IMPLEMENTATION OF A PHYSICS BASED MODEL FOR THE GaAs MESFET

J.W. Bandler, Q.J. Zhang and Q. Cai

SOS-90-3-L

January 1990

IMPLEMENTATION OF A PHYSICS BASED MODEL FOR THE GaAs MESFET

J.W. Bandler, Q.J. Zhang and Q. Cai

Abstract A physics based model for the GaAs MESFET is dynamically integrated into a linear/nonlinear simulation environment. The model is based on the work of Khatibzadeh and Trew. The program was written in C language and has been embedded into our research system called McCAE. This document contains the source code of all program modules related to the physical model.

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grants STR0040923 and OGP0007239.

The authors are with the Simulation Optimization Systems Research Laboratory and the Department of Electrical and Computer Engineering, McMaster University, Hamilton, Canada L8S 4L7.

I. INTRODUCTION

The physics based model for the GaAs MESFET formulated in [1] was implemented by program modules described in a companion report [2]. These modules can be embedded into harmonic balance simulator to analyze and design circuits which contain MESFETs. Since the physical/geometrical/process parameters are directly integrated into the model, it is very useful for device or circuit design optimization before the device is fabricated and is quite suitable for statistical modeling and yield optimization.

The program is written in C. There are three main files: (1) "trew.c" for the physics based model, (2) "spline.c" for cubic spline interpolation and (3) "integral.c" for the calculation of integrations used in trew.c, and three include files: "trew.h", "spline.h" and "integral.h" in the program. This document includes the source code of all the six files of the model. The theory and technical description can be found in [1] and [2]. The key modules and functions are listed in Table I.

TABLE I

LIST OF THE FUNCTIONS AND MODULES

File	Module or Function	Listing from Page
Trew.c	initrew	6
	ctrew	7
	CubicSplineWithDeriv	14
Spline.c	CubicSplineValue	16
	CubicSplineDeriv	17
	Double_Integral	18
Integral.c	Single_Integral	19
	Single_Integral_Fix	20

II. REFERENCES

[1] J.W. Bandler, Q.J. Zhang and Q. Cai, "Formulation of a physics based model for the GaAs MESFET", Department of Electrical and Computer Engineering, McMaster University, Hamilton, Canada, Report SOS-90-2-R, 1990.

[2] J.W. Bandler, Q.J. Zhang and Q. Cai, "Implementation of a physics based model for the GaAs MESFET", Department of Electrical and Computer Engineering, McMaster University, Hamilton, Canada, Report SOS-90-3-U, 1990.

