Appendix C

Numerical Techniques in C++[1]

What follows are four programs that exemplify some of the concepts covered in the previous appendix on C++. Each program applies a different numerical technique to solving particular kinds of problems in electromagnetics. The methods are namely:

1) Finite Difference
2) Finite Element
3) Transmission-line-matrix
4) Monte Carlo Fixed-Random walk

Listings 1 to 4, respectively, correspond to the FORTRAN programs in Figure 3.31 on FDTD, Figure 6.10 on the finite element method, Figure 7.14 on TLM, and Figure 8.13 on the Monte Carlo method.

---

[1] Written by Raymond C. Garcia, Georgia Institute of Technology, Atlanta, GA 30332-0320.

Listing 1: Finite Difference Program:

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>


#define IMAX 19
#define JMAX 39
#define KMAX 19
#define ITER 100
#define NMAX 2
#define NHW 40
#define MED 2
#define JS 3
const double DELTA = 3E-3;
const double CL = 3.0E8;
const double F = 2.5E9;
const double PIE = 3.141592654;

// scatter dimensions
#define OI 19.5
#define OJ 20
#define OK 19
#define RADIUS 15

class matrix{
   int a,b,c,d;
   int amax,bmax,cmax,dmax;
   float *data;
   int checkMinus(int index);
   void getSize(void){cout << amax <<"x"<<
   bmax <<"x"<< cmax <<"x"<< dmax << endl;}
public:
   // constructors
   matrix();
   matrix(int s);
   matrix(int d1,int d2);
   matrix(int d1,int d2,int d3);
   matrix(int d1,int d2,int d3, int d4);

   void dumpit();

   // overload the parens
   float &operator()(int a);
   float &operator()(int a,int b);
   float &operator()(int a,int b,int c);
   float &operator()(int a,int b,int c, int d);
};


matrix::matrix(){
   cout << "void constructor\n";
}
```

```
matrix::matrix(int s){
   data = new float[s];
   amax=s,bmax=0,cmax=0,dmax=0;
   for (a=0;a<amax;a++)
      data[a]=0;
}

matrix::matrix(int d1,int d2){
   data = new float[d1*d2];
   amax=d1; bmax=d2,cmax=0,dmax=0;

   for(a=0;a<d1;a++)
      for(b=0;b<d2;b++)
         data[a*d2 + b]=0;
}

matrix::matrix(int d1,int d2,int d3){
   data = new float[d1*d2*d3];
   amax=d1; bmax=d2; cmax=d3,dmax=0;

   for(a=0;a<d1;a++)
      for(b=0;b<d2;b++)
         for(c=0;c<d3;c++)
            data[a*d2*d3 + b*d3 + c]=0;
}

matrix::matrix(int d1,int d2,int d3, int d4){
   data = new float[d1*d2*d3*d4];
   amax=d1; bmax=d2; cmax=d3; dmax=d4;

   for(a=0;a<d1;a++)
      for(b=0;b<d2;b++)
         for(c=0;c<d3;c++)
            for(d=0;d<d4;d++)
               data[a*d2*d3*d4 + b*d3*d4 + c*d4 + d]=0;
}


void matrix::dumpit(){

   if (bmax==0){
      for (a=0;a<amax;a++)
         cout <<   data[a] << endl;
   }
   else if (cmax==0){
      for(a=0;a<amax;a++){
         for(b=0;b<bmax;b++)
            cout <<   data[a*bmax + b]<<" ";
         cout << endl;
      }
}
else if (dmax==0){
    for(a=0;a<amax;a++){
        for(b=0;b<bmax;b++){
            for(c=0;c<cmax;c++)
                cout << data[a*bmax*cmax + b*cmax + c]<<" ";
            cout << endl;
        }
        cout << endl;
    }
}
else{
        for(a=0;a<amax;a++){
            for(b=0;b<bmax;b++){
                for(c=0;c<cmax;c++){
                    for(d=0;d<dmax;d++)
                        cout << data[a*bmax*cmax*dmax
                        + b*cmax*dmax + c*dmax + d]<<" ";
                    cout << endl;
                }
                cout << endl;
            }
            cout << endl;
        }
    }
}

int matrix::checkMinus(int index){
    if (index < 0){
        //getSize();
        return 0;
    }
    else
        return index;
}

float & matrix::operator()(int a)
{a = checkMinus(a);return data[a];}
float & matrix::operator()(int a,int b)
{a = checkMinus(a); b=checkMinus(b);return data[a*bmax + b];}
float & matrix::operator()(int a,int b,int c){
    a=checkMinus(a);
    b=checkMinus(b);
    c=checkMinus(c);
    return data[a*bmax*cmax + b*cmax + c];
}
float & matrix::operator()(int a,int b,int c, int d){
    a=checkMinus(a);
    b=checkMinus(b);
    c=checkMinus(c);
    d=checkMinus(d);
    return data[a*bmax*cmax*dmax + b*cmax*dmax + c*dmax + d];
}
```

