tlap.c

```c
/*
 *  tlap.c
 *
 *  This program reads in a physical description of the resonator system
 *  and constructs the capacitance matrix for the components.
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/types.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

/*
 *  Grid size to use, in R and Z directions.
 */

#define NCELR 2048
#define NCELZ 2048

#define default_Ncp 32

int NCP;

/*
 *  Scaling factor for X display.
 */

#define XSCALE 0.2  /* 0.4 */

#define SCRWIDTH ((int)(NCELR * XSCALE))
#define SCRHEIGHT ((int)(NCELZ * XSCALE))


int ground_plane = 0;  /*
                        *  Set non-zero if a ground plane is specified in the
                        *  input file.
                        */

int mvflag;  /*
              *  If set non-zero before a relaxation pass causes the pass to
              *  record a picture of which cells have moved. Used on the pass
              *  prior to a display update.
              */

unsigned char mvs[ NCELZ * NCELR];  /*
                                     *  Array to hold the movements picture.
                                     */

#define MV(R, Z)     (mvs[(Z)*NCELR+(R)])


float maxdim = 0.0;  /*
                      *  Set during input parsing, records the maximum
                      *  dimension from the origin of any physical component.
                      */

float SCALE = 0.0; /*
                    *  The radius in metres of the cylinder represented by
                    *  the grid. Established during input parsing.
                    */

#define EPSILON 8.854188e-12

FILE *fdi;

int round = 0;
double total_rxed_charge;

#define USE_SUBGRID 8  /*
                        *  Use the lazy grid optimisation at or below this
                        *  granularity. Set to zero to disable lazy grid.
                        */

float total_coil_length;


double *coil_charge;

char sysname[ 100];      /* The name of the resonator system being processed */

/*
 *  Solution grid cells. We generously use two grids ca and cb, and bounce
 *  the field from one to the other on each relaxation pass.
 */

typedef double CELLTYPE;

CELLTYPE *c1 = NULL, *c2 = NULL;                        /* Allocation handles */
CELLTYPE *cs = NULL, *cd = NULL;         /* Source and dest working pointers */

#define CELL( A, R, Z)  ((A)[(Z)*NCELR+(R)])
#define CELL_P( A, R, Z)  ((A) + (Z)*NCELR+(R))

#define TOGGLE_GRIDS() { CELLTYPE *t = cs; cs = cd; cd = t; }

/*
 *  Transformations between coordinates: world, grid, and screen.
 */

#define WORLD_TO_GRID_R( R)  ((int)(rbound/SCALE * (double)(R)))
#define WORLD_TO_GRID_Z( Z)  ((int)(zbound/SCALE * (double)(Z)))

#define GRID_TO_WORLD_R( R)  ((double)(SCALE/rbound * (double)(R)))
#define GRID_TO_WORLD_Z( Z)  ((double)(SCALE/zbound * (double)(Z)))

#define GRID_TO_SCREEN_X( R)  ((int)((R * SCRWIDTH)/rbound))
#define GRID_TO_SCREEN_Y( Z)  ((int)(SCRHEIGHT - (Z * SCRHEIGHT)/zbound))

/*
 *  Physical object tables.
 */

struct TOROID
{
   char name[ 50];
   float outer_radius;
   float inner_radius;
   float altitude;

   float Q;
}
 toroids[ 50];

int n_toroids = 0;

struct COIL
{
```

tlap.c


```
   char name[ 50];
   float radius;
   float altitude;
   float length;
}
 coils[ 50];

int n_coils = 0;

struct DISC
{
   char name[ 50];
   float inner_radius;
   float outer_radius;
   float altitude;
   float Q;
}
 discs[ 50];

int n_discs = 0;

#define FLAG(R, Z)      (flags[(Z)*NCELR+(R)])
unsigned char flags[ NCELZ * NCELR];

int gran, rbound, zbound;
float tot_earth, tot_sky;

CELLTYPE top_bound[ NCELR];
CELLTYPE right_bound[ NCELZ];

float init_scale;

void set_dimension( float f)
{
   if( f > maxdim) maxdim = f;

   for( init_scale = 0.1; init_scale < 1.5 * maxdim;
             init_scale += init_scale < 1.0 ? 0.1 : 1.0);
}

/******************************************************************************/
/*  X11 Display Stuff                                                         */
/******************************************************************************/

Display *display;
Window window, root_window;
Font ch_font;
GC gc;
XSizeHints hints;
int screen, depth;
XImage *xim = NULL;
Visual *visual;

XColor colors[ 16];