III. THE SOURCE CODE

A. trew.c

```
/******************************************************************************************
 *
 * File: trew.c
 *
 * Description:  Physics based model for the GaAs MESFET
 *
 ******************************************************************************************/

/*************** CONSTANTS ****************
*  q ---------- Electronic charge.
*  Epsrom0 ---- Dielectric constant in vacuum space.
*  Pi --------- Ratio of the circumference.
*
************ INPUT COEFFICIENTS **********
*  L ---------- Gate length (μm).
*  Z ---------- Gate width (μm).
*  a ---------- Channel thickness (μm).
*  EpsromR ---- Relative dielectric constant.
*  Ec --------- Critical electric field (KV/cm).
*  Vs --------- Saturation velocity (cm/sec).
*  Vbi -------- Built in potential (V).
*  Mu0 -------- Low field mobility (cm2/V•sec).
*  D ---------- High field diffusion coefficient (cm2/sec).
*  Nd --------- Uniform doping density.
*  Lamda ------ Parameter for computing transition function.
*  Alfa ------- Parameter for computing weighted average mobility.
*  Nd1  ------- Parameter for piecewise doping.
*  Y0_doping -- Parameter for quadratic or piecewise doping.
*  Ynom_doping --- Parameter for quadratic or piecewise doping.
*  Ynom1_doping -- Parameter for piecewise doping.
*  Eps -------- allowable error for single and double integral.
*/

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

/**  Define the constants  **/

#define  q               1.6021892e-010
#define  Epsrom0         8.8541878e-009
#define  M_number        11
#define  Pi              3.1415926
#define  Eps             1.e-5
#define  UNIFORM_DOPING  1
#define  N               10
#define  REAL      float
#define  INTEGER   long int
#define  CHAR      char
#define  MIN_REAL   1.0E-34
#define  R_M_number  (REAL)(M_number - 1)

#include "trew.h"
#include "spline.h"
#include "integral.h"

REAL L, Z, a, EpsromR, Ec, Vs, Vbi, Mu0, D, Temp, Nd, Lamda, Alfa, qZdL;
REAL Nd1_doping, Y0_doping, Ynom_doping, Ynom1_doping, aa_velocity, bb_velocity,
     cc_velocity, dd_velocity;
REAL Epsrom, Pid2a, PiLd2a, EpsZdL, Chita, ChitadL, coshV, sinhV, Epsromdq;
REAL EpsdqdL, EpsromZ, LD, Ec2, Vsd4Ec2, Mus, Mud, Nd0, Qgpar;
REAL dx, dx_0, dx_L1, F1d_0, F1d_L1, F1d_x, F2d_L1, L1, L_L1, qZ, Gama, Vpo;
REAL dxv[N_SPLINE_POINTS], F1dv[N_SPLINE_POINTS], Cubspcoef[4][N_SPLINE_POINTS];
REAL V0, V1, Vgs, Vds;
CHAR Mode;
REAL dx0, F1d, Eta;
REAL cc1, cc2, cc3, cc4, cc5, cc6, cc6dL, cc7;
INTEGER  n_f1_intervals;
INTEGER OperationModes;  /* = 1,2,3 for HB, SP or DC simulation,
                            = 0 for non-simulation */

INTEGER DopingType;          /* = 1,0 for uniform and arbitrary doping */
```

```
/*************************************************************************************
 *
 * Module: initrew
 *
 * Description: Initialize the parameters, evaluate the constants and create
 *               cubic spline interpolation parameters for F1(d) w.r.t. d
 *
 *************************************************************************************/

initrew(PA)
REAL *PA;
{
   int i, t1, t2;

   n_f1_intervals = N_SPLINE_POINTS - 1;
   chkopmode(&OperationModes);

   if (a == PA[2]) t1 = 1;
   else t1 = 0;
   if (Nd == PA[9] && Nd1_doping == PA[13] &&
       Y0_doping == PA[14] && Ynom_doping == PA[15] &&
       Ynom1_doping == PA[16]) t2 = 1;
   else t2 = 0;

   L = PA[0];
   Z = PA[1];
   a = PA[2];
   EpsromR = PA[3];
   Ec = PA[4];
   Vs = PA[5];
   Vbi = PA[6];
   Mu0 = PA[7];
   D = PA[8];
   Nd = PA[9];
   Lamda = PA[10];
   Alfa = PA[11];