```
class pdesc{
   int a,b,c,d;
   double e0;  // permitivity
   double u0;  // permeability
   double dt;
   double r;
   double ra;
   double rb;
   double tpifdt;

   matrix ex, ey, ez;
   matrix hx, hy, hz;
   matrix ey1, ez1;
   matrix er, sig;
   matrix ca, cb;
   matrix ixmed, iymed, izmed;
   matrix cbmrb;


   // methods
   double position(double ri,double rj,double rk){
      return( sqrt(pow((ri-OI),2) + pow((rj-OJ),2)
      + pow((rk-OK),2) ) );
   };

public:

   // constructor
   pdesc():ex(IMAX+1,JMAX+1,KMAX+1,NMAX+1),
          ey(IMAX+1,JMAX+1,KMAX+1,NMAX+1),
          ez(IMAX+1,JMAX+1,KMAX+1,NMAX+1),
          hx(IMAX+1,JMAX+1,KMAX+1,NMAX+1),
          hy(IMAX+1,JMAX+1,KMAX+1,NMAX+1),
          hz(IMAX+1,JMAX+1,KMAX+1,NMAX+1),
          ey1(JMAX+1),
          ez1(JMAX+1),
          er(MED),
          sig(MED),
          ca(MED),
          cb(MED),
          ixmed(IMAX+1,JMAX+1,KMAX+1),
          iymed(IMAX+1,JMAX+1,KMAX+1),
          izmed(IMAX+1,JMAX+1,KMAX+1),
          cbmrb(2)
   {
      e0=(1e-9)/(36*PIE);  // permitivity
      u0=(1e-7)*4*PIE;
      dt=DELTA/(2*CL);
      r=dt/e0;
      ra=pow(dt,2)/(u0*e0*pow(DELTA,2));
      rb=dt/(u0*DELTA);
      tpifdt = 2*PIE*F*dt;

      er(0)=1.0;er(1)=4.0;
      sig(0)=0.1;sig(1)=0;
   }

   void compute_media_parms(); // step #1
   void gen_field_components(); // step #2
   void create_output(char*);
};

void pdesc::create_output(char* fname)
{
   int j;
   ofstream out(fname);
   if (!out)
      cout << "There is a problem with opening a file.\n";

   cout << "ey1\n";
   out << "ey1\n";

   for (j=0;j<=JMAX;j++){
      out << ey1(j);
      cout << ey1(j);
   }

   cout << "\nez1\n";
   out << "\nez1\n";


   for (j=0;j<=JMAX;j++){
      out << ez1(j);
      cout << ez1(j);
    }
    cout << endl;

 }
 void pdesc::gen_field_components()
 {
    // since only field components at current time (t)
    // and previous two time steps (t-1 and t-2) are
    // required for computation,

    // we save memory space by using the following indices
    // ncur is index in for time t
    // npr1 is index in for t-1
    // npr2 is index in for t-2
```

```
int ncur = 2, npr1 = 1,npr2 = 0;
int i,j,k,nn;  // loop indices
double m,cam,temp;

for (nn =1; nn<=ITER; nn++){
   if ((nn % 10) == 0)
      cout << "Iteration Count: " << nn << endl; // status

   npr2 = npr1;
   npr1 = ncur;
   ncur = (ncur+1) % 3;