unsigned long black_pixel, white_pixel;
#define DC_BLACK  0
#define DC_WHITE  15

#define COLOR0 "black"
#define COLOR15 "white"

unsigned short *image;

int xerror_handler( Display *d, XErrorEvent *e)
{
   fprintf( stderr, "error in X connection\n");
   exit( 1);
}

int xferror_handler( Display *d)
{
   fprintf( stderr, "error in X connection\n");
   exit( 1);
}

void draw_window( void)
{
   XPutImage( display, window, gc, xim, 0, 0, 0, 0, SCRWIDTH, SCRHEIGHT);
   XSync( display, 0);
}


static void inline setpix( int x, int y, int r, int g, int b)
{
   int ix = (SCRWIDTH * x)/NCELR;
   int iy = SCRHEIGHT - (SCRHEIGHT * y)/NCELZ;

   if( r > 31) r = 31;
   if( g > 63) g = 63;
   if( b > 31) b = 31;

   image[ iy * SCRWIDTH + ix] = (r << 11) | (g << 5) | b;
}

void init_image( void)
{
   int x, y;

   for( y=0; y<NCELZ; y++)
      for( x=0; x<NCELR; x++) if( FLAG( x, y) == 1) setpix( x, y, 25, 0, 0);
}

void update_image( void)
{
   int r, z;
   int x1, y1;

   for( z=0; z<zbound; z++)
      for( r=0; r<rbound; r++)
      {
         if( !FLAG( r, z))
         {
            int v = (int)(30 * CELL( cs, r, z));

            for( x1 = r*gran; x1<(r+1)*gran; x1++)
               for( y1 = z*gran; y1<(z+1)*gran; y1++)

            if( MV( r, z)) setpix( x1, y1, 255, 255, 0);
            else setpix( x1, y1, v, 2*v, v);
         }
         else
         {
            for( x1 = r*gran; x1<(r+1)*gran; x1++)
               for( y1 = z*gran; y1<(z+1)*gran; y1++)

            setpix( x1, y1, 25, 0, 0);
         }
      }
}

void init_display( int argc, char *argv[])
{
```

tlap.c

```
   XColor exact;
   Colormap cmap;
   XEvent ev;

   if( (display = XOpenDisplay( "localhost:0")) == NULL)
   {
      printf( "cannot open display\n");
      exit( 1);
   }

   screen = DefaultScreen( display);
   root_window = DefaultRootWindow( display);

   XSetErrorHandler( xerror_handler);
   XSetIOErrorHandler( xferror_handler);

   visual = DefaultVisual( display, screen);
   depth = DefaultDepth( display, screen);

   cmap = DefaultColormap( display, screen);
   XAllocNamedColor( display, cmap, COLOR0, &exact, colors + DC_BLACK);
   XAllocNamedColor( display, cmap, COLOR15, &exact, colors + DC_WHITE);

   black_pixel = colors[ DC_BLACK].pixel;
   white_pixel = colors[ DC_WHITE].pixel;

   hints.x = hints.y = 5;
   hints.width = SCRWIDTH;
   hints.height = SCRHEIGHT;

   hints.flags = PPosition | PSize;

   window = XCreateSimpleWindow(
              display, root_window,
              0, 0, hints.width, hints.height, 2,
              white_pixel,  black_pixel
             );

   XSetStandardProperties( display, window,
                           "Laplace", "LP",
                           None, argv, argc, &hints );

   gc = XCreateGC( display, window, 0L, 0 );
   XSetForeground( display, gc, white_pixel);
   XSetBackground( display, gc, black_pixel);

   XSelectInput( display, window, ExposureMask );
   XMapRaised( display, window );

   if((image = malloc( SCRWIDTH * SCRHEIGHT * sizeof( unsigned short))) == 0)
   {
      perror( "cannot malloc image");
      exit( 1);
   }

   xim = XCreateImage( display, visual, depth, ZPixmap, 0,
          NULL, SCRWIDTH, SCRHEIGHT, depth, 0);

   xim->data = (char *) image;

   do
   {
      XNextEvent( display, &ev );
   }
    while( ev.type != Expose );

   XSelectInput( display, window,
                 StructureNotifyMask | KeyPressMask | KeyReleaseMask |
                 ExposureMask |
                 EnterWindowMask | LeaveWindowMask | ButtonPressMask);

   XClearWindow( display, window );

   XSync( display, 1);
   draw_window();
}

void service_display( void)
{
   XEvent ev;
   while( XPending( display))
   {
      XNextEvent( display, &ev);

      switch( ev.type )
      {
         case DestroyNotify:

              exit( 0);
              break;

         case Expose :

               if( ev.xexpose.count == 0) draw_window();

         default : break;
      }
   }
}

/****************************************************************************/
/* Log File Routines                                                        */
/****************************************************************************/

void logmsg( char *format, ...)
