
  char fileList[4][20];
  struct timeval tval;
```

```
// initialize seed number to generate random number
gettimeofday(&tval, NULL);
srand(tval.tv_usec / 1000);
seed = (int)rand();
init_genrand ( seed ); //

// scaled time step
deltaT = 0.00003;
// nucleation probability
p_nucle = 1e-8;

// cutoff radius for soft-sphere potential (min of L-J curve)
rCut = pow (2., 1./6.);
maxBrownianForce = sqrt(2.0 / deltaT);

// the number of atoms per one edge // 0.5 is added for rounding to int
atomsPerEdge = (int) (pow ( (double)nAtom, 1. / 3.) + 0.5);
// width of the simulation box
region = (double) GAP * atomsPerEdge;
regionH = 0.5 * region; // The half length of region
// the number of cells per width
cells = (int)(region / (1. + LJ_MAR) * 1.2);
// The width of one cell which is equal to gap here
cellWid = region / (double)cells;
// Cubics where monomers will be placed for calculating the LJ potential
cellCubic = (int *)malloc( sizeof(int) * ( nAtom + cells * cells * cells) );
angLMMM = cos(0. + MMM_ANGDEV * PI / 180.);

NL = (int *)malloc(sizeof(int) * 2 * nAtom * 10);
r = (double *)malloc(sizeof(double) * NDIM * nAtom);
force = (double *)malloc(sizeof(double) * NDIM * nAtom);
totDisp = (double *)malloc(sizeof(double) * NDIM * nAtom);
Brown = (double *)malloc(sizeof(double) * NDIM * nAtom);
rPre = (double *)malloc(sizeof(double) * NDIM * nAtom);
chains = (int *)malloc(sizeof(int) * nAtom * 2);
memset(chains, -1, sizeof(int) * nAtom * 2);

// Distribute monomers in a cubic lattice
for (nZ = 0; nZ < atomsPerEdge; nZ ++) {
  for (nY = 0; nY <  atomsPerEdge; nY ++) {
    for (nX = 0; nX <  atomsPerEdge; nX ++) {
                // Atom currently being placed
                ind = (nZ * atomsPerEdge + nY) * atomsPerEdge + nX;
                P2(r,ind,0) = ((double)nX + 0.5) * GAP;
                P2(r,ind,1) = ((double)nY + 0.5) * GAP;
                P2(r,ind,2) = ((double)nZ + 0.5) * GAP;
         }
       }
}
// The initial value of the number of unbinded monomers
monomers = nAtom;
// The intialization of stepcount
stepCount = 0;
cntFila = 0;
cntElong = 0;

// Total displacements and forces are set to zero
for(n = 0; n < nAtom * NDIM; n++) {
        totDisp[n] = 0.;
}
UpdateNeighborList();
UpdateRpre();

strcpy(fileList[0], "energy.dat");
strcpy(fileList[1], "progress.dat");
strcpy(fileList[2], "vmd.pdb");
```

```
  strcpy(fileList[3], "vmd.psf");
  for(n = 0; n < 4; n++) {
          if (access(fileList[n], 0) == 0) {
                  remove(fileList[n]);
          }
  }
}

void UpdateNeighborList(void) {
  int j1, j2, m1C[NDIM], m2C[NDIM], offset, c, m1, m2, n, k, CS, CS2;
  int iofX[] = {0,1,1,0,-1,0,1,1,0,-1,-1,-1, 0, 1};
  int iofY[] = {0,0,1,1, 1,0,0,1,1, 1, 0,-1,-1,-1};
  int iofZ[] = {0,0,0,0, 0,1,1,1,1, 1, 1, 1 ,1, 1};
  double dr[NDIM], shift[NDIM], rr;

  NLCur = 0;
  // initialize the cellCubic matrix to all -1
  memset(cellCubic, -1, (nAtom + cells * cells * cells) * sizeof(int));
  // Assigns particles to cells
  for (n = 0; n < nAtom; n++) {
          c = ((int) (P2(r,n,2) / cellWid) * cells + (int) (P2(r,n,1) / cellWid))
                  * cells + (int) (P2(r,n,0) / cellWid) + nAtom;
          cellCubic[n] = cellCubic[c];
          cellCubic[c] = n;
  }

  for (n = 0; n < cells * cells * cells; n++) {
          m1C[0] = n % cells;
          m1C[1] = (n / cells) % cells;
          m1C[2] = (n / cells) / cells;

          m1 = n + nAtom;
          j1 = cellCubic[m1];
          while (j1 > -1) {
                  for (offset = 0; offset < 14; offset++) {
                          CS = 1; j2 = -1;
                          m2C[0] = m1C[0] + iofX[offset];
                          m2C[1] = m1C[1] + iofY[offset];
                          m2C[2] = m1C[2] + iofZ[offset];