   for (k=0; k<=KMAX;k++)
      for (j=0; j<=JMAX;j++){
         for (i=0;i<=IMAX;i++){
            if (i == 0){ // x=delta/2
               if ((k != KMAX) && (k != 0)){
                  hy(0,j,k,ncur) = (hy(1,j,k-1,npr2) +
                                 hy(1,j,k,npr2) +
                                 hy(1,j,k+1,npr2))/3;
                  hz(0,j,k,ncur) = (hz(1,j,k-1,npr2) +
                                 hz(1,j,k,npr2) +
                                 hz(1,j,k+1,npr2))/3;
               }
               else if (k == KMAX){
                     hy(0,j,KMAX,ncur) =
                     (hy(1,j,KMAX-1,npr2) +
                     hy(1,j,KMAX,npr2))/2;
                     hz(0,j,k,ncur) = (hz(1,j,k-1,npr2) +
                                    hz(1,j,k,npr2))/2;
                  }
               else {
                  hy(0,j,k,ncur) = (hy(1,j,k,npr2) +
                                 hy(1,j,k+1,npr2))/2;
                  hz(0,j,0,npr2) = (hz(1,j,0,npr2) +
                                 hz(1,j,1,npr2))/2;
               }
            }
            if (j==0){ // y = 0
               ex(i,0,k,ncur) = ex(i,1,k,npr2);
               ez(i,0,k,ncur) = ez(i,1,k,npr2);
            }
            else if (j == JMAX){ // y = ymax
               ex(i,JMAX,k,ncur) = ex(i,JMAX-1,k,npr2);
               ez(i,JMAX,k,ncur) = ez(i,JMAX-1,k,npr2);
            }
            if (k == 0){ // z=0
               if ((i != 0) && (i != IMAX)){
                      ex(i,j,0,ncur) = (ex(i-1,j,1,npr2) +
                                     ex(i,j,1,npr2) +
                                     ex(i+1,j,1,npr2))/3;
                      ey(i,j,0,ncur) = (ey(i-1,j,1,npr2) +
                                     ey(i,j,1,npr2) +
                                     ey(i+1,j,1,npr2))/3;
                   }
                   else if (i == 0){
                      ex(0,j,0,ncur) = (ex(0,j,1,npr2) +
                                     ex(1,j,1,npr2))/2;
                      ey(i,j,0,ncur) = (ey(i,j,1,npr2) +
                                     ey(i+1,j,1,npr2))/2;
                   }
                   else{
                      ex(i,j,0,ncur) = (ex(i-1,j,1,npr2) +
                                     ex(i,j,1,npr2))/2;
                      ey(i,j,0,ncur) = (ey(i-1,j,1,npr2) +
                                     ey(i,j,1,npr2))/2;
                   }
                }
//
//     (iii) apply FD/TD algorithm
//
                // hx generation
                hx(i,j,k,ncur) = hx(i,j,k,npr1) +
                              rb*(ey(i,j,k+1,npr1) -
                              ey(i,j,k,npr1) +
                              ez(i,j,k,npr1) -
                              ez(i,j+1,k,npr1));
                // hy generation
                hy(i,j,k,ncur) = hy(i,j,k,npr1) +
                              rb*(ez(i+1,j,k,npr1) -
                              ez(i,j,k,npr1) +
                              ex(i,j,k,npr1) -
                              ex(i,j,k+1,npr1));
                // hz generation
                hz(i,j,k,ncur) = hz(i,j,k,npr1) +
                              rb*(ex(i,j+1,k,npr1) -
                              ex(i,j,k,npr1) +
                              ey(i,j,k,npr1) -
                              ey(i+1,j,k,npr1));
                // k = kmax  : symmetry
                if (k == KMAX){
                   hx(i,j,KMAX,ncur) = hx(i,j,KMAX-1,ncur);
                   hy(i,j,KMAX,ncur) = hy(i,j,KMAX-1,ncur);
```

```
}
// ex generation
if ((j != 0) && (j != JMAX) && (k != 0)){
   m = ixmed(i,j,k);
   ex(i,j,k,ncur) = ca(m)*ex(i,j,k,npr1) +
                  cbmrb(m)*(hz(i,j,k,ncur) -
                            hz(i,j-1,k,ncur) +
                           hy(i,j,k-1,ncur) -
                           hy(i,j,k,ncur));
}
// ey generation
if (k != 0){
   m = iymed(i,j,k);
   ey(i,j,k,ncur) = ca(m)*ey(i,j,k,npr1) +
                  cbmrb(m)*(hx(i,j,k,ncur) -
                            hx(i,j,k-1,ncur) +
                           hz(i-1,j,k,ncur) -
                           hz(i,j,k,ncur));