{
   FILE *fl;
   char temp[ 500];

   sprintf( temp, "%s.log", sysname);
   if( (fl = fopen( temp, "a+")) == NULL) return;

   vsprintf( temp, format, &format + 1);

   fprintf( fl, "%s\n", temp);
   fclose( fl);
}

/****************************************************************************/
/* Input File Routines                                                      */
/****************************************************************************/

char pbuff[ 500], *pb;
int lino;
int eof = 0;

int parse_reload( void)
{
   *(pb = pbuff) = 0;

   if( fgets( pbuff, 499, fdi)) { lino++; return 1; }
```

tlap.c

```
   eof = 1;
   return 0;
}

void parse_error( char *msg)
{
   fprintf( stderr, "error in input file: line %d: %s\n", lino, msg);
   exit( 1);
}

void skip_whitespace( void)
{
   while( 1)
   {
      if( !*pb)
         if( !parse_reload()) return;
         else continue;

      switch( *pb)
      {
         case 32:
         case '\t':
         case '\n':
         case '\r': pb++; continue;

         case ';': *pb = 0; continue;

         default: return;
      }
   }
}

int read_word( char *s)
{
   int qflag = 0;
   int n = 0;

   skip_whitespace();

   while( *pb || qflag)
   {
      if( qflag)
      {
         if( !*pb || *pb == '\r' || *pb == '\n')
            parse_error( "unterminated string");
         if( *pb == '"')
         {
            pb++;
            break;
         }
      }
      else
      {
         if( *pb == 32 || *pb == '\t' || *pb == '\n' || *pb == '\r') break;
         if( !n && *pb == '"') { pb++; qflag = 1; continue; }
      }

      *s++ = *pb++; n++;
   }

   *s = 0;
   return n;
}

int parse_word( char *target)
{
   int n = strlen( target);

   skip_whitespace();
   if( strncasecmp( pb, target, n)) return 0;

   pb += n;
   return 1;
}

int parse_toroid( void)
{
   struct TOROID *t = toroids + n_toroids++;
   char temp[ 100];

   if( !parse_word( "{")) parse_error( "missing { after TOROID");

   while( !parse_word( "}"))
   {
      if( eof) return 0;

      if( parse_word( "name"))
      {
         read_word( t->name);
         printf( "toroid name [%s]\n", t->name);
      }
      else
      if( parse_word( "outer_radius"))
      {
         read_word( temp);
         t->outer_radius = atof( temp);
      }
      else
      if( parse_word( "inner_radius"))
      {
         read_word( temp);
         t->inner_radius = atof( temp);
      }
      else
      if( parse_word( "altitude"))
      {
         read_word( temp);
         t->altitude = atof( temp);
      }
      else parse_error( "unrecognised TOROID parameter");
   }

   t->Q = 0.0;
   set_dimension( t->altitude + (t->outer_radius - t->inner_radius)/2);
   set_dimension( t->outer_radius);

   return 1;
}

int parse_disc( void)
{
   struct DISC *t = discs + n_discs++;
   char temp[ 100];

   if( !parse_word( "{")) parse_error( "missing { after DISC");

   while( !parse_word( "}"))
   {
      if( eof) return 0;

      if( parse_word( "name"))
      {
```

tlap.c

```c
         read_word( t->name);
         printf( "disc name [%s]\n", t->name);
      }
      else
      if( parse_word( "outer_radius"))
      {
         read_word( temp);
         t->outer_radius = atof( temp);
      }
      else
      if( parse_word( "inner_radius"))
      {
         read_word( temp);
         t->inner_radius = atof( temp);
      }
      else
      if( parse_word( "altitude"))
      {
         read_word( temp);
         t->altitude = atof( temp);
      }
      else parse_error( "unrecognised DISC parameter");
   }

   t->Q = 0.0;
   set_dimension( t->altitude);
   set_dimension( t->outer_radius);

   return 1;
}

int parse_coil( void)
{
   struct COIL *c = coils + n_coils++;
   char temp[ 100];

   if( !parse_word( "{")) parse_error( "missing { after COIL");

   while( !parse_word( "}"))
   {
      if( eof) return 0;

      if( parse_word( "name"))
      {
         read_word( c->name);
         printf( "coil name [%s]\n", c->name);
      }
      else
      if( parse_word( "radius"))
      {
         read_word( temp);
         c->radius = atof( temp);
      }
      else
      if( parse_word( "length"))
      {
         read_word( temp);
         c->length = atof( temp);
      }
      else
      if( parse_word( "altitude"))
      {
         read_word( temp);
         c->altitude = atof( temp);
      }
      else
      if( parse_word( "inductance") ||
          parse_word( "turns") ||
          parse_word( "resistance") ||
          parse_word( "conductor_radius"))
      {
         read_word( temp);
      }
      else
      if( !parse_word( "use_lundin"))
         parse_error( "unrecognised COIL parameter");
   }

   set_dimension( c->altitude + c->length);
   set_dimension( c->radius);

   return 1;
}

int parse_ground_plane( void)
{
   if( !parse_word( "{") || !parse_word( "}"))
      parse_error( "missing {} after GROUND_PLANE");

   ground_plane = 1;
   return 1;
}

int parse_begin( void)
{
   if( !parse_word( "BEGIN"))
   {
      fprintf( stderr, "missing BEGIN\n");
      return 0;
   }

   while( 1)
   {
      char temp[ 50];

      if( eof) { parse_error( "unexpected end of file"); return 0; }
      if( parse_word( "END")) return 1;
      else
      if( parse_word( "NCP")){ read_word( temp); NCP = atoi( temp); }
      else
      if( parse_word( "TOROID") && !parse_toroid()) return 0;
      else
      if( parse_word( "DISC") && !parse_disc()) return 0;
      else
      if( parse_word( "COIL") && !parse_coil()) return 0;
      else
      if( parse_word( "GROUND_PLANE") && !parse_ground_plane()) return 0;
   }

   return parse_word( "END");
}

int cmp_coil( const void *a, const void *b)
{
   if( ((struct COIL *)a)->altitude < ((struct COIL *)b)->altitude) return -1;
   if( ((struct COIL *)a)->altitude > ((struct COIL *)b)->altitude) return 1;
   return 0;
}

void parse_input_file( void)
{
   char temp[ 100];
```

tlap.c

```
   pbuff[ 0] = 0; pb = pbuff;
   eof = lino = 0;

   NCP = 0;

   sprintf( temp, "%s.in", sysname);
   if( (fdi = fopen( temp, "r")) == NULL)
   {
      fprintf( stderr, "cannot open input file %s\n", temp);
      perror( "");
      exit( 1);
   }

   if( !parse_begin()) exit( 1);

   fclose( fdi);

   /*
    *  Sort coil sections into order of height, to put the lowest first
    *  int the coils[] array.