                          for (k = 0; k < NDIM; k++) { shift[k] = 0.; }

                          for (k = 0; k < NDIM; k++) {
                                  if (m2C[k] >= cells) {
                                          m2C[k] = 0;
                                          shift[k] = region;
                                  }
                                  else if (m2C[k] < 0) {
                                          m2C[k] = cells - 1;
                                          shift[k] = -region;
                                  }
                          }
                          if (m2C[0] >= cells || m2C[0] < 0 || m2C[1] >= cells
                                   || m2C[1] < 0 || m2C[2] >= cells || m2C[2] < 0 ) {
                                  CS = 0;
                          }
                          if (CS == 1) {
                                  m2 = ( m2C[2] * cells + m2C[1] ) * cells + m2C[0]
                                                  + nAtom;
                                  j2 = cellCubic[m2];
                          }

                          while (j2 > -1 & CS == 1) {
                                  CS2 = 1;
                                  if (!(m1 != m2 || (m1 == m2 && j2 < j1))) { CS2 = -1; }

                                  if (P2A(chains,j1,0,2) == j2 || P2A(chains,j1,1,2) == j2)
```

```
                                        { CS2 = -1; }

                                        if (((P2A(chains,j1,0,2) == P2A(chains,j2,1,2))
                                                        && P2A(chains,j1,0,2) > -1)  || ((P2A(chains,j1,1,2)
                                                        == P2A(chains,j2,0,2)) && P2A(chains,j1,1,2) > -1))
                                        { CS2 = -1; }

                                        if (CS2 == 1) {
                                                for (k = 0; k < NDIM; k++) {
                                                        dr[k] = P2(r,j1,k) - P2(r,j2,k) - shift[k];
                                                }
                                                AppBound(dr);
                                                rr = DotProd (dr, dr);
                                                if ( rr < (1. + LJ_MAR) * (1. + LJ_MAR) ) {
                                                        P2A(NL,NLCur,0,2) = j1;
                                                        P2A(NL,NLCur,1,2) = j2;
                                                        NLCur++;
                                                }
                                        }
                                        j2 = cellCubic[j2];
                                }       // WHILE-J2
                        }       // OFFSET
                        j1 = cellCubic[j1];
                }       // WHILE-J1
        }
}
void ComputeRepulsiveForces (void)
{
  int n, k, nln[2];
  double dr[NDIM], f, fcVal, rr, rr_r, rr_r3;

  for (n = 0; n < NLCur; n++) {
          for (k = 0; k < 2; k++) {
                  nln[k] = P2A(NL,n,k,2);
          }
          for (k = 0; k < NDIM; k++) {
                  dr[k] = P2(r,nln[0],k) - P2(r,nln[1],k);
          }
          AppBound(dr);
          rr = sqrt(DotProd(dr, dr));

          if ( rr < 1.0 ) {
                  rr_r = rr - 1. + rCut;
                  rr_r3 = rr_r * rr_r * rr_r;
                  fcVal = 48. * LAMBDA * rr_r3 * (rr_r3 - 0.5) * rr_r / rr;
                  for (k = 0; k < NDIM; k++) {
                          f = fcVal * dr[k];
                          P2(force,nln[0],k) += f;
                          P2(force,nln[1],k) -= f;
                  }
          }
          UpdateChains(nln[0], nln[1], rr);
  }
}

void MeasurePreviousDisplacement(void) {
  double max, rr, dr[NDIM];
  int n, k;

  max = -1.;
  for(n = 0; n < nAtom; n++) {
          for(k = 0; k < NDIM; k++) {
                  dr[k] = P2(r,n,k) - P2(rPre,n,k);
          }
          AppBound(dr);
          rr = DotProd (dr, dr);
          if (rr > max) { max = rr; }
```