}
// ez generation
if ((j != 0) && (j != JMAX)){
   m = izmed(i,j,k);
   // sig(ext) = for Ez only from Taflove[14]
   if (m == 1)
      cam = 1;
   else
      cam = ca(m);

   ez(i,j,k,ncur) = cam*ez(i,j,k,npr1) +
                  cbmrb(m)*(hy(i,j,k,ncur) -
                           hy(i-1,j,k,ncur) +
                           hx(i,j-1,k,ncur) -
                           hx(i,j,k,ncur));

   // (iv) apply the plane-wave source
   if (j == JS)
      ez(i,JS,k,ncur) = ez(i,JS,k,ncur) +
      sin(tpifdt*nn);
}
if (j == IMAX){ // i = imax +1/2 : symmetry
   ey(IMAX+1,j,k,ncur) = ey(IMAX,j,k,ncur);
   ez(IMAX+1,j,k,ncur) = ez(IMAX,j,k,ncur);
```

```
                    }
                    if (k == KMAX){ // k = kmax
                        ex(i,j,KMAX+1,ncur) = ex(i,j,KMAX-1,ncur);
                        ey(i,j,KMAX+1,ncur) = ey(i,j,KMAX-1,ncur);
                    }
                    }
                //---------------------------------------------
                // Step 4 - retain the maximum absolute values
                //          during the last half-wave
                //---------------------------------------------
                if ((k== KMAX) && (nn > (ITER-NHW))){
                    temp = abs(ey(IMAX,j,KMAX-1,ncur));
                    if (temp > ey1(j))
                        ey1(j) = temp;
                    temp = abs(ez(IMAX,j,KMAX,ncur));
                    if (temp > ez1(j))
                        ez1(j) = temp;
                }
        }
    }

}
void pdesc::compute_media_parms()
{
    int m,i,j,k;

    for (m=0;m<MED;m++){
        ca(m)=1-r*sig(m)/er(m);
        cb(m)=ra/er(m);
        cbmrb(m) = cb(m)/rb;
    }

  //  (i) Calculate the real/actual grid points
  // Initialize the media arrays.  Index (m) determines
  // which medium each point is actually located in and
  // is used to index into arrays which determine the
  // constitutive parameters of the medium.  There are
  // separate M determining arrays as opposed to computing
  // them each time they are needed saves a large amount
  // of computation time.

    for (i=0;i<=IMAX;i++)
        for (j=0;j<=JMAX;j++)
            for (k=0;k<=KMAX;k++){
                if (position(i+.5,j,k) <= RADIUS)
                    ixmed(i,j,k) = 1;
                else
                    ixmed(i,j,k) = 0;
            if (position(i,j+.5,k) <= RADIUS)
               iymed(i,j,k) = 1;
            else
               iymed(i,j,k) = 0;

            if (position(i,j,k+.5) <= RADIUS)
               izmed(i,j,k) = 1;
            else
               izmed(i,j,k) = 0;
         }
}

void main()
{

   pdesc prob1;


   prob1.compute_media_parms(); // step #1
   prob1.gen_field_components(); // step #2
   prob1.create_output("fd1.out");
}
```

Listing 2: Finite Element Program:

```
// Finite Element solution of Laplace's Equation for
// two-dimensional Problems
// Triangular elements are used.
//
// The Unknown potentials are obtained using
// the iteration method.

/************************************************************/
/***********         Variable Descriptions        ***********/
/************************************************************/
/** ND = # of nodes                                        **/
/** NE = # of Elements                                     **/
/** NP = # of Fixed nodes ( where potential is Prescribed  **/
/** NDP(I) = Node # of prescribed potential, I = 1,2,...NP **/
/** VAL(I) = Value of prescribed potential at node NDP(I)  **/
/** NL(I,J) = List of nodes for each element I, where      **/
/** LF(I) = List of free nodes I = 1,2,...,NF = ND - NP    **/
/** J = 1, 2, 3, is the local node number                  **/
/** CE(I,J) = Element Coefficient Matrix                   **/
/** ER(I) = Value of the relative permittivity for         **/
/**         element I                                      **/
/** C(I,J) = Global Coefficient Matrix                     **/
/** X(I), Y(I) = Global Coordinates of Node I              **/
/** XL(J), YL(J) = Local Coordinates of Node J = 1,2,3     **/
/** V(I) = Potential at node I                             **/
/** Matrices P(I) and Q(I) are Defined earlier in the text **/
/************************************************************/