   Nd1_doping    = PA[13];
   Y0_doping     = PA[14];
   Ynom_doping   = PA[15];
   Ynom1_doping  = PA[16];
   aa_velocity   = PA[17];
   bb_velocity   = PA[18];
   cc_velocity   = PA[19];
   dd_velocity   = PA[20];

   if (Ynom_doping > 1.0e30) DopingType = UNIFORM_DOPING;
   else DopingType = UNIFORM_DOPING - 1;

   /**    Computing the constants    **/

   Epsrom = Epsrom0 * EpsromR;
   Pid2a = Pi / (2.0 * a);
   PiLd2a = Pid2a * L;
   EpsromZ = Epsrom * Z;
   EpsZdL = EpsromZ / L;
   Epsromdq = Epsrom / q;
   EpsdqdL = Epsromdq / L;
   coshV = cosh(PiLd2a);
   sinhV = sinh(PiLd2a);
   Chita = PiLd2a / sinh(PiLd2a);
   ChitadL = Chita / L;
   LD = L * D;
   Ec2 = 2.0 * Ec;
   Vsd4Ec2 = Vs / (Ec2 * Ec2);
   qZ = q * Z;
   qZdL = qZ / L;
   cc1 = 2.0 * a / (3.0 * L);
   cc3 = ChitadL / Pid2a;
   cc2 = cc3 * EpsromZ;
   cc4 = cc2 * coshV;
   cc5 = cc4 - cc2;
   cc6 = Chita * coshV;
   cc6dL = cc6 / L;
   cc7 = cc6dL / Pid2a;
   Nd0 = 2.0 * Epsromdq / Nd;

   if (t1 != 1  || t2 != 1) {
      /**    Calculate the pinch-off voltage   **/

      Double_Integral(0.0, a, Eps, DopingDensity, FunY, 8, &Vpo);
      Vpo = Vbi - Vpo / Epsromdq;

      /**    Computing F1(d) with N_SPLINE_POINTS values of d in [0, 1.1a]   **/

      if (DopingType != UNIFORM_DOPING) {
         dx0 = 1.1 * a / (REAL) n_f1_intervals;
         for (i = 0; i < N_SPLINE_POINTS; i++) {
            dx = (REAL)i * dx0;
            dxv[i] = dx;
            Double_Integral(0.0, a, Eps, Fun, FunY, 8, &F1d);
            F1dv[i] = F1d;
         }

      /**   creating the cubic spline interpolation for d(x) and F1(x)  **/

         CubicSplineWithDeriv(N_SPLINE_POINTS, F1dv, dxv, 0, 0.0, 0.0, Cubspcoef);
      }
   }

   /**   Parameter for calculating Qg   **/

   Single_Integral_Fix(0.0, a, DopingDensity, N, &Qgpar);
}


/**********************************************************************************
 *
 * Function: Fun
 *
 * Description: Used in Double_Integral() for calculating F1(d)
 *
 **********************************************************************************/

REAL Fun(x, y)
REAL x, y;
{
   REAL f;
   f = (1.0 - Transition(dx, y)) * DopingDensity(y);
   return(f);
}


/**********************************************************************************
 *
 * Function: FunY
 *
 * Description: Computing the boundary of second integral used in Double_Integral()
 *              for calculating F1(d)
 *
 **********************************************************************************/

FunY(x, yy1, yy2)
REAL x, *yy1, *yy2;
{
   *yy1 = x;
   *yy2 = a;
 }


/**********************************************************************************
 *
 * Module: ctrew
 *
 * Description: Compute the conduction currents $I_d$, $I_s$ and accumulative
 *              charges $Q_g$, $Q_d$ and $Q_s$ from $V_1$, $V_{gs}$ and $V_{ds}$.
 *
 **********************************************************************************/

void ctrew(V1f, Vgsf, Vdsf, Id, Is, Qg, Qd, Qs)
REAL *V1f, *Vgsf, *Vdsf;
REAL *Id, *Is, *Qg, *Qd, *Qs;
{
   REAL Id_c, Is_c, Qdd, Qss, Qg0, Qg1;
   REAL Es, Ed, E1, E2;
```

```
V1 = *V1f;
Vgs = *Vgsf;
Vds = *Vdsf;
V0 = Vds - V1;

/**  Computing the critical length L1 and determining the operation mode  **/

E1 = V1 / L;
E2 = V0 * ChitadL;
Es = E1 + E2;
Ed = E1 + E2 * coshV;
Es = fabs(Es);
Ed = fabs(Ed);

if ((Es <= Ed) && (Ed <= Ec2)) {
   Mode = 'A';
   L1 = L;
}
else if ((Ec2 <= Es) && (Es <= Ed)) {
   Mode = 'C';
   L1 = 0.0;
}
else {
   Mode = 'B';
   if( V0 < MIN_REAL) L1 = L;
   else {
      Eta = (Ec2 * L - V1) / (V0 * Chita);