    */

   qsort( coils, n_coils, sizeof( struct COIL), cmp_coil);

   if( !NCP) NCP = default_Ncp;

   logmsg( "NCELR %d NCELZ %d", NCELR, NCELZ);
   logmsg( "Ncp %d ground plane %s", NCP, ground_plane ? "yes" : "no");
   coil_charge = malloc( sizeof( double) * NCP);
}

/*****************************************************************************/
/*  Laplace Solver                                                           */
/*****************************************************************************/

void init_cells( void)
{
   int i;

   if( !c1) c1 = malloc( sizeof( CELLTYPE) * NCELR * NCELZ);
   if( !c2) c2 = malloc( sizeof( CELLTYPE) * NCELR * NCELZ);

   if( !c1 || !c2)
   {
      perror( "cannot allocate enough memory");
      exit( 1);
   }

   for( i=0; i<NCELZ*NCELR; i++) c1[ i] = 0;

   cs = c1;
   cd = c2;
}

void install_world( int cno, int tno, int dno)
{
   int r, z, i, k;
   float tl;

   memset( flags, 0, sizeof( flags));

   /*
    *  Install ground plane if required.
    */

   if( ground_plane)
      for( r=0; r<rbound; r++)
      {
         FLAG( r, 0) = 1;
         CELL( cs, r, 0) = 0;
      }

   /*
    *  Simulate infinity along top and right edges. First time round the
    *  boundaries are set at zero potential. Subsequent rounds after boundary
    *  contraction they are set to the boundary potential profiles extracted
    *  from the previous round.
    *
    */

   if( round == 0)
   {
      for( z=zbound-1, r=0; r<rbound; r++)
      {
         FLAG( r, z) = 1;
         CELL( cs, r, z) = 0;
      }

      for( r=rbound-1, z=0; z<zbound; z++)
      {
         FLAG( r, z) = 1;
         CELL( cs, r, z) = 0;
      }
   }
   else
   {
      for( z=zbound-1, r=0; r<rbound; r++)
      {
         FLAG( r, z) = 1;
         CELL( cs, r, z) = top_bound[ r * gran];
      }

      for( r=rbound-1, z=0; z<zbound; z++)
      {
         FLAG( r, z) = 1;
         CELL( cs, r, z) = right_bound[ z * gran];
      }
   }

   /*
    *  Install toroids.
    */

   for( k=0; k<n_toroids; k++)
   {
      float w = (toroids[ k].outer_radius + toroids[ k].inner_radius)/2;
      float rd = (toroids[ k].outer_radius - toroids[ k].inner_radius)/2;

      int th = WORLD_TO_GRID_R( w);
      int tv = WORLD_TO_GRID_Z( toroids[ k].altitude);
      int tr = WORLD_TO_GRID_R( rd);

      for( r=th-tr; r<=th+tr; r++)
         for( z=tv-tr; z<= tv+tr; z++)
            if( (r-th)*(r-th) + (z-tv)*(z-tv) < tr*tr)
            {
               CELLTYPE v = (CELLTYPE)(tno == k ? 1.0 : 0.0);

/*               v /= gran * gran; */

               if( FLAG( r, z))
               {
```

tlap.c

```
                  if( gran == 1)
                     fprintf( stderr, "toroid %s overlaps something else\n",
                            toroids[ k].name);
                  CELL( cs, r, z) += v;
               }
               else
               {
                  CELL( cs, r, z) = v;
                  FLAG( r, z) = 1;
               }
            }
   }

   /*
    *  Install coils.
    */

   for( total_coil_length = 0.0, k=0; k<n_coils; k++)
      total_coil_length += coils[ k].length;

   tl = 0.0;
   for( k=0; k<n_coils; tl += coils[ k].length, k++)
   {
      int z1 = WORLD_TO_GRID_Z( coils[ k].altitude);
      int z2 = WORLD_TO_GRID_Z( coils[ k].length);
      r = WORLD_TO_GRID_R( coils[ k].radius);

      for( z=z1; z<z1+z2; z++)
      {
         int n = (int)(NCP * (GRID_TO_WORLD_Z( z - z1) + tl)/total_coil_length);

         CELLTYPE v = (CELLTYPE)(cno == n ? 1.0 : 0.0);

  /*      v /= gran; */

         if( FLAG( r, z))
         {
            if( gran == 1)
               fprintf( stderr, "coil %s overlaps something else\n",
                            coils[ k].name);
            CELL( cs, r, z) += v;
         }
         else
         {
            CELL( cs, r, z) = v;
            FLAG( r, z) = 1;
         }
      }
   }

   /*
    *  Install discs.