/******************************************************/
/******************* Please Note **********************/
/******************************************************/
/**                                                  **/
/** This program assumes the input data is in a file **/
/** called "fem.in" in the following format:         **/
/**                                                  **/
/**           NE        ND        NP            **/
/**      NL(I,1)   NL(I,2)   NL(I,3)            **/
/**                    ...                **/
/**                    ...                           **/
/**                    ...                           **/
/**      NL(I+a,1) NL(I+a,2) NL(I+a,3)          **/
/**      X(I)    Y(I)                           **/
/**            ...                           **/
/**            ...                                   **/
/**            ...                                   **/
/**      X(I+b)    Y(I+b)                       **/
/**       NDP(I)  VAL(I)                             **/
/**            ...                                   **/
/**            ...                                   **/
/**            ...                                   **/
/**       NDP(I+c)  VAL(I+c)                         **/
/**                                                  **/
/** The output is in a file called "fem.out"          **/
/*****************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>

class CNodeCoordRecord{
public: double x,y;
   CNodeCoordRecord(){x=0.0;y=0.0;};

   friend istream &operator<<(istream &istr,
   CNodeCoordRecord obj); // used to read in values
};

istream &operator>>(istream &istr, CNodeCoordRecord &obj)
{
   istr >> obj.x;
   istr >> obj.y;

   return istr;
}


class CElementRecord{
public:   int FirstLocalNodeNumber,SecondLocalNodeNumber,
      ThirdLocalNodeNumber;
      CElementRecord(){FirstLocalNodeNumber=0;
                   SecondLocalNodeNumber=0;
                   ThirdLocalNodeNumber=0;};
   friend istream &operator<<(istream &istr,
   CElementRecord obj);  // used to read in values
};

istream &operator>>(istream &istr, CElementRecord &obj)
{
   istr >> obj.FirstLocalNodeNumber;
   istr >> obj.SecondLocalNodeNumber;
   istr >> obj.ThirdLocalNodeNumber;

   return istr;
}
```

```
class CPotentialRecord{
public:   int NodeNumber;
public:   double PrescribedPotential;
       CPotentialRecord(){NodeNumber = 0;
       PrescribedPotential=0.0;};

   friend istream &operator<<(istream &istr,
   CPotentialRecord obj);  // used to read in values
};

istream &operator>>(istream &istr, CPotentialRecord &obj)
{
   istr >> obj.NodeNumber;
   istr >> obj.PrescribedPotential;

   return istr;
}

void main()
{
   double x[100], y[100],c[100][100], ce[100][100];
   double ndp[100], val[100];
   double v[100], p[3], q[3], xl[3], yl[3], er[100];
   int nl[100][3], lf[100];


   /**************************************************/
   /** Get Geometry and boundary conditions from the **/
   /** command line.                                **/
   /**************************************************/

   intni = 50,ne,nd,np; // # of iterations
   ofstream out("fem.out");
   if (!out)
   {
      cout << "Error with creating output file 'fem.out.'\n";
      exit(0);
   }
   ifstream in("fem.in");
   if (!in)
   {
      cout << "Error with opening file.
      Make sure the file 'fem.in' exists\n";
      exit(0);
   }

   in >> ne; // get the number of elements
   in >> nd; // get the number of nodes
   in >> np; // get the number prescribed nodes
```

```
CElementRecord ElRec;
   // instantiate a CElementRecord object
CNodeCoordRecord NodeRec;
   // instantiate a CNodeCoordRecord object
CPotentialRecord PotRec;
   // instantiate a CPotentialRecord object
// populate nl
int i;
for (i = 0;i<ne;i++){
   in >> ElRec;
   nl[i][0] = ElRec.FirstLocalNodeNumber-1;
   nl[i][1] = ElRec.SecondLocalNodeNumber-1;
   nl[i][2] = ElRec.ThirdLocalNodeNumber-1;
}

// populate x and y
for (i = 0;i<nd;i++){
   in >> NodeRec;
   x[i] = NodeRec.x;
   y[i] = NodeRec.y;
}

