tsim.c

```c
/*
 *  tsim.c - Tesla resonator simulation program.
 *
 *  author: paul@abelian.demon.co.uk
 *
 *  See http://www.abelian.demon.co.uk/tssp/ for the latest version of
 *  this software.
 *
 *  Equation numbers and notes refer to items in the accompanying tsim-notes.ps
 *
 *  Support for complex number arithmetic is provided by complex.{c,h} which
 *  are included in the distribution.
 *
 *  All the dull stuff, like parsing input files, is in tsim-util.c
 */

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

#include "complex.h"

/*
 *  Definitions concerning the simulator.
 */

#define VERSION "v0.2a, 24th August 2000"
#define S 1                                     /* Number of sections per turn */
#define Vin ((COMPLEX){ 1.0, 0.0})                  /* Input reference voltage */

double Dp = 0.5e-4;            /* Loss factor of shunt capacitance to ground */
double W = 0.5;                         /* Extra coefficient of winding loss */

/*
 *  Description of the test resonator. These values are set from the resonator
 *  description file, so the initialisations here are the defaults.
 */

char sysname[ 50];                           /* System name from command line */

int use_lundin = 0; /*
                     *  True if the computed inductance is to be used instead
                     *  of the given measured value.
                     */

int use_palermo = 0; /*
                      *  True if the Palermo inter-turn capacitance is to be
                      *  included.
                      */

int Nt = 0;                                     /* Number of turns in the coil */
double Lcoil = 0.0;         /* Total inductance of coil at low frequency - H */
double RcoilDC = 0.0;          /* Total DC resistance of coil winding - Ohms */
double Wrad = 0.0;                        /* Radius of coil conductor - metres */
double spacing_ratio = 0.0;                        /* Spacing ratio of winding */
double Hcoil = 0.0;                           /* Total length of coil - metres */
double Crad = 0.0;                                  /* Radius of coil - metres */
double Ctop = 0.0;                              /* Capacitance of top loading */

int Ncp = 0;                               /* Number of capacitance segments */
double Cext[ 100];                        /* Matrix of external capacitances */
double Ctor[ 100];                  /* Matrix of toroid to coil capacitances */
double Cint[ 100][ 100];                  /* Matrix of internal capacitances */

int Ns = 0;                      /* Total number of elementary LCR sections */

/*
 *  Some physical constants.
 */

#define MU (4*3.141*1.0E-7)
#define EPSILON 8.854188e-12

COMPLEX *I, *V;   /* Complex arrays to hold the voltage and current profiles */
double w;                    /* Angular frequency currently being solved for */

/*
 *  Prototypes of utility functions defined in tsim-util.c
 */

void load_system_description( void);
void logmsg( char *format, ...);
void *safe_malloc( int size);

/*****************************************************************************/
/*  Mutual Inductance                                                        */
/*****************************************************************************/

/*
 *  Estimate the mutual inductance between the coil sections i and j,
 *  using equ 4.5.
 */

double mdata[] =
{
   0.0,
   0.008297, 0.004941, 0.003224, 0.0021502, 0.0014186,
   0.0008990, 0.0005251, 0.00025998, 0.00008438
};

double filament( int i, int j, double s)
{
   int kx;
   double xd, K, fk, M, g1, g2;

   /*
    *  Return the self inductance if i and j are the same section.
    */

   if( i == j) return MU * Crad * ( log(8*Crad/Wrad) - 2) + (MU * Crad/4.0);

   xd = ((i-j) * Hcoil)/Nt;

   K = xd * xd / (2*Crad*Crad + xd*xd);
   kx = (int)(K*10);

   if( kx < 1)
   {
      fk = 0.014468 * ( log10(1/K)-0.53307);
   }
   else
   if( kx >= 9)
   {
      fk = pow( 10.0, -3.39224 + 1.5 * log10(1-K));
   }
   else
   {
      g1 = mdata[ kx];   g2 = mdata[ kx+1];
      fk = (K-(kx/10.0))/0.1 * (g2-g1) + g1;
   }

   M = 1e3 * MU/(4*M_PI) * Crad * fk;
```

tsim.c

```
/* printf( "(%d,%d)  K=%.3f fk = %.6f M=%.3fuH\n", i, j, K, fk, M * 1e6);  */
   return M * s;
}

/*
 *  Compute the bulk DC inductance of the coil using Lundins formula.