    */

   for( k=0; k<n_discs; k++)
   {
      int r1 = WORLD_TO_GRID_R( discs[ k].inner_radius);
      int r2 = WORLD_TO_GRID_R( discs[ k].outer_radius);
      z = WORLD_TO_GRID_Z( discs[ k].altitude);

      for( r=r1; r <= r2; r++)
      {
         CELLTYPE v = (CELLTYPE)(dno == k ? 1.0 : 0.0);

/*         v /= gran * gran; */

         if( FLAG( r, z))
         {
            if( gran == 1)
               fprintf( stderr, "disc %s overlaps something else\n",
                            discs[ k].name);
            CELL( cs, r, z) += v;
         }
         else
         {
            CELL( cs, r, z) = v;
            FLAG( r, z) = 1;
         }
      }
   }

   for( i=0; i<NCELZ*NCELR; i++)
      if( flags[ i]) cd[ i] = cs[ i];
}

/*
 *  The relaxation employs subset solving for a big improvement in performance.
 *  When subset_flag is zero, a normal relaxation pass is performed, during which
 *  the subgrid matrix is initialised with a map of active regions of the main
 *  grid, ie those regions still containing moving grid values. Then, for
 *  subsequent passes, when subgrid_flag != 0, only the active regions listed
 *  in subgrid_map are processed. subgrid_flag is incremented on each pass, and
 *  set to zero if SUBSOLVE_MAX is reached to force a rescan of the whole grid.
 *  If a subgrid pass produces no movement in a region, that region is
 *  deselected in subgrid_map. When no active regions remain, subgrid_flag is
 *  set to zero to force a full pass.
 */

#define SUBSOLVE_MAX 50

unsigned char subgrid_map[ NCELR/16][ NCELZ/16];

void clear_subgrid( void)
{
   memset( subgrid_map, 0, sizeof( subgrid_map));
}

int subset_flag;
#define SUBGRID( r, z) (subgrid_map[ (r)/16][ (z)/16])

int moves;
int skips;

double move_threshold;

static void inline update_cell( int r, int z)
{
   register CELLTYPE *p;
   register CELLTYPE f;

   /*
    *  Apply the finite difference relations appropriate for a
    *  pseudo-2D space in cylindrical coordinates.
    */

   p = CELL_P( cs, r, z);
   f = (*(p+1) + *(p-1) + *(p+NCELR) + *(p-NCELR))/4 +
      (*(p+1) - *(p-1))/(8*r);
/*
   CELLTYPE f = CELL( cs, r-1, z) + CELL( cs, r+1, z) +
                CELL( cs, r, z+1) + CELL( cs, r, z-1);
```

tlap.c

```
   f /= 4;
   f += (CELL( cs, r+1, z) - CELL( cs, r-1, z))/(8*r);
*/

   CELL( cd, r, z) = f;

   /*
    *  Check if movement has occured, and count the number of moves.
    */

   {
      CELLTYPE g;

      if( (g = CELL( cs, r, z)) != 0)
      {
         g = fabs( f - g)/g;
         if( g > move_threshold)
         {
            moves++;
            if( mvflag) MV( r, z)=1;
            SUBGRID( r, z) = 1;
         }
      }
   }
}

int relax_field( int nc)
{
   int r, z;

   moves = 0;
   if( mvflag) memset( mvs, 0, sizeof( mvs));

   /*
    *  Reset the subset_flag if necessary. Clear the subgrid_map prior to
    *  a full scan.
    */

   if( nc == 0 ||                      /* At the start of a relaxation phase */
       gran > USE_SUBGRID ||        /* When granularity is above USE_SUBGRID */
       subset_flag >= SUBSOLVE_MAX       /* or after SUBSOLVE_MAX scans done */
     )
       subset_flag = 0;

   if( !subset_flag && gran <= USE_SUBGRID) clear_subgrid();
   skips = 0;

   /*
    *  Impose Neumann boundary condition on left edge and bottom edge,
    *  excluding any cells that are within conductors.
    */

   for( z=1; z<zbound-1; z++)
      if( !FLAG( 0, z)) CELL( cd, 0, z) = CELL( cs, 1, z);

   for( r=1; r<rbound-1; r++)
      if( !FLAG( r, 0)) CELL( cd, r, 0) = CELL( cs, r, 1);

   /*
    *  Setup the movements threshold. If a cell value changes by more than this
    *  factor, it is counted as active.
    */

   move_threshold = 0.0001 /(gran*gran);

   if( !subset_flag)
   {
      for( z=1; z<zbound-1; z++)
         for( r=1; r<rbound-1; r++) if( !FLAG( r, z)) update_cell( r, z);

      if( gran <= USE_SUBGRID)
         for( z=1; z<zbound-1; z++)
            for( r=1; r<rbound-1; r++)
               if( !SUBGRID( r, z)) CELL( cs, r, z) = CELL( cd, r, z);

      /*
       *  The relaxation is complete when there are no cell value movements
       *  following a full scan of the array.
       */

      if( !moves) return 1;
   }
   else
   {
      int r1, z1;

      /*
       *  Do a partial scan, using only the regions in which subgrid_map != 0.