// populate ndp and val
for (i = 0;i<np;i++){
   in >> PotRec;
   ndp[i] = PotRec.NodeNumber-1;
   val[i] = PotRec.PrescribedPotential;
}

in.close();
/****************************************************/

/***************************************************/
/** Evaluate Coefficient Matrix for each element **/
/** and assemble globally.                        **/
/***************************************************/
int m,n;
for (m =0;m<nd;m++)
   for (n=0;n<nd;n++)
      c[m][n] = 0.0;
int j,l,k;
double area;
for (i = 0;i<ne;i++)
{
   for(j=0;j<3;j++)
   {
      k = nl[i][j];
      xl[j] = x[k];
      yl[j] = y[k];
   }
   p[0] = yl[1] - yl[2];
   p[1] = yl[2] - yl[0];
   p[2] = yl[0] - yl[1];
   q[0] = xl[2] - xl[1];
   q[1] = xl[0] - xl[2];
   q[2] = xl[1] - xl[0];
   area = 0.5*fabs(p[1]*q[2] - q[1]*p[2]);

   // determine the coefficient matrix for element i
   for (m = 0;m<3;m++)
      for (n=0;n<3;n++)
         ce[m][n] = (p[m]*p[n] + q[m]*q[n])/(4.0*area);
   // assemble globally - find c[i][j]
   int ir,ic;
   for (j=0;j<3;j++)
   {
      ir = nl[i][j];
      for(l=0;l<3;l++)
      {
         ic = nl[i][l];
         c[ir][ic] += ce[j][l];
      }
   }

}

/*********************************************/
/** Solve the resulting system Iteratively **/
/*********************************************/

// initialize and determine lf[i] - list of
// free nodes i

int nf = 0;
int PrescribedNodeFlag = 0;
for (i=0;i<nd;i++)
{
   v[i] = 0.0;
   for(k=0;k<np;k++)
   {
      if (i == ndp[k])
      {
         v[i] = val[k];
         out << i << " " << v[i] << endl;
         PrescribedNodeFlag =1;
         break;
```

```
            }
        }
        if (!PrescribedNodeFlag)
        {
            lf[nf++] = i; // if i is not a prescribed node,
            it is free
        }
        else
            PrescribedNodeFlag = 0;
    }

    out << nf << " " << nd - np << " Check if these are equal.\n";

    // apply the iterative method
    double sum;
    for (n = 0; n< ni; n++)
    {
        for( i = 0;i<nf;i++)
        {
            sum = 0.0;
            k = lf[i];
            for (j = 0;j<nd;j++)
            {
                if (j!=k)
                    sum+= (v[j]*c[j][k]);
            }
            v[k] = -sum/c[k][k]; // applies only to free nodes
        }
    }

    /************************/
    /** Output the results **/
    /************************/
    out << "       # of Nodes: " << nd << endl;
    out << "    # of Elements: " << ne << endl;
    out << " # of Fixed Nodes: " << np << endl;

    for (i=0;i<nd;i++)
        out << "I: " << i << "\tx[" << i << "]\t"<< x[i]
            << "\t\ty[" << i << "]\t" << y[i]
            << "\t\tv[" << i << "]\t" << v[i] << endl;
}
```

Listing 3: Transmission-line-matrix:

```
/*******************************************************/
/** This Program applies the tlm method to solve        **/
/** One-dimensional wave problems.  The Specific Example **/
/** is described as follows:                            **/
/**                                                     **/
/** The TEM waves on a 25 x 11 matrix                   **/
/** The boundaries are at x = 2 and x = 10.             **/
/** Initial impulse excitation is along z = 4 at t = 0   **/
/** and subsequently this line is set to zero.  The grid **/
/** is terminated at z = 25.  Output is taken at z = 14, **/
/** x = 6 for Ey and Hx for 100, 150, 200 iterations     **/
/**                                                     **/
/** vi(it, i j k) -- array for incident voltage          **/
/** vr(it,i,j,k)  -- array for reflected voltage         **/
/** it = 1        -- for previous pulse value            **/
/** it = 2        -- for current pulse value             **/
/** i,j           -- correspond to node location (z,x)   **/
/** k = 1 ..4     -- for terminals,                      **/
/** nx            -- index of nodes in x-direction       **/
/** nz            -- index of nodes in z-direction       **/
/** nx/nz b,e     -- index of beginning, end nod         **/
/** nx/nz 0       -- index of output node                **/
/** Gamma         -- reflection of coefficient at the    **/
/**                      boundary c                      **/
/** Delta         -- Mesh Size Divided by lambda         **/
/** itrate        -- # of iterations                     **/
/*******************************************************/

/***************************************************/
/****************** Please Note *********************/
/***************************************************/
/** The output is in a file called "tlm.out"        **/
/***************************************************/