 */

double lundin( void)
{
   double a = Crad;
   double b = Hcoil;

   double x = 4 * a*a/(b*b);

   double f1 = (1 + 0.383901*x + 0.017108*x*x)/(1+0.258952*x);

   double L = MU*Nt*Nt*a*a*M_PI/b *( f1 - 8*a/(3*M_PI*b)) ;

   return L;
}

/*
 *  Initialise Mpt[][] and normalise as per equ 4.7.
 *
 *  Programming note: we use Mpt[1..Nt][1..Nt] to match the indices in the
 *  theory notes. Mpt[0][0] is not used.
 */

double *Mptx;                    /* Storage for the mutual inductance profile */
#define Mpt( A, B) Mptx[ (A)*(Nt+1) + (B)]

void setup_Mpt( void)
{
   int i, j;
   double s, sigma;

   double Ln = MU * Crad * ( log(8*Crad/Wrad) - 2) + (MU * Crad/4.0);

   logmsg( "Standard Ln: %.3fuH", Ln * 1e6);

   logmsg( "Quoted inductance: %.2fmH", Lcoil * 1E3);
   logmsg( "Lundin inductance: %.2fmH", lundin() * 1E3);
   if( use_lundin)
   {
      Lcoil = lundin();
      logmsg( "using Lundin inductance");
   }

   /*
    *
    *  Do an initial approximation of Mpt with a default sigma. Compare
    *  with Lcoil and adjust sigma. Iterate until the error in Lcoil is < .1%
    */

   sigma = 50;

   while( 1)
   {
      for( s=0.0, i=1; i<=Nt; i++)
         for( j=1; j<=Nt; j++) s += Mpt( i, j) = filament( i, j, sigma);

      logmsg( "sigma loop sigma=%.1f turns,  Lcoil=%.1fuH", sigma, s * 1e6);

      if( fabs( Lcoil - s)/s < 0.001) break;                /* Close enough? */
      sigma *= Lcoil/s;                        /* Revise sigma and try again */
   }

   /*
    *  Report the chosen sigma value, and also the embedded effective
    *  inductance at low frequency of the middle and an end turn.
    */

   logmsg( "chose sigma for Mpt: %.2f turns", sigma);

   for( s=0.0, j=1; j<=Nt; j++) s += Mpt( Nt/2, j);
   logmsg( "mid coil turn: %.2fuH", s * 1e6);

   for( s=0.0, j=1; j<=Nt; j++) s += Mpt( 1, j);
   logmsg( "end coil turn: %.2fuH", s * 1e6);
}

/****************************************************************************/
/*  Series Resistance                                                       */
/****************************************************************************/

double Rs;                     /* AC resistance per LCR section at w rad/sec */

/*
 *  The Medhurst phi table.