       */

      for( z1=0; z1<zbound/16; z1++)
         for( r1=0; r1<rbound/16; r1++)
         {
            if( !subgrid_map[ r1][ z1])
            {
                skips += 16*16;
                continue;
            }
            else
            {
               subgrid_map[ r1][ z1] = 0;
               for( z=z1*16; z<(z1+1)*16; z++)
                  for( r=r1*16; r<(r1+1)*16; r++)
                     if( r && z && !FLAG( r, z)) update_cell( r, z);
            }
         }

      /*
       *  If no movements have occured during a partial scan, invoke a full
       *  scan on the next cycle by setting subset_flag to zero.
       */

      if( !moves)
      {
         subset_flag = 0;
         return 0;
      }
   }

   subset_flag++;

   return 0;
}

void expand_cells( void)
{
   int r, z;

   printf( "expand to %d\n", gran/2);

   for( r=0; r<rbound; r++)
      for( z=0; z<zbound; z++)
```

tlap.c

```
      {
         CELLTYPE v = CELL( cs, r, z);

         CELL( cd, 2*r+0, 2*z+0) = v;
         CELL( cd, 2*r+1, 2*z+0) = v;
         CELL( cd, 2*r+0, 2*z+1) = v;
         CELL( cd, 2*r+1, 2*z+1) = v;
      }

   TOGGLE_GRIDS();
}

void save_boundaries( void)
{
   int r, z, k;

   for( z = (zbound-1)/2, r = 0; r < rbound/2; r++)
   {
      CELLTYPE q = CELL( cs, r, z);
      for( k = 0; k < gran*2; k++) top_bound[ 2*gran*r + k] = q;
   }

   for( r = (rbound-1)/2, z = 0; z < zbound/2; z++)
   {
      CELLTYPE q = CELL( cs, r, z);
      for( k=0; k < gran*2; k++) right_bound[ 2*gran*z + k] = q;
   }
}


void sum_field( int r, int z)
{
   float flux;
   float a, v, tl;
   float field = 0;
   int k;

   v = CELL( cs, r, z);

   if( z > 0 && !FLAG( r, z-1)) field += CELL( cs, r, z-1) - v;
   if( z < zbound-1 && !FLAG( r, z+1)) field += CELL( cs, r, z+1) - v;
   if( r > 0 && !FLAG( r-1, z)) field += CELL( cs, r-1, z) - v;
   if( r < rbound-1 && !FLAG( r+1, z)) field += CELL( cs, r+1, z) - v;

   if( field == 0.0) return;

   a = gran * SCALE/NCELR;
   flux = 2 * M_PI * r * a * field;

   /*
    *  Find which object holds this charge.
    */

   for( k=0; k<n_toroids; k++)
   {
      int th = WORLD_TO_GRID_R(
                 (toroids[ k].outer_radius + toroids[ k].inner_radius)/2.0);
      int tv = WORLD_TO_GRID_Z( toroids[ k].altitude);
      int tr = WORLD_TO_GRID_R(
                 (toroids[ k].outer_radius - toroids[ k].inner_radius)/2.0);

      if( z >= tv - tr-1 && z <= tv + tr + 1 &&
          r >= th - tr-1 && r <= th + tr + 1)
      {
         toroids[ k].Q += flux * EPSILON;
         return;
      }
   }

   tl = 0.0;
   for( k=0; k<n_coils; tl += coils[ k].length, k++)
   {
      int ch = WORLD_TO_GRID_R( coils[ k].radius);
      int z1 = WORLD_TO_GRID_Z( coils[ k].altitude);
      int z2 = WORLD_TO_GRID_Z( coils[ k].length);

      if( r == ch &&
          z >= z1 && z < z1+z2)
      {
         int n = (int)(NCP * (GRID_TO_WORLD_Z( z - z1) + tl)/total_coil_length);
         coil_charge[ n] += flux * EPSILON;
         return;
      }
   }

   for( k=0; k<n_discs; k++)
      if( z == WORLD_TO_GRID_Z( discs[ k].altitude) &&
          r >= WORLD_TO_GRID_R( discs[ k].inner_radius) &&
          r <= WORLD_TO_GRID_R( discs[ k].outer_radius))
      {
         discs[ k].Q += flux * EPSILON;
         return;
      }

   /*
    *  Any remaining conductors must be part of the sky or ground boundaries.
    */

   if( z == 0) tot_earth += flux * EPSILON;
   else
   if( z == zbound - 1 ||
       r == rbound - 1) tot_sky += flux * EPSILON;
   else
   {
      /*
       *  There must be a programming error to reach here.
       */

      fprintf( stderr, "error, cannot find home for charge at %d %d!\n", r, z);
      exit( 1);
   }
}

/***************************************************************************/
/*  Capacitance Computations                                               */
/***************************************************************************/

/*
 *  Compute total charge on each conductor.