#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>

double transc(double delta)
{
   double lambda,lambdac;
   double pie = 3.14159265;
   double teta = pie*delta;
   double tet = sqrt(2.0)*sin(teta);
   return(teta/asin(tet));
}

// This is a support class to handle complex numbers
```

```
class CComplex
{
public: double Real, Imag;
   CComplex(){Real = 0;Imag = 0;};
    // constructor takes imag and real parts

   double mag();
   double ang();
};

double CComplex::mag()
{
   double result;

   result = sqrt(pow(Real,2) + pow(Imag,2));
   return(result);
}

double CComplex::ang()
{
   double result,pie = 3.1415927;

   result = atan(Imag/Real);

   if (Imag < 0 && Real > 0 && result
      >= pie/2 && result <= pie) // 4th quadrant
      result -= pie;
   if (Imag < 0 && Real < 0 && result
      >= 0 && result <= pie/2)   // 3rd quadrant
      result += pie;
   if (Imag > 0 && Real < 0 && result
      >= 3*pie/2 && result <= 2*pie)   // 2nd quadrant
      result -= pie;

   return (result);
}

void main()
{
   double vi[2][25][11][4], vr[2][25][11][4];
   double out[20][10], efi[20], efr[20], hfi[20],
   hfr[20]; int nxb=2,nxe=10,nzb=4,nze=24,nt=4,
   itrate=200, nxo=6,nzo=14, ii;
   double pie = 3.1415927,gamma=0,delta =.002;
   double ei = 0.0,hi,del;

   for (int row = 0;row<20;row++)
      for (int col =0;col<10;col++)
         out[row][col] = 0;
```

```
ofstream outf("tlm.out");
if (!outf)
{
   cout << "Error with creating output file 'tlm.out.'\n";
   exit(0);
}

/********************* Step 1 ***********************/
/** Insert initial pulse excitation along line z=4 **/
/****************************************************/
int j;

for (j=nxb-1;j<nxe;j++)
   vi[0][nzb][j][1] = 1.0;

/********************* Step 2 ***********************/
/** Calculate the reflected voltage and submit it  **/
/** directly to the neighboring node.              **/
/****************************************************/
int itime,i,k,it;
double sum;
for (itime = 0;itime < itrate; itime++){
   it = 1;
   for (i=nzb;i<nze;i++){
      for(j=nxb-1;j<nxe;j++){
         sum = 0.0;
         for (k=0;k<nt;k++)
            sum += vi[it-1][i][j][k];
         for (k=0;k<nt;k++)
            vr[it][i][j][k] = 0.5*sum - vi[it-1][i][j][k];
         vi[it][i][j-1][2] = vr[it][i][j][0];
         vi[it][i-1][j][3] = vr[it][i][j][1];
         vi[it][i+1][j][0] = vr[it][i][j][2];
         vi[it][i+1][j][1] = vr[it][i][j][3];

/********************* Step 3 ***********************/
/**          Insert boundary conditions            **/
/****************************************************/
         if (j == nxe-1)
            vi[it][i][nxe-1][2] = vr[it][i][nxe-1][0];
         if (j == nxb-1)
            vi[it][i][nxb-1][0] = vr[it][i][nxb-1][2];
         if (j == nze-1)
            vi[it][nze-1][j][3] = gamma*vr[it][nze-1][j][3];
      }
   }
```

```
/******************** Step 4 **********************/
/** Calculate impulse response of Ey and Hx at     **/
/** z = nzo, x = nxo                               **/
/**************************************************/
   for (k=0;k<nt;k++)
      ei += vi[it][nzo-1][nxo-1][k]*0.5;
   hi = vi[it][nzo-1][nxo-1][1] - vi[it][nzo-1][nxo-1][3];

   // sum the frequency response (imaginary and real
   // parts) for different values of mesh-size divided
   // by wavelength

   double del = delta;
   ii = 0;
   double t=itime;
   for (k = 0;k<20;k++){
      efi[k] = efi[k] + ei*sin(2*pie*t*del);
      efr[k] = efr[k] + ei*cos(2*pie*t*del);
      hfi[k] = hfi[k] + hi*sin(2*pie*t*del);
      hfi[k] = hfr[k] + hi*cos(2*pie*t*del);
      out[k][ii] = del;
      del += 0.002;
   }