```
  }
  if (max > ( (LJ_MAR - BUFFER_LJ_MAR) * (LJ_MAR - BUFFER_LJ_MAR) * 0.25) ) {
          UpdateNeighborList();
          UpdateRpre();
  }
}
void UpdateRpre(void) {
  int n;
  for (n = 0; n < nAtom * NDIM; n++) {
          rPre[n] = r[n];
  }
}

void UpdateChains (int j1, int j2, double rr)
{
  double rdm, cosAngle, dr1[NDIM], dr2[NDIM];
  int k, nCh1 = 0, nCh2 = 0, *a, *b, ind;

  if (rr >= MIN_DIST_BOND_FILA && rr <= MAX_DIST_BOND_FILA) {
          for (k = 0 ; k < 2 ; k++) {
                  if ( P2A(chains,j1,k,2) < 0 ) { nCh1++; }
                  if ( P2A(chains,j2,k,2) < 0 ) { nCh2++; }
          }

          if (nCh1 == 2 && nCh2 == 2) {
                  rdm = genrand_real3();
                  if (rdm <= p_nucle) {
                          if ( genrand_real3 () > 0.5 ) {
                                  P2A(chains,j1,1,2) = j2;
                                  P2A(chains,j2,0,2) = j1;
                          }
                          else {
                                  P2A(chains,j1,0,2) = j2;
                                  P2A(chains,j2,1,2) = j1;
                          }
                          monomers -= 2;
                          cntFila++;
                  }
          }
          else if ((nCh1 == 1 && nCh2 == 2 && P2A(chains,j1,0,2) < 0) ||
                          (nCh1 == 2 && nCh2 == 1 && P2A(chains,j2,0,2) < 0)) {
                  if (nCh1 < nCh2) { a = &j1;  b = &j2; }
                  else { b = &j1;    a = &j2; }

                  ind = P2A(chains,*a,1,2); // Monomer next to the fiber tip
                  for (k = 0; k < NDIM; k++) {
                          dr1[k] = P2(r,*a,k) - P2(r,ind,k);
                          dr2[k] = P2(r,*b,k) - P2(r,*a,k);
                  }
                  AppBound(dr1);
                  AppBound(dr2);
                  cosAngle = fabs(DotProd(dr1, dr2)
                                  / sqrt(DotProd(dr1, dr1) * DotProd(dr2, dr2)));
                  if (cosAngle > angLMMM) {
                          P2A(chains,*b,1,2) = *a;
                          P2A(chains,*a,0,2) = *b;
                          monomers--;
                          cntElong++;
                  }
          }
  }
}

void ComputeSpringForces (void)
{
  double dr[NDIM], f, fcVal, rr;
  int m, n, k, ind;
```

```
  for (n = 0; n < nAtom; n++) {
          for(m = 0; m < 2 ; m++) {
                  ind = P2A(chains,n,m,2);
                  if (ind > -1) {
                          for (k = 0; k < NDIM; k++) {
                                  dr[k] = P2(r,n,k) - P2(r,ind,k);
                          }
                          AppBound(dr);
                          rr = sqrt(DotProd(dr, dr));
                          fcVal = -1. * SPR_STIFF_FILA * (rr - 1.);
                          fcVal /= rr;
                          for (k = 0; k < NDIM; k++) {
                          f = fcVal * dr[k];
                                  P2(force,n,k) += f;
                          }
                  }
          }
  }
}

void ComputeBendingForces (void)
{
  double dr1[NDIM], dr2[NDIM], c, f, f1, f2;
  double c11, c22, c12, cD, fe1[NDIM], fe2[NDIM], mag1, mag2;
  int k, n, n1, n2;

  // Bending forces for straightening actin filaments
  for (n = 0; n < nAtom; n++) {
          if (P2A(chains,n,0,2) > -1 && P2A(chains,n,1,2) > -1) {
                  n1 = P2A(chains,n,0,2);
                  n2 = P2A(chains,n,1,2);
                  for (k = 0; k < NDIM; k ++) {
                          dr1[k] = P2(r,n,k) - P2(r,n1,k);
                          dr2[k] = P2(r,n2,k) - P2(r,n,k);
                  }
                  AppBound(dr1);
                  AppBound(dr2);

                  CalCosine(dr1, dr2, &c11, &c12, &c22, &cD, &c);
                  f = BEND_STIFF * Myacos(c);
                  for (k = 0; k < NDIM; k ++) {
                          fe1[k] = (c12 / c11) * dr1[k] - dr2[k];
                          fe2[k] = dr1[k] - (c12 / c22) * dr2[k];
                  }
                  mag1 = sqrt(c11 * DotProd(fe1, fe1));
                  mag2 = sqrt(c22 * DotProd(fe2, fe2));
                  if ( mag1 > REALMIN && mag2 > REALMIN) {
                          for (k = 0; k < NDIM; k ++) {
                                  f1 = f * fe1[k] / mag1;
                                  f2 = f * fe2[k] / mag2;
                                  P2(force,n1,k) += f1;
                                  P2(force,n,k) -= f1 + f2;
                                  P2(force,n2,k) += f2;
                          }
                  }
          }
  }
}

void ComputeBrownianForces (void)
{
  double *ip, r1, r2;
  int dedu;

  ip = Brown;
  dedu = (nAtom * NDIM) % 2;
  while (ip < Brown + nAtom * NDIM - dedu) {
```