      if (Eta > 1.0e15) Eta = 1.0e15;
      if (Eta < 1.0) Eta = 1.0;
      L1 = log(Eta + sqrt(Eta * Eta - 1.0)) / Pid2a;
   }
   if (L <= L1) {
      Mode = 'A';
      L1 = L;
   }
   if (L1 <= 0.0) {
      Mode = 'C';
      L1 = 0.0;
   }
}

L_L1 = L - L1;

/**   Computing the depletion-layer-width at x = 0 and x = L1   **/

E1 = Vbi - Vgs;
E2 = V1 * L1 / L + E1;
F1d_L1 = Epsromdq * E2;

if (DopingType == UNIFORM_DOPING) {
   if (E1 < 0.0) E1 = 0.0;
   if (E2 < 0.0) E2 = 0.0;
   dx_0 = sqrt(E1 * Nd0);
   dx_L1 = sqrt(E2 * Nd0);
}
else {
   F1d_0 = Epsromdq * E1;
   dx_0 = CubicSplineValue(F1d_0, n_f1_intervals, F1dv, Cubspcoef);
   dx_L1 = CubicSplineValue(F1d_L1, n_f1_intervals, F1dv, Cubspcoef);
}

/**    Computing the coefficient Gama    **/

F2d_L1 = - Epsromdq * (Vpo - Vbi) - F1d_L1;
if (Mode == 'A') Gama = 0.0;
else Gama = - EpsdqdL * V1 / F2d_L1;

/**    Evaluating the conducting currents Id_c and Is_c    **/

/**  Is_c  **/

Mus = Wamobility(0.0);
Single_Integral_Fix(0.0, a, FunIs, N, &Is_c);
*Is = qZdL * Is_c ;

/**  Id_c  **/
```

```
   Mud = Wamobility(L);
   Single_Integral_Fix(0.0, a, FunId, N, &Id_c);
   *Id = qZdL * Id_c;

   /**    Evaluating accumulative charges Qg, Qd and Qs    **/

   if ((int)OperationModes == 3) {    /* DC simulation */
      *Qd = 0.0;
      *Qs = 0.0;
      *Qg = 0.0;
   }
   else {

      /**   Qs and Qd   **/

      Qss = cc1 * V1;
      Qdd = Qss + cc7 * V0;
      *Qd = EpsromZ * Qdd;
      Qss = - Qss - cc3 * V0;
      *Qs =  EpsromZ * Qss;

      /**   Qg   **/

      if (Mode == 'C') Qg1 = 0.0;
      else Double_Integral(0.0, L1, Eps, FunQg1, FunYy, 8, &Qg1);
      Qg0 = qZ * Qg1;
      Single_Integral_Fix(0.0, a, FunQg2, N, &Qg1);
      Qg0 += qZ * L_L1 * (Qgpar - Qg1 - 0.5 * Gama * L_L1 * Qg1);
      Qg0 += cc5 * V0;
      *Qg = - Qg0;
   }
}

/***************************************************************************************
 *
 * Function: FunIs
 *
 * Description: Used in Single_Integral_Fix() for calculating Is
 *
 ***************************************************************************************/

REAL FunIs(y)
REAL y;
{
   REAL f1, f2, f3, Exi, Eyi;
   f1 = y/a - 1.0;
   f2 = V1 * (1.0 - f1 * f1) - (V1 - Vds) * Chita * sin(Pid2a * y);
   f2 = f2 * Mus;
   if (Mode == 'C') f2 = f2 - LD * Gama;
   f3 = f2 * Transition(dx_0, y) * DopingDensity(y);
   return(f3);
}

/***************************************************************************************
 *
 * Function: FunId
 *
 * Description: Used in Single_Integral_Fix() for calculating Id
 *
 ***************************************************************************************/

REAL FunId(y)
REAL y;
{
   REAL f, f1, f2, f3, f4, Exi, Eyi;
   f1 = Mud * (1.0 + Gama * L_L1);
   f2 = y/a -1.0;
   f2 = (1.0 - f2 * f2) * V1;
   f3 = (V1 - Vds) * cc6 * sin(Pid2a * y);
   f4 = (f2 - f3) * f1 - LD * Gama;
   f = f4 * Transition(dx_L1, y) * DopingDensity(y);
   return(f);
}
```

```
/*************************************************************************************
 *
 * Function: FunQg1
 *
 * Description: Used in Double_Integral() for calculating Qg
 *
 *************************************************************************************/

REAL FunQg1(x, y)
REAL x, y;
{
   REAL ff, f1;
   f1 = V1 * x / L + Vbi - Vgs;
   if (DopingType == UNIFORM_DOPING) {
      if (f1 < 0.0 ) f1 = 0.0;
      dx = sqrt(Nd0 * f1);
   }
   else {
      F1d_x = Epsromdq * f1;
      dx = CubicSplineValue(F1d_x, n_f1_intervals, F1dv, Cubspcoef);
   }
   ff = (1.0 - Transition(dx, y)) * DopingDensity(y);
   return(ff);
}