   // save the current pulse magnitude for nex iteration

   for (i=nzb-1;i<nze;i++)
      for(j=nxb-1;j<nxe;j++)
         for(k=0;k<nt;k++)
            vi[it-1][i][j][k];
   it = itime;

   if ((it == 100) || (it == 150) || (it == 200)){
      if (it == 100 ){
         ii = 1;
         outf << "@ " << it << " Iterations\n";
      }
      else if (it == 150){
         ii = 2;
         outf << "@ " << it << " Iterations\n";
      }
      else if (it == 200){
         ii = 3;
         outf << "@ " << it << " Iterations\n";
      }

/******************** Step 5 **********************/
/** Calculate Magnitude and argument of impedance  **/
/**************************************************/
      CComplex cef,chf;
      for (k = 0;k<20;k++){
         cef.Real = efr[k]; cef.Imag = efi[k];
         chf.Real = hfr[k]; chf.Imag = hfi[k];
         out[k][ii] = cef.mag()/chf.mag();
```

```
            out[k][ii+4] = (cef.ang() - chf.ang());
         }
      }
   }

   /********************* Step 6 **********************/
   /**       alculate exact value of impedance       **/
   /*************************************************/

   del = delta;
   double r2,rig,rr,r3;
   CComplex cnum,cdem;
   for (k = 0;k<20;k++){
      r2 = 1/transc(del);
      outf << r2 << endl;
      r3 = tan(21.0*r2*pie*del);
      cnum.Real = r2; cnum.Imag = r3;
      rig = r2*r3;
      rr = 1;
      cdem.Real = rr; cdem.Imag = rig;
      out[k][4] = cnum.mag()/(cdem.mag()*r2);
      out[k][8] = cnum.ang() - cdem.ang();
      del += .002;
   }

   for (k = 0;k<20;k++){
      for (j = 0;j<10;j++)
         outf << "out[" << k <<"][" << j << "]:\t" << out[k][j] << "\t";
      outf << endl;
   }

}
 // end of main
```

Listing 4: Fixed-Random Walk Monte Carlo Method:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream.h>
#include <fstream.h>


void main()
{
   // create an ouput file
   ofstream out("monte.out");
   if (!out)
   {
      cout << "Error with creating output file
      'monte.out.'\n"; exit(0);
   }

   /**********************************************/
   /**********     Input Parameters     ***********/
   /**********************************************/
   double v1 = 0, v2 = 0, v3 = 100, v4 = 0,v;
   double p1 = 0.25, p2 = 0.5, p3=0.75;

   int nrun = 10000;
   double delta = 0.05;

   // initializing
   double xo = 0.75, yo = 0.25, io = xo/delta, jo = yo/delta;
   double imax = 1/delta, jmax = 1/delta, sum = 0.0;
   int ms =0, m1 = 0, m2 = 0, m3 = 0, m4 = 0;
   // number of walks at each boundary
   double r;

      /* Seed the random-number generator */
      srand( (unsigned)time( NULL ) );


   /***********************************************/
   /***********  Begin Simulation *****************/
   /***********************************************/
   double k,i,j;
   int inside = true,ns;
   for (k=1;k<=nrun;k++){
      i = io;
      j = jo;
      while (inside){
         r = rand();
         r = (double)r/RAND_MAX;
         ns++;
         if (r >= 0 && r < p1)
            i++;
         else if( r >= p1 && r < p2)
            j++;
         else if (r >= p2 && r < p3)
            i--;
         else if(r >= p3)
            j--;
         // is the next step at the boundary?
         if (i == 0){
            sum += v4;
            m4++;
            inside = false;
         }
         else if (i > imax){
            sum += v2;
            m2++;
            inside = false;
         }
         else if (j == 0){
            sum += v1;
            m1++;
            inside = false;
         }
         else if (j > jmax){
            sum += v3;
            m3++;
            inside = false;
         }
      }

      inside = true;

   }
   v = (double)sum/nrun;
   out << "------------------------------------------" << endl;
   out << "------------------------------------------" << endl;
   out << "Point: " << xo << ", " << yo << "  Voltage:
   " << v << endl;
   out << "------------------------------------------" << endl;
   out << "Walk Distribution  (out of " << nrun << " total)
   " << endl;
   out << "------------------------------------------" << endl;
   out << "       1st Border: " << m1 << endl;
   out << "       2nd Border: " << m2 << endl;
   out << "       3rd Border: " << m3 << endl;
   out << "       4th Border: " << m4 << endl;
   out << "------------------------------------------" << endl;
   out << "------------------------------------------" << endl;
}
```