 */

void sum_charges( int cno, int tno, int dno)
{
   int k, r, z;

   tot_earth = tot_sky = 0.0;
   for( k=0; k<n_toroids; k++) toroids[ k].Q = 0.0;
   for( k=0; k<n_discs; k++) discs[ k].Q = 0.0;
   for( k=0; k<NCP; k++) coil_charge[ k] = 0.0;

   for( r=0; r<rbound; r++)
      for( z=0; z<zbound; z++)
```

tlap.c

```
      if( FLAG( r, z)) sum_field( r, z);

   logmsg( "Q earth=%.3f sky=%.3f", tot_earth * 1e12, tot_sky* 1e12);

   for( k=0; k<n_toroids; k++)
      logmsg( "Q toroid %d=%.5f", k, toroids[ k].Q * 1e12);

   for( k=0; k<NCP; k++)
      logmsg( "Q coil %d=%.6f", k, coil_charge[ k] * 1e12);

   for( k=0; k<n_discs; k++)
      logmsg( "Q disc %d=%.5f", k, discs[ k].Q * 1e12);

   total_rxed_charge = 0.0;
   for( k=0; k<NCP; k++) if( k != cno) total_rxed_charge += coil_charge[ k];
   for( k=0; k<n_toroids; k++) if( k != tno) total_rxed_charge += toroids[ k].Q;
   for( k=0; k<n_discs; k++) if( k != dno) total_rxed_charge += discs[ k].Q;
   total_rxed_charge += tot_earth + tot_sky;

   logmsg( "total destination charge %.4f", total_rxed_charge * 1e12);
}

FILE *open_caps_file( void)
{
   char temp[ 100];
   FILE *fh;

   sprintf( temp, "%s.caps", sysname);

   if( (fh = fopen( temp, "a+")) == NULL)
   {
      fprintf( stderr, "cannot open %s", temp);
      perror( "");
      exit( 1);
   }

   return fh;
}

void derive_coil_caps( int cno)
{
   FILE *fh = open_caps_file();
   int k;
   float C;
   double Q1, Q2, QX, V1, V2, Qratio;

   /*
    *  Compute caps.
    */

   Q1 = coil_charge[ cno];
   V1 = 1.0;
   logmsg( "source charge %.4f", Q1 * 1e12);

   Q2 = total_rxed_charge;
   V2 = 0.0;

   QX = (Q2 - Q1)/2;
   Qratio = QX/Q2;
   logmsg( "mean charge %.4f, Q adjustment ratio %.3f", QX * 1e12, Qratio);

   C = (tot_earth + tot_sky)/(V1-V2) * Qratio;
   fprintf( fh, "external coil %d %.4f\n", cno+1, C * 1e12);

   for( k=0; k<NCP; k++)
   {
      if( k == cno) continue;

      C = coil_charge[ k]/(V1-V2) * Qratio;
      fprintf( fh, "internal coil %d to coil %d %.4f\n",
                cno+1, k+1, C * 1e12);
   }

   for( k=0; k<n_toroids; k++)
   {
      C = toroids[ k].Q/(V1-V2) * Qratio;
      fprintf( fh, "internal coil %d to toroid %d %.4f\n",
                cno+1, k+1, C * 1e12);
   }

   for( k=0; k<n_discs; k++)
   {
      C = discs[ k].Q/(V1-V2) * Qratio;
      fprintf( fh, "internal coil %d to disc %d %.4f\n",
                cno+1, k+1, C * 1e12);
   }

   fclose( fh);
}

void derive_toroid_caps( int tno)
{
   FILE *fh = open_caps_file();
   float C;
   int k;
   double Q1, Q2, QX, V1, V2, Qratio;

   Q1 = toroids[ tno].Q;
   V1 = 1.0;
   logmsg( "source charge %.4f", Q1 * 1e12);

   Q2 = total_rxed_charge;
   V2 = 0.0;

   QX = (Q2 - Q1)/2;
   Qratio = QX/Q2;
   logmsg( "mean charge %.4f, Q adjustment ratio %.3f", QX * 1e12, Qratio);

   C = (tot_earth + tot_sky)/(V1-V2) * Qratio;
   fprintf( fh, "external toroid %d %.4f\n", tno+1, C * 1e12);

   for( k=0; k<NCP; k++)
   {
      C = coil_charge[ k]/(V1-V2) * Qratio;
      fprintf( fh, "internal toroid %d to coil %d %.4fpF\n",
           tno+1, k+1, C * 1e12);
   }

   for( k=0; k<n_discs; k++)
   {
      C = discs[ k].Q/(V1-V2) * Qratio;
      fprintf( fh, "internal toroid %d to disc %d %.4f\n",
                tno+1, k+1, C * 1e12);
   }

   fclose( fh);
}

void derive_disc_caps( int dno)
{
   FILE *fh = open_caps_file();
   float C;
```

tlap.c


```
   int k;
   double Q1, Q2, QX, V1, V2, Qratio;

   Q1 = discs[ dno].Q;
   V1 = 1.0;
   logmsg( "source charge %.4f", Q1 * 1e12);

   Q2 = total_rxed_charge;
   V2 = 0.0;

   QX = (Q2 - Q1)/2;
   Qratio = QX/Q2;
   logmsg( "mean charge %.4f, Q adjustment ratio %.3f", QX * 1e12, Qratio);

   C = (tot_earth + tot_sky)/(V1-V2) * Qratio;
   fprintf( fh, "external disc %d %.4f\n", dno+1, C * 1e12);

   if( n_coils)
      for( k=0; k<NCP; k++)
      {
         C = coil_charge[ k]/(V1-V2) * Qratio;
         fprintf( fh, "internal disc %d to coil %d %.4fpF\n",
              dno+1, k+1, C * 1e12);
      }

   for( k=0; k<n_toroids; k++)
   {
      C = toroids[ k].Q/(V1-V2) * Qratio;
      fprintf( fh, "internal disc %d to toroid %d %.4f\n",
                dno+1, k+1, C * 1e12);
   }

   for( k=0; k<n_discs; k++)
   {
      if( k == dno) continue;

      C = discs[ k].Q/(V1-V2) * Qratio;
      fprintf( fh, "internal disc %d to disc %d %.4f\n",