 */

float phi[ 10][ 12] =
{
   { 5.31, 5.45, 5.65, 5.80, 5.80, 5.55, 4.10, 3.54, 3.31, 3.20, 3.23, 3.41 },
   { 3.73, 3.84, 3.99, 4.11, 4.17, 4.10, 3.36, 3.05, 2.92, 2.90, 2.93, 3.11 },
   { 2.74, 2.83, 2.97, 3.10, 3.20, 3.17, 2.74, 2.60, 2.60, 2.62, 2.65, 2.81 },
   { 2.12, 2.20, 2.28, 2.38, 2.44, 2.47, 2.32, 2.27, 2.29, 2.34, 2.37, 2.51 },
   { 1.74, 1.77, 1.83, 1.89, 1.92, 1.94, 1.98, 2.01, 2.03, 2.08, 2.10, 2.22 },
   { 1.44, 1.48, 1.54, 1.60, 1.64, 1.67, 1.74, 1.78, 1.80, 1.81, 1.83, 1.93 },
   { 1.20, 1.29, 1.33, 1.38, 1.42, 1.45, 1.50, 1.54, 1.56, 1.57, 1.58, 1.65 },
   { 1.16, 1.19, 1.21, 1.22, 1.23, 1.23, 1.28, 1.32, 1.34, 1.34, 1.35, 1.40 },
   { 1.07, 1.08, 1.08, 1.10, 1.10, 1.10, 1.13, 1.15, 1.16, 1.16, 1.17, 1.19 },
   { 1.02, 1.02, 1.03, 1.03, 1.03, 1.03, 1.04, 1.04, 1.04, 1.04, 1.04, 1.05 }
};

double compute_Rs( int f)
{
   int ds, ld;
   float t, dx, dy;
   double R_ac_straight;

   ds = 10 - (int)(spacing_ratio * 10);
   dy = 10*spacing_ratio - (int)(spacing_ratio * 10);

   if( ds < 0) ds = 0;
   else
   if( ds > 8) { ds = 8; dy += 1.0; }

   t = Hcoil/2/Crad;
   if( t < 1.0)
   {
      ld = (int)(5 * t);
      dx = 5*t - (int)(5*t);
   }
   else
   if( t < 2.0)
   {
      ld = 5;
      dx = t - (int) t;
   }
   if( t < 10.0)
```

tsim.c

```c
   {
      ld = 5 + (int)(t/2);
      dx = t/2 - (int)(t/2);
   }
   else
   {
      ld = 10;
      dx = 0.5;
   }

   R_ac_straight = RcoilDC/Ns *
            (1+sqrt( f * MU * Wrad * Wrad * 2 * M_PI /17.3e-9));

   t = phi[ ds][ ld] +
       dx * (phi[ ds][ ld+1] - phi[ ds][ ld]) +
       dy * (phi[ ds-1][ ld] - phi[ ds][ ld]);

   printf( "RAC %d %.2f %.2f\n", f, R_ac_straight * Ns, R_ac_straight * Ns * t);

   return R_ac_straight * t * W;
}


/*****************************************************************************/
/*  Internal Capacitance                                                     */
/*****************************************************************************/

/*
 *  Compute the interwinding capacitance Cs(u,v) between two sections
 *  u,v as per equ 6.3, but with a bit of interpolation.
 */

#define section_to_segment( A)  ((int)(1 + (Ncp * ((A)-1))/Ns))
#define segment_to_section( A)  ((int)((((A)+0.5-1)*Ns)/Ncp))

double Cs( int u, int v)
{
   int r, s, d;
   int ux, vx, t, sx;
   int s1, s2, s1x, s2x;
   double a;

   if( (d = u-v) < 0) d = -d;
   if( d < S) return 0.0;

   if( use_palermo && d == S)
   {
      double a = 1.0/spacing_ratio;

      return 2 * M_PI * M_PI * EPSILON * Crad/log(a + sqrt( a*a-1))/S;
   }

   if( u < v) { t = u; u = v; v = t; }
   r = section_to_segment( u);
   s = section_to_segment( v);

   if( r == s || r == s+1 || r == s-1) return 0.0;
   ux = segment_to_section( r);
   t = u - ux;
   vx = v - t;
   s = section_to_segment( vx);
   if( s > Ncp) s = Ncp; if( s < 1) s = 1;

   sx = segment_to_section( s);

   if( vx < sx)
   {
      s1 = s - 1; if( s1 < 1) { s1 = 1;  return 0.0; }
      s2 = s1 + 1;
   }
   else
   if( vx > sx)
   {
      s1 = s + 1; if( s1 > Ncp) { s1 = Ncp; return 0.0;}
      s2 = s1 - 1;
   }
   else return 0.0;

   s1x = segment_to_section( s1);
   s2x = segment_to_section( s2);

   a =
      (vx - s1x) * (Cint[ s2][ r] - Cint[ s1][ r])/(s2x - s1x) + Cint[ s1][ r];

   return (Ncp * Ncp) * a/(Ns * Ns);
}

/*
 *  Compute the capacitance between the top load and the coil section k.