```
        genrand_gauss(&r1, &r2);
        *ip = r1 * maxBrownianForce;          ip++;
        *ip = r2 * maxBrownianForce;          ip++;
  }
  if (dedu == 1) {
        genrand_gauss(&r1, &r2);
        *ip = r1 * maxBrownianForce;
  }
}

void RecordVMD(void) {
  int n, k, curr, next, count, amp = 1;
  double dr[NDIM], rr;
  FILE *fOut;

  fOut = fopen("vmd.pdb", "w");
  fprintf(fOut, "REMARK    Region  Monomers  Gap   DeltaT  ");
  fprintf(fOut, "REMARK\n");

  for (n = 0; n < nAtom; n++) {
        fprintf (fOut, "ATOM %6d  %c   LYS    1    ", n + 1, 'C');
        fprintf (fOut, "%8.3f%8.3f%8.3f  1.00  0.00     %s\n",
                 (P2(r,n,0) - regionH) * amp, (P2(r,n,1) - regionH) * amp,
                 (P2(r,n,2) - regionH) * amp, "A1");
  }
  fprintf(fOut, "ENDMDL\n");
  fclose(fOut);

  fOut = fopen("vmd.psf", "w");
  fprintf(fOut, "PSF CMAP\n\n");
  fprintf(fOut, "%8d !NTITLE\n\n", 7);
  fprintf(fOut, "%8d !NATOM\n", nAtom);

  for (n = 0; n < nAtom; n++) {
        fprintf (fOut, "%8d %-4s 1    LYS  %c%8d%11.5f%14.4f%11d\n",
                 n + 1, "A1", 'C', 11, 0.0, 12., 0);
  }
  count = 0;
  for (n = 0; n < nAtom; n++) {
        if (P2A(chains,n,0,2) < 0 && P2A(chains,n,1,2) > - 1) {
                 curr = n;
                 while (P2A(chains,curr,1,2) > -1) {
                          count++;
                          next = curr;
                          curr = P2A(chains,curr,1,2);

                          for (k = 0; k < NDIM; k++) {
                                   dr[k] = P2(r,next,k) - P2(r,curr,k) ;
                          }
                          rr = sqrt(DotProd (dr, dr));
                          if (rr > regionH) {
                                   count --;
                          }
                 }
        }
  }
  fprintf(fOut, "\n%8d !NBOND: bonds\n", count);


  count = 0;
  for (n = 0; n < nAtom; n++) {
        if (P2A(chains,n,0,2) < 0 && P2A(chains,n,1,2) > - 1) {
                 curr = n;
                 while (P2A(chains,curr,1,2) > -1) {
                          count++;
                          next = curr;
                          curr = P2A(chains,curr,1,2);
```

```
                        for (k = 0; k < NDIM; k++) {
                                dr[k] = P2(r,next,k) - P2(r,curr,k) ;
                        }
                        rr = sqrt(DotProd (dr, dr));
                        if ( rr < regionH ) {
                                fprintf(fOut, "%8d%8d", next + 1, curr + 1);
                                count ++;
                        }
                        if (count == 4) {
                                fprintf(fOut, "\n");
                                count = 0;
                        }
                }
        }
  }
  fclose(fOut);
}

void RecordFilamentLength (void)
{
  int n, len, curr, chainLen[nAtom + 2], numFila;
  double avgLeng, stdLeng;
  FILE *fOut;

  memset(chainLen, 0, sizeof(int) * (nAtom + 2) );
  numFila = 0; avgLeng = 0.; stdLeng = 0.;

  for (n = 0; n < nAtom; n ++) {
        // this is a monomer
        if (P2A(chains,n,0,2) < 0 && P2A(chains,n,1,2) < 0)
        { chainLen[1]++; }
        // this is a pointed end
        if (P2A(chains,n,0,2) > -1 && P2A(chains,n,1,2) < 0) {
                // If two monomers meet, min length of chain is 2
                len = 2;
                curr = P2A(chains,n,0,2);
                while (P2A(chains,curr,0,2) > -1) {
                        curr = P2A(chains,curr,0,2);
                        len++;
                }
                chainLen[len]++;
        }
  }