                dno+1, k+1, C * 1e12);
   }

   fclose( fh);
}

void monitor_point( void)
{
   int r = WORLD_TO_GRID_R( 2.0);
   int z = WORLD_TO_GRID_Z( 2.5);

   double v = CELL( cs, r, z);

   logmsg( "monitor gran=%d r=%d z=%d v=%.8f", gran, r, z, v);
}

void solve_laplace( int cno, int tno, int dno)
{
   int nc;                                          /* Relaxation cycle counter */
   int initgran = 128;
   time_t t1, tstart;

   init_cells();

   SCALE = init_scale * 2;

   logmsg( "initial scale=%.1f cno=%d tno=%d dno=%d", SCALE, cno, tno, dno);

   gran = initgran; rbound = NCELR/gran; zbound = NCELZ/gran;
   nc = 0; round = 0;

   install_world( cno, tno, dno);

   /*
    *  Compute the potential distribution.
    */

   time( &tstart); t1 = tstart;

   while( 1)
   {
      int done;
      service_display();

      /*
       *  If the display update timer has expired, set the mvflag and reset
       *  the timer.
       */

      if( (mvflag = (time( NULL) - t1) > 10) != 0) time( &t1);

      /*
       *  Run a relaxation pass over the grid. If mvflag is set it will also
       *  record for the X display a picture of which areas are moving.
       */

      done = relax_field( nc);
      TOGGLE_GRIDS();

      printf( "%d scale %.1f gran %d moves %d skips %d\n",
               nc++, SCALE, gran, moves, skips);

      /*
       *  Update the X display if either mvflag is set, or we have completed
       *  this phase of relaxation.
       */

      if( mvflag || done)
      {
        update_image();
        draw_window();
      }

      if( !done) continue;  /*
                             *  Cell values are still changing, so continue
                             *  relaxation.
                             */

      /*
       *  Relaxation is completed for this granularity and SCALE.
       */

      logmsg( "scale=%.1f gran=%d cycles=%d elapsed=%d",
            SCALE, gran, nc, time( NULL) - tstart);

      if( SCALE == init_scale && gran == 1) break; /*
                                                    *  The field is completed.
                                                    */

      /*
       *  Select granularity and SCALE for the next phase of relaxation.
       */

      if( gran > 16 || SCALE == init_scale)
```

tlap.c


```
      {
         /*
          *  Retain the same SCALE and double the resolution by halving the
          *  granularity. Expand the grid by a factor of two.
          */

         expand_cells();
         gran /= 2; rbound = NCELR/gran; zbound = NCELZ/gran;
      }
      else
      {
         /*
          *  Do a boundary contraction, freezing the field values on
          *  the new boundary, and discarding the outer and upper halves.
          *  We halve the SCALE and double the granularity, so the grid cells
          *  stay put.
          */

         save_boundaries();
         round++;
         gran *= 2; rbound = NCELR/gran; zbound = NCELZ/gran;
         SCALE /= 2;
      }

      /*
       *  Install the boundaries and electrodes. Update the display to show
       *  the new settings.
       */

      install_world( cno, tno, dno);

      update_image();
      draw_window();

      /*
       *  Reset counter and timer for the next phase.
       */

      nc=0;
      time( &tstart);
   }
}

void usage( void)
{
   fprintf( stderr, "tlap: usage: ./tlap system\n");
   fprintf( stderr, "where 'system' is the name of the tesla resonator\n");
}

int main( int argc, char *argv[])
{
   int cno, tno, dno;

   if( argc < 2)
   {
      usage();
      exit( 1);
   }

   strcpy( sysname, argv[ argc-1]);
   parse_input_file();

   init_display( argc, argv);

   /*
    *  Run the laplace solver once for every electrode or coil section, with
    *  only the selected electrode containing a potential.
    */

   for( tno=0; tno < n_toroids; tno++)
   {
      solve_laplace( -1, tno, -1);
      sum_charges( -1, tno, -1);
      derive_toroid_caps( tno);
   }

   for( dno=0; dno < n_discs; dno++)
   {
      solve_laplace( -1, -1, dno);
      sum_charges( -1, -1, dno);
      derive_disc_caps( dno);
   }

   if( n_coils)
      for( cno=0; cno < NCP; cno++)
      {
         solve_laplace( cno, -1, -1);
         sum_charges( cno, -1, -1);
         derive_coil_caps( cno);
      }

   printf( "tlap completed\n");
   return 0;
}
```