 */

double Ct( int k)
{
   int r = 1 + (Ncp * (k-1))/Ns;

   return Ncp * Ctor[ r]/Ns;
}

/*****************************************************************************/
/*  External Capacitance                                                     */
/*****************************************************************************/

/*
 *  Compute Y(k) according to equ 5.2.
 */

COMPLEX Y( int k)
{
   int u = 1+ (int)(Ncp * (k-1))/Ns;  /* Map section k to Cs index 1...Ncp-1 */

   return cx_scale( (COMPLEX){ Dp, 1.0}, Ncp * w * Cext[ u]/Ns);
}

/*****************************************************************************/
/*  Construction of the [A] and [C] matrices                                 */
/*****************************************************************************/

/*
 *  Complex arrays in which to set up the network equations.
 */

COMPLEX *C;
COMPLEX *AX;
COMPLEX *Vcoeff;

/*
 *  Macros for accessing A[][] array.
 */

#define coeff( row, col) \
                       AX[ (row) * (Ns+2) + (col)]   /* col, row = 1 .. Ns+1 */
#define add_coeff( row, col, f) coeff( row, col) = cx_add( coeff( row, col), f)
```

tsim.c


```
/*
 *  Add coefficients into [A] for the terms in Vm(k) as given by equ 4.8.
 *  Each coefficient is multiplied by a scaling factor.
 */

void add_Vm_coeffs( int row, int k, COMPLEX factor)
{
   int r, u;

   u = 1 + (int)((k-1)/S);                  /* Map section k to turn number u */

   for( r = 1; r <= Ns; r++)
   {
      int v = 1 + (int)((r-1)/S);                        /* Ditto for r to v */
      add_coeff( row, r+1,
               cx_scale( cx_mul( (COMPLEX){0.0, w/S}, factor), Mpt( u, v)));
   }
}

/*
 *  Install coefficients in [A] for Is(k) as given by equ 6.2.
 */

void add_Is_coeffs( int k)
{
   COMPLEX s;
   double c;
   int r;

   for( r=1; r<=Ns; r++)
   {
      c = Cs( k, r);
      s = (COMPLEX){0.0, w*c};                                /* jw Cs(k,r) */

      Vcoeff[ r] = cx_add( Vcoeff[ r], s);              /* + jw Cs(k,r) V(r) */
      Vcoeff[ k] = cx_sub( Vcoeff[ k], s);              /* - jw Cs(k,r) V(k) */
   }
}

/*
 *  Add coefficients for toroid capacitance current. Equ 6.4.
 */

void add_It_coeffs( int k)
{
   double c = Ct( k);
   COMPLEX s = (COMPLEX){0.0, w*c};

   Vcoeff[ Ns+1] = cx_add( Vcoeff[ Ns+1], s);
   Vcoeff[ k] = cx_sub( Vcoeff[ k], s);
}

/*
 *  Take the coefficients of V(k) in the Vcoeff[] array and expand them into
 *  coefficients of I(1..Ns+1), using equ 7.5.
 *
 *  A slight optimisation is used, to avoid O(Ns^2) calls to add_Vm_coeffs(),
 *  the loop in this function is equivalent to the statement:
 *
 *    for( j=Ns+1; j>= 1; j--) add_V_coeffs( row, j, Vcoeff[ j]);
 *
 *  which is only O(Ns).
 */

void substitute_V_coeffs( int row)
{
   int j;

   COMPLEX c = (COMPLEX){0.0, 0.0};

   for( j=Ns+1; j>= 1; j--)
   {
      c = cx_add( c, Vcoeff[ j]);

      if( j > 1)
      {
         add_coeff( row, j, cx_scale( c, -Rs));
         add_Vm_coeffs( row, j-1, cx_scale( c, -1));
      }
   }

   C[ row] = cx_add( C[ row], cx_mul( Vin, c));
}

/*
 *  Setup the [A] and [C] matrices of equ 7.7. We stick closely to
 *  the formulation in section 7.