  // the array is (nAtom + 2) long
  // in case only one fiber forms with all monomers
  n = nAtom + 1;
  while (chainLen[n] == 0) {
    if (chainLen[n - 1] == 0) { n--; }
        // put flag at the end of useful information in the array
    else { chainLen[n] = -1; }
  }

  // Compute average and standard deviation
  n = 2;
  while (chainLen[n] != -1) {
    numFila += chainLen[n];
    avgLeng += n * chainLen[n];
    stdLeng += n * n * chainLen[n];
    n++;
  }
  avgLeng = avgLeng / (double)numFila;
  stdLeng = stdLeng / (double)numFila;
  stdLeng -= avgLeng * avgLeng;
  stdLeng = sqrt(stdLeng);
```

```
  // output
  fOut = fopen("filaLeng.dat", "w");
  n = 2;
  while (chainLen[n] != -1) {
    fprintf(fOut, "%d\t%d\n", n * (int)SCALE_DIA, chainLen[n]);
    n++;
  }
  fprintf (fOut, "# avg: %f, std dev: %f, # of filaments: %d\n",
                  avgLeng * SCALE_DIA, stdLeng * SCALE_DIA, numFila);
  fclose (fOut);
}

// print output
void RecordProgress(void) {
  FILE *fOut;

  printf("%9d %5d %5d %5d\n", stepCount, monomers, cntFila, cntElong);
  fOut = fopen("progress.dat", "a");
  fprintf(fOut, "%9d %5d %5d %5d\n", stepCount, monomers, cntFila, cntElong);
  fclose(fOut);
}

// calculate energy
void RecordEnergy(void) {
  int m, n, k, n1, n2, count, j2;
  double Ek = 0, Eb = 0, dr1[NDIM], dr2[NDIM];
  double rr, c11, c12, c22, cD, c, acosValue;
  FILE *fOut;

  // calculate bending energy
  count = 0;
  for (n = 0; n < nAtom; n++) {
          if (P2A(chains,n,0,2) > -1 && P2A(chains,n,1,2) > -1) {
                    count += 1;
                    n1 = P2A(chains,n,0,2);
                    n2 = P2A(chains,n,1,2);
                    for (k = 0; k < NDIM; k ++) {
                              dr1[k] = P2(r,n,k) - P2(r,n1,k);
                              dr2[k] = P2(r,n2,k) - P2(r,n,k);
                    }
                    AppBound(dr1);
                    AppBound(dr2);

                    CalCosine(dr1, dr2, &c11, &c12, &c22, &cD, &c);
                    acosValue = Myacos(c);
                    Eb += 0.5 * BEND_STIFF * acosValue * acosValue;
          }
  }
  if (count != 0) { Eb /= (double)count; }

  // calculate spring energy
  count = 0;
  for (n = 0; n < nAtom; n++) {
          for (m = 0; m < 2 ; m ++) {
                    j2 = P2A(chains,n,m,2);
                    if ( j2 > -1 ) {
                              for (k = 0; k < NDIM; k ++) {
                                        dr1[k] = P2(r,n,k) - P2(r,j2,k);
                              }
                              AppBound(dr1);
                              rr = sqrt( DotProd (dr1, dr1) );
                Ek += 0.5 * SPR_STIFF_FILA * (rr - 1.) * (rr - 1.);
                              count += 1;
                    }
    }
  }
  if (count != 0) { Ek /= (double)count; }
```

```
  // output
  fOut = fopen("energy.dat", "a");
  fprintf(fOut, "%d\t%lf\t%lf\n", stepCount, Eb, Ek);
  fclose(fOut);
}
```

2. tools.c

```
#include "header.h"

// Convert an integer value to string
char* Litoa(int val, int base){
    static char buf[32] = {0};
    int i = 30;
    for(; val && i ; --i, val /= base)
        buf[i] = "0123456789abcdef"[val % base];
    return &buf[i+1];
}

// Calculate a cross product of two vectors
void CrossProd(double *x, double *y, double *z)
{
  int k, c1, c2;
  for(k = 0; k < NDIM; k++) {
    c1 = (k + 1) % 3;
    c2 = (k + 2) % 3;
    z[k] = x[c1] * y[c2] - x[c2] * y[c1];
  }
}

// Calculate an inner product of two vectors
double DotProd (double *x, double *y)
{
  return (x[0] * y[0] + x[1] * y[1] + x[2] * y[2]);
}

// Apply boundary conditions
void AppBound(double *val) {
  int k;
  for (k = 0; k < NDIM; k++) {
    if (fabs(val[k]) > regionH) {
        val[k] -= SignR(region, val[k]);
    }
  }
}