 */

void setup_A_and_C( COMPLEX Zl)
{
   int row, col, k, j;

   /*
    *  First zero out the [A] and [C] matrices.
    */

   for( row=1; row<=Ns+1; row++)
   {
      for( col=1; col<=Ns+1; col++) coeff( row, col) = (COMPLEX){0.0, 0.0};
      C[ row] = (COMPLEX){0.0, 0.0};
   }

   /*
    *  Loop through the Ns equations 7.1 assigning coefficients to the
    *  rows 1..Ns of [A] and [C].
    *
    *  0 = -I(k+1) + I(k) - Y(k)V(k) + Is(k)
    */

   for( row=1, k=1; k<=Ns; k++, row++)
   {
      for( j=1; j<=Ns+1; j++) Vcoeff[ j] = (COMPLEX){0.0, 0.0};

      add_coeff( row, k+1, ((COMPLEX){-1.0, 0.0}));               /* -I(k+1) */
      add_coeff( row, k,   ((COMPLEX){1.0, 0.0}));                  /* +I(k) */

      Vcoeff[ k] = cx_add( Vcoeff[ k],  cx_scale( Y(k), -1));   /* -Y(k)V(k) */

      add_Is_coeffs( k);                                           /* +Is(k) */
      add_It_coeffs( k);                                           /* +It(k) */

      substitute_V_coeffs( row);       /* Convert coeffs of V to coeffs of I */
   }

   /*
    *  Put the coefficients from equ 7.3 into the last row of the matrix.
    *
    *  0 = -ZlI(Ns+1) + V(Ns+1) + sum[ZlIt(j)]
    */
```

tsim.c

```
   for( j=1; j<=Ns+1; j++) Vcoeff[ j] = (COMPLEX){0.0, 0.0};

   add_coeff( row, Ns+1, cx_scale( Zl, -1));                   /* -ZlI(Ns+1) */
   Vcoeff[ Ns+1] = cx_add( Vcoeff[ Ns+1], (COMPLEX){1.0, 0.0});  /* +V(Ns+1) */

   for( j=1; j<=Ns; j++)
   {
      COMPLEX s = cx_mul( (COMPLEX){0.0, w * Ct(j)}, Zl);       /* +Zl It(j) */

      Vcoeff[ Ns+1] = cx_add( Vcoeff[ Ns+1], s);
      Vcoeff[ j] = cx_sub( Vcoeff[ j], s);
   }

   substitute_V_coeffs( row);          /* Convert coeffs of V to coeffs of I */
}

/****************************************************************************/
/*  Computing the Current Distribution.                                     */
/****************************************************************************/

/*
 *  Solve equation 7.7, [A][I] + [C] = [0] by gaussian elimination to
 *  obtain [I].
 *
 *  Remember that rows and columns run 1..Neq and the elements A[0][0] and
 *  C[0] are not used.
 *
 *  Neq is the number of equations.
 */

int *used;                    /* Flags to remember which rows have been used */
int *row_use;                                  /* The row used for each column */

void solve_A_and_C( int Neq)
{
   int row, col;

   for( row = 1; row <= Neq; row++) used[ row] = 0;

   for( col = 1; col < Neq; col++)     /* For each column except the last... */
   {
      double t, g;   int pivot = 0;

      /*
       *  Select a pivot row. Look for the unused row with the highest
       *  magnitude in this column.
       */

      for( t=0.0, row=1; row <= Neq; row++)
         if( !used[ row] && (g=fabs(cx_mag( coeff( row, col)))) > t)
         {
            t = g;
            pivot = row;
         }

      if( t == 0.0)                                        /* Sanity check */
      {
         fprintf( stderr, "error: empty column %d in elimination\n", col);
         exit( 1);
      }

      /*
       *  Mark the row used, so we dont use it again, and remember the
       *  column it was used to eliminate, for use when back substituting.