// Reverse the sign of a value
double SignR (double x, double y)
{
  if (y >= 0.) return (x);
  else return (-x);
}

// A subroutine for ApplyBoundaryCond()
double ApplyBoundaryCondSub (double x) {
    double y;
    if (x >= region) { y = x - region; }
    else if (x < 0.) { y = x + region; }
    return y;
}

// Apply the periodic boundary condition on the positions of whole things
void ApplyBoundaryCond (void)
{
  int n;
  for (n = 0; n < nAtom * NDIM; n++) {
       if ( r[n] >= region || r[n] < 0. ) {
           r[n] = ApplyBoundaryCondSub(r[n]);
       }
  }
}
```

```
double Myacos(double c) {
  if ( c >  1. ) { c =  1.; }
  if ( c < -1. ) { c = -1.; }
  return acos(c);
}

double TrimAng(double angle) {
  if (angle < 0.) { angle = 0.; }
  else if (angle > 180.) { angle = 180.; }
  return angle;
}

void CalCosine(double *d1, double *d2, double *cc11, double *cc12, double *cc22, double *ccD, double *cc) {
  *cc11 = DotProd (d1, d1);
  *cc12 = DotProd (d1, d2);
  *cc22 = DotProd (d2, d2);
  *ccD = sqrt (*cc11 * (*cc22) );
  *cc = (*cc12) / (*ccD);
}
```

3. header.h

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>


// ************************* USER-DEFINED PARAMETERS ********************
#define NDIM                                    3         // This code works only in 3D
#define PI                                      3.141592

#define nAtom                                   8000 // Number of particles (must be cube)
#define GAP                                     2.0       // Distance between monomers in a cubic lattice

#define MIN_DIST_BOND_FILA   0.9       // Min. dist. for bond b/w monomers
#define MAX_DIST_BOND_FILA   1.2       // Max. dist. for bond b/w monomers
#define MMM_ANGDEV                      10.0
#define BEND_STIFF                      2428.6    // Bending stiffness of filaments
#define SPR_STIFF_FILA                  2000.0    // Spring constant between monomers

#define LAMBDA                                  1.0       // Epsilon/kT
#define SCALE_DIA                       7.0
#define LJ_MAR                                  0.8
#define BUFFER_LJ_MAR                   0.1
#define REALMIN                                 1e-10

#define P2(a,b,c)                       a[(b)*NDIM+(c)]
#define P2A(a,b,c,d)                    a[(b)*(d)+(c)]

// ******************* FUNCTION PROTOTYPES **********************
// main.cpp
void Initialize(void), Polymerize(void), CloseJob(void);
void ComputeRepulsiveForces (void), ComputeBendingForces (void);
void ComputeSpringForces (void), ComputeBrownianForces (void);
void UpdateChains (int, int, double), MeasurePreviousDisplacement(void);
void UpdateNeighborList(void), UpdateRpre(void);
void RecordVMD (void), RecordFilamentLength(void);
void RecordProgress(void), RecordEnergy(void);
void MoveParticles (void);

// tools.cpp
void CrossProd(double *, double *, double *);
void AppBound(double *), ApplyBoundaryCond(void);
void CalCosine(double *, double *, double *, double *, double *,
                   double *, double *);
double SignR(double, double), DotProd (double *, double *), Myacos(double);
double TrimAng(double), Myacos(double);
char *Litoa(int, int);

// rng.cpp
void init_genrand(unsigned long), genrand_gauss(double *, double *);
double genrand_real3(void);

// ******************* GLOBAL VARIABLES *********************************
extern double region, regionH, cellWid;
extern double bendStiff, deltaT, p_nucle, maxBrownianForce, rCut, angLMMM;
extern double *r, *force, *totDisp, *Brown, *rPre;
extern int stepCount, monomers, cells, atomsPerEdge, *cellCubic;
extern int *NL, NLCur, *chains, cntFila, cntElong;
```

```
extern time_t initTime;
```

4. calcmsd.c

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

#define REGION 10.
#define LENSCALE 7e-9
#define DT         20.134e-11

void RearrangeData(int, int);
void CalculateMSD(int, int);

int main(int argc, char *argv[]) {
  int nLine, nMono, nStep, CS = 1;
  char ch[80];
  FILE *fIn;

  fIn = fopen("traj.dat", "r");
  nLine = nMono = 0;
  while((fgets(ch, 80, fIn)) != NULL) {
          nLine++;
          if (CS > 0) { nMono++; }
          if (strcmp(ch, "\n") == 0) { CS = 0; }
  }
  nStep = nLine / nMono;
  nMono--;
  printf("Number of traced monomers = %d\n", nMono);
  printf("Number of time steps  = %d\n", nStep);
  fclose(fIn);