       */

      used[ pivot] = col;
      row_use[ col] = pivot;

      /*
       *  Subtract multiples of the pivot row from all the other unused
       *  rows.
       */

      for( row=1; row<=Neq; row++)
         if( !used[ row])
         {
            int ct;
            COMPLEX f = cx_div( coeff( row, col), coeff( pivot, col));

            for( ct=col; ct<=Neq; ct++)
               coeff( row, ct) = cx_sub( coeff( row, ct),
                                   cx_mul( f, coeff( pivot, ct)));

            C[ row] = cx_sub( C[ row], cx_mul( f, C[ pivot]));
         }
   }

   /*
    *  One un-used row remains, which is the starting point for back
    *  substitution. Find it and add to the used list.
    */

   for( row=1; row <= Neq && used[ row]; row++);

   used[ row] = Neq;
   row_use[ Neq] = row;

   /*
    *  Back-substitute to obtain the [I] values.
    */

   for( col = Neq; col >= 1; col--)
   {
      row = row_use[ col];
      used[ row] = 0;

      if( cx_mag( coeff( row, col)) == 0.0)
      {
         fprintf( stderr, "empty coefficient\n"); exit(1);
      }

      I[ col] = cx_scale( cx_div( C[row], coeff( row, col)), -1.0);

      /*
       *  Substitute for I[col] in the remaining rows.
       */

      for( row=1; row<=Neq; row++)
         if( used[ row])
            C[ row] = cx_add( C[ row], cx_mul( coeff( row, col), I[ col]));
   }
}

/****************************************************************************/
/*  Computing the Voltage Distribution.                                     */
/****************************************************************************/

/*
 *  Given the current distribution [I], use equ 4.8 to compute the induced
 *  voltages.
 */
```

tsim.c

```
COMPLEX Vm( int k)
{
   int r, u;
   COMPLEX t = (COMPLEX){0.0, 0.0};

   u = 1 + (int)((k-1)/S);

   for( r = 1; r <= Ns; r++)
   {
      int v = 1 + (int)((r-1)/S);
      t = cx_add( t,
               cx_mul( cx_scale( (COMPLEX){ 0.0, w/S}, Mpt( u, v)), I[r+1]));
   }

   return t;
}

/*
 *  Given the current distribution, compute the array V[1..Ns+1] using
 *  equ 7.5.
 */

void compute_V( COMPLEX Zl)
{
   int k, r;

   for( k = 1; k <= Ns + 1; k++)
   {
      COMPLEX t = (COMPLEX){0.0, 0.0};

      for( r = 2; r <= k; r++) t = cx_add( t, I[r]);

      t = cx_scale( t, -Rs);

      t = cx_add( t, Vin);

      for( r=1; r <= k-1; r++) t = cx_sub( t, Vm( r));

      V[ k] = t;
   }
}

/*****************************************************************************/
/*  Stored Energy, Power Dissipated, and Q Factor                            */
/*****************************************************************************/

double Pc;          /* Total power lossed in the shunt capacitance due to Dp */
double Pw;                     /* Total I^2R losses in the winding due to Rs */
double Pl;                                /* Power dissipated in the load Zl */
double Pin;                                               /* The input power */

/*
 *  This provides a useful cross check on the program, compute the power
 *  dissipations using equ 7.8.  Pin should equal the sum Pc + Pw + Pl.
 */

void compute_power( COMPLEX Zl, COMPLEX Zin)
{
   int k;
   double g;

   /*
    *  Sum the distributed losses.
    */

   Pc = 0.0;
   Pw = 0.0;

   for( k = 1; k <= Ns; k++)
   {
      g = cx_mag( V[k]);   Pc += g * g * Y(k).re;
      g = cx_mag( I[k+1]);   Pw += g * g * Rs;
   }

   /*
    *  The input power and load power.
    */

   g = cx_mag( I[1]);      Pin = g * g * Zin.re;
   g = cx_mag( I[Ns+1]);    Pl = g * g * Zl.re;
}

double Ew;                                /* Energy stored in magnetic field */
double Ecx;                           /* Energy stored in external capacitance */
double Eci;                           /* Energy stored in internal capacitance */

/*
 *  Compute the stored energy in the resonator, using the instantaneous
 *  voltages and currents at phase 0 of Vin.
 */

void compute_energy( void)
{
   int r, s;

   Ew = Ecx = Eci = 0.0;

   /*
    *  Energy stored in external capacitance.
    */

   for( r=1; r<=Ns; r++)
   {
      double c = Y(r).im/w;
      double ph = atan2( V[r].im, V[r].re);
      double v = cx_mag( V[r]) * sin( ph);
      Ecx += 0.5 * c * v * v;
   }

   {
      double ph = atan2( V[Ns+1].im, V[Ns+1].re);
      double v = cx_mag( V[Ns+1]) * sin( ph);
      Ecx += 0.5 * Ctop * v * v;
   }

   /*
    *  Energy stored in magnetic field.
    */

   for( r = 1; r <= Ns; r++)
      for( s = 1; s <= Ns; s++)
      {
         double ph_r = atan2( I[r+1].im, I[r+1].re);
         double i_r = cx_mag( I[r+1]) * sin( ph_r);
         double ph_s = atan2( I[s+1].im, I[s+1].re);
         double i_s = cx_mag( I[s+1]) * sin( ph_s);

         Ew += 0.5 * Mpt( r, s)/S * i_r * i_s;
      }

   /*
```

tsim.c

```
    *  Energy stored in internal capacitance.
    */

   for( r = 1; r <= Ns; r++)
      for( s = 1; s <= Ns; s++)
      {
         double ph_r = atan2( V[r].im, V[r].re);
         double v_r = cx_mag( V[r]) * sin( ph_r);
         double ph_s = atan2( V[s].im, V[s].re);
         double v_s = cx_mag( V[s]) * sin( ph_s);

         Eci += 0.5 * Cs( r, s) * (v_r - v_s) * (v_r - v_s);
      }

   for( r = 1; r <= Ns; r++)
   {
      double ph_r = atan2( V[r].im, V[r].re);
      double v_r = cx_mag( V[r]) * sin( ph_r);
      double ph_s = atan2( V[Ns+1].im, V[Ns+1].re);
      double v_s = cx_mag( V[Ns+1]) * sin( ph_s);

      Eci += 0.5 * Ct( r) * (v_r - v_s) * (v_r - v_s);
   }
}

/*
 *  Compute the Q factor.
 */

double Q;

void compute_Q( int f)
{
   Q = 2 * M_PI * (Ew + Ecx + Eci) * f/Pin;
}


/****************************************************************************/
/*  Main Program                                                            */
/****************************************************************************/

void initialise( void)
{
   int i, j;
   float a;

   Ns = Nt * S;                              /* Total number of sections */

   /*
    *  Allocate memory for all the arrays.
    */

   C = (COMPLEX *) safe_malloc( sizeof( COMPLEX) * (Ns+2));
   Mptx = (double *) safe_malloc( sizeof( double) * (Nt+1) * (Nt+1));

   AX = (COMPLEX *) safe_malloc( sizeof( COMPLEX) * (Ns+2) * (Ns+2));
   I = (COMPLEX *) safe_malloc( sizeof( COMPLEX) * (Ns+2));
   V = (COMPLEX *) safe_malloc( sizeof( COMPLEX) * (Ns+2));
   Vcoeff = (COMPLEX *) safe_malloc( sizeof( COMPLEX) * (Ns+2));

   used = (int *) safe_malloc( sizeof( int) * (Ns+2));
   row_use = (int *) safe_malloc( sizeof( int) * (Ns+2));

   spacing_ratio = Nt * Wrad * 2.0/Hcoil;

   /*
    *  Report some of the info read from the description file.