  RearrangeData(nMono, nStep);
  CalculateMSD(nMono, nStep);
}
// rearrange trajectory data in a different order
void RearrangeData(int nMono, int nStep) {
  double ***orig, ***modi, oft[3], region;
  int m, n, k;
  FILE *fIn, *fOut;

  // definition of arrays
  orig = (double ***)malloc(sizeof(double **) * nMono);
  modi = (double ***)malloc(sizeof(double **) * nMono);
  for(n = 0; n < nMono; n++) {
          orig[n] = (double **)malloc(sizeof(double *) * nStep);
          modi[n] = (double **)malloc(sizeof(double *) * nStep);
          for(k = 0; k < nStep; k++) {
                    orig[n][k] = (double *)malloc(sizeof(double) * 3);
                    modi[n][k] = (double *)malloc(sizeof(double) * 3);
          }
  }

  // read data from file
  fIn = fopen("traj.dat", "r");
  region = REGION;
  for(n = 0; n < nStep; n++) {
          for(k = 0; k < nMono; k++) {
                    fscanf(fIn, "%lf%lf%lf", &orig[k][n][0], &orig[k][n][1],
                              &orig[k][n][2]);
          }
  }
  // modify the data considering periodic boundary condition
  for(m = 0; m < nMono; m++) {
```

```
        for(k = 0; k < 3; k++) {
                oft[k] = 0.;
                modi[m][0][k] = orig[m][0][k];
        }

        for(n = 1; n < nStep; n++) {
                for(k = 0; k < 3; k++) {
                        if (fabs(orig[m][n][k] - orig[m][n - 1][k]) > region * 0.5) {
                                if (orig[m][n][k] > orig[m][n - 1][k]) { oft[k] -= region; }
                                else { oft[k] += region; }
                        }
                        modi[m][n][k] = orig[m][n][k] + oft[k];
                }
        }
  }
  // write the modified trajectory
  fOut = fopen("modiTraj.dat", "w");
  for(n = 0; n < nMono; n++) {
        for(k = 0; k < nStep ; k++) {
                fprintf(fOut,"%g\t%g\t%g\n", modi[n][k][0], modi[n][k][1],
                                modi[n][k][2]);
        }
        fprintf(fOut, "\n");
  }
  fclose(fIn);

  for(n = 0; n < nMono; n++) {
        for(k = 0; k < nStep; k++) {
                free(orig[n][k]);
                free(modi[n][k]);
        }
        free(orig[n]);
        free(modi[n]);
  }
  free(orig);
  free(modi);
}

void CalculateMSD(int nMono, int nStep) {
  int m, n, k, l, intv = 4967;
  double *msd, *t, sum, **traj;
  FILE *fIn, *fOut;

  // definition of arrays and the initialization of the MSD array
  msd = (double *)malloc(sizeof(double)*nStep);
  t = (double *)malloc(sizeof(double)*nStep);
  traj = (double **)malloc(sizeof(double *)*nStep);
  for (n = 0; n < nStep; n++) {
        traj[n] = (double *)malloc(sizeof(double)*3);
        msd[n] = 0.;
  }

  fIn = fopen("modiTraj.dat", "r");

  // read data and accumulate msd data
  for (m = 0; m < nMono; m++) {
    for(n = 0; n < nStep; n++) {
                fscanf(fIn, "%lf%lf%lf", &traj[n][0], &traj[n][1], &traj[n][2]);
        }
        for(n = 1; n < nStep; n++) {
                sum = 0;
                for(k = 0; k < nStep - n; k++) {
                        for(l = 0; l < 3; l++) {
                                sum += (traj[k+n][l]-traj[k][l]) * (traj[k+n][l]-traj[k][l]);
                        }
                }
                // ensemble over time steps
```

```
                msd[n] += sum / (double)(nStep - n);
        }
        printf("%d is done!!\n", m);
  }
  fclose(fIn);

  // ensemble over monomers
  for (n = 0; n < nStep - 1; n++) {
        t[n] = (double)(n * intv) * DT;
        msd[n] /= (double)nMono;
        msd[n] *= LENSCALE * LENSCALE;
  }

  // write the MSD data as a file
  fOut = fopen("msd.dat", "w");
  for (n = 0; n < nStep - 2; n++) {
        fprintf(fOut, "%g\t%g\n", t[n], msd[n]);
  }
  fclose(fOut);
}
```

5. rng.c

```
#include <math.h>

/*
   A C-program for MT19937, with initialization improved 2002/1/26.
   Coded by Takuji Nishimura and Makoto Matsumoto.