    */

   logmsg( "LCR Sections %d, Capacitance segments %d", Ns, Ncp);
   logmsg( "Coil turns: %d, resistance: %.2f", Nt, RcoilDC);
   logmsg( "Coil radius: %.3f, length: %.3f", Crad, Hcoil);
   logmsg( "Conductor radius: %.3f mm", Wrad * 1e3);
   logmsg( "Spacing ratio: %.3f", spacing_ratio);

   for( i=1, a=0.0; i <= Ncp; i++) a += Cext[ i];
   logmsg( "Total external cap: %.2fpF", a * 1e12);

   for( i=1, a=0.0; i <= Ncp; i++) a += Ctor[ i];
   logmsg( "Total toroid cap to coil: %.2fpF", a * 1e12);

   logmsg( "Top loading capacitance: %.2fpF", Ctop * 1e12);
   for( i=1, a=0.0; i <= Ns; i++)
      for( j=1; j <= Ns; j++)
      {
         if( i > j) a += (i-j) * Cs( i, j);
         else
         if( i < j) a += (j-i) * Cs( i, j);
      }

   logmsg( "Weighted internal capacitance: %.2fpF", a * 1e12);
}

int main( int argc, char *argv[])
{
   int f, k;
   int f1, f2, fs;

   if( argc != 5)
   {
      fprintf( stderr, "usage: ./tsim [system] [from] [to] [step]\n\n");
      fprintf( stderr, "Frequencies must be integer Hz.\n");
      fprintf( stderr, "Results go to stdout\n");
      exit( 1);
   }

   strcpy( sysname, argv[ 1]);
   f1 = atoi( argv[ 2]);                                  /* Begin at f1 Hz */
   f2 = atoi( argv[ 3]);                                  /* End with f2 Hz */
   fs = atoi( argv[ 4]);                                      /* Step f3 Hz */

   logmsg( "simulator version: %s", VERSION);
   load_system_description();
   initialise();

   /*
    *  Construct the mutual inductance profile.
    */

   setup_Mpt();

   /*
    *  Loop over the range of frequencies.
    */

   for( f = f1; f <= f2; f += fs)
   {
      COMPLEX Zl,                                   /* The terminating load */
              Zin;                                   /* The input impedance */

      w = 2 * M_PI * f;
```

tsim.c

```
      /*
       *  Work out the AC resistance Rs per section. Initialise the terminating
       *  impedance Zl, and setup the matrices.
       */

      Rs = compute_Rs( f);
      Zl = cx_cap( w, Ctop);
      setup_A_and_C( Zl);

      /*
       *  Solve the system to obtain the current profile.
       */

      solve_A_and_C( Ns+1);

      /*
       *  From the current profile, compute the voltages, powers, and
       *  input impedance.
       */

      compute_V( Zl);

      Zin = cx_div( V[ 1], I[ 1]);
      compute_power( Zl, Zin);
      compute_energy();
      compute_Q( f);

      /*
       *  Report the results for this frequency step. The output stream is
       *  flushed so that the output file can be tailed by other utilities.
       */

      printf( "FREQ %d ", f);
      printf( "%.1f %.1f ", Zin.re, Zin.im);

      printf( "%.3f ", cx_mag( V[ Ns+1])/cx_mag( Vin));

      printf( "%.1f %.1f %.1f ", 100.0 * Pc/Pin,
                                 100.0 * Pw/Pin,
                                 100.0 * Pl/Pin);

      printf( "%.0f %.1f\n", Q, 100.0 * (Pl+Pc+Pw-Pin)/Pin);

      for( k=1; k<=Ns+1; k++)
      {
         printf( "VI %d %d ", f, k);
         printf( "%.3f %.3f ", V[k].re, V[k].im);
         printf( "%.6f %.6f\n", I[k].re, I[k].im);
      }
      fflush( stdout);
   }

   return 0;
}
```