   Before using, initialize the state by using init_genrand(seed)
   or init_by_array(init_key, key_length).

   Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions
   are met:

     1. Redistributions of source code must retain the above copyright
        notice, this list of conditions and the following disclaimer.

     2. Redistributions in binary form must reproduce the above copyright
        notice, this list of conditions and the following disclaimer in the
        documentation and/or other materials provided with the distribution.

     3. The names of its contributors may not be used to endorse or promote
        products derived from this software without specific prior written
        permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
   "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
   LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
   A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
   CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
   PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
   LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
   NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
   SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


   Any feedback is very welcome.
   http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
   email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
*/

#include <stdio.h>

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0dfUL   /* constant vector a */
#define UPPER_MASK 0x80000000UL /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL /* least significant r bits */

static unsigned long mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */

/* initializes mt[N] with a seed */
void init_genrand(unsigned long s)
{
    mt[0]= s & 0xffffffffUL;
    for (mti=1; mti<N; mti++) {
        mt[mti] =
```

```
            (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
        /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
        /* In the previous versions, MSBs of the seed affect   */
        /* only MSBs of the array mt[].                        */
        /* 2002/01/09 modified by Makoto Matsumoto             */
        mt[mti] &= 0xffffffffUL;
        /* for >32 bit machines */
    }
}

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
/* slight change for C++, 2004/2/26 */
void init_by_array(unsigned long init_key[], int key_length)
{
    int i, j, k;
    init_genrand(19650218UL);
    i=1; j=0;
    k = (N>key_length ? N : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525UL))
          + init_key[j] + j; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++; j++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
        if (j>=key_length) j=0;
    }
    for (k=N-1; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941UL))
          - i; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
    }

    mt[0] = 0x80000000UL; /* MSB is 1; assuring non-zero initial array */
}

/* generates a random number on [0,0xffffffff]-interval */
unsigned long genrand_int32(void)
{
    unsigned long y;
    static unsigned long mag01[2]={0x0UL, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if init_genrand() has not been called, */
            init_genrand(5489UL); /* a default initial seed is used */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];

        mti = 0;
    }

    y = mt[mti++];
```

```
    /* Tempering */
    y ^= (y >> 11);
    y ^= (y << 7) & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);

    return y;
}

/* generates a random number on [0,0x7fffffff]-interval */
long genrand_int31(void)
{
    return (long)(genrand_int32()>>1);
}

/* generates a random number on [0,1]-real-interval */
double genrand_real1(void)
{
    return genrand_int32()*(1.0/4294967295.0);
    /* divided by 2^32-1 */
}

/* generates a random number on [0,1)-real-interval */
double genrand_real2(void)
{
    return genrand_int32()*(1.0/4294967296.0);
    /* divided by 2^32 */
}

/* generates a random number on (0,1)-real-interval */
double genrand_real3(void)
{
    unsigned long y;
    static unsigned long mag01[2]={0x0UL, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if init_genrand() has not been called, */
            init_genrand(5489UL); /* a default initial seed is used */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];

        mti = 0;
    }

    y = mt[mti++];

    /* Tempering */
    y ^= (y >> 11);
    y ^= (y << 7) & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);

    return (((double)y) + 0.5)*(1.0/4294967296.0);
    /* divided by 2^32 */
```

```
}

/* generates a random number on [0,1) with 53-bit resolution*/
double genrand_res53(void)
{
   unsigned long a=genrand_int32()>>5, b=genrand_int32()>>6;
   return(a*67108864.0+b)*(1.0/9007199254740992.0);
}

/* Added by Tae Yoon Kim in January 2006 */
void genrand_gauss(double *r1, double *r2)
{
  double x1, x2, w, y1, y2;
  do {
          x1 = 2.0 * genrand_real3() - 1.0;
          x2 = 2.0 * genrand_real3() - 1.0;
          w = x1*x1 + x2*x2;
  } while (w >= 1.0);

  w = sqrt(( -2.0 * log( w )) / w);
  y1 = x1 * w;
  y2 = x2 * w;

  if (y1 >= 3.0) y1 = 3.0;
  else if (y1 <= -3.0) y1 = -3.0;

  *r1 = y1;
  *r2 = y2;
}
```