/*************************************************************************************
 *
 * Function: FunQg2
 *
 * Description: Used in Single_Integral_Fix() for calculating Qg
 *
 *************************************************************************************/

REAL FunQg2(y)
REAL y;
{
   REAL f;
   f = Transition(dx_L1, y) * DopingDensity(y);
   return(f);
}


/*************************************************************************************
 *
 * Function: ElectricField_x
 *
 * Description: Computing the x component of electric field
 *
 *************************************************************************************/

REAL ElectricField_X(x, y)
REAL x, y;
{
   REAL f, yy;
   yy = y/a - 1.0;
   f = - V1 * (1.0 - yy * yy ) / L - ChitadL * cosh(Pid2a * x) * sin(Pid2a * y) * V0;
   return(f);
}


/*************************************************************************************
 *
 * Function: ElectricField_Y
 *
 * Description: Computing the y component of electric field
 *
 *************************************************************************************/

REAL ElectricField_Y(x, y)
REAL x, y;
{
   REAL f1, f2, f;
   if (y == a) f = 0.0;
   else {
      if (x == 0.0) dx = dx_0;
      else if (L1 <= x) dx = dx_L1;
      else {
         f1 = V1 * x / L + Vbi - Vgs;
         if (DopingType == UNIFORM_DOPING) {
            if (f1 < 0.0) f1 = 0.0;
            dx = sqrt(Nd0 * f1);
         }
         else {
            F1d_x = Epsromdq * f1;
            dx = CubicSplineValue(F1d_x, n_f1_intervals, F1dv, Cubspcoef);
         }
      }
      Single_Integral_Fix(y, a, FunEy1, N, &f1);
      f2 = ChitadL * sinh(Pid2a * x) * cos(Pid2a * y) * V0;
      f = - f1 / Epsromdq - f2;
      if (x > L1) {
         Single_Integral_Fix(y, a, FunEy2, N, &f1);
         f2 = Gama * (x - L1) * f1 / Epsromdq;
         f = f + f2;
      }
   }
   return(f);
}


/***************************************************************************************
 *
 * Function: FunEy1
 *
 * Description: Used in Single_Integral_Fix() for calculating the y component
 *              of electric field
 *
 ***************************************************************************************/

REAL FunEy1(y)
REAL y;
{
   REAL f;
   f = (1.0 - Transition(dx, y)) * DopingDensity(y);
   return(f);
}


/***************************************************************************************
 *
 * Function: FunEy2
 *
 * Description: Used in Single_Integral_Fix() for calculating the y component
 *              of electric field
 *
 ***************************************************************************************/

REAL FunEy2(y)
REAL y;
{
   REAL f;
   f = Transition(dx, y) * DopingDensity(y);
   return(f);
}


/***************************************************************************************
 *
 * Function: FreeElectronDensity
 *
 * Description: Calculating the free electron density
 *
 ***************************************************************************************/

REAL FreeElectronDensity(x, y)
REAL x, y;
{
   REAL f, f1;
   f1 = Transition(dx, y) * DopingDensity(y);
   if (x <= L1) return(f1);
   f = (1.0 + Gama * (x - L1)) * f1;
   return(f);
}
```

```
/**********************************************************************************
 *
 * Function: Transition
 *
 * Description: Evaluating the transition function
 *
 *********************************************************************************/

REAL Transition(d, y)
REAL d, y;
{
   REAL f;
   f = (y - d)/Lamda;
   if( f > 40.0 )  f = 40.0;
   f = 1.0 + exp(f);
   f = 1.0 - 1.0 / f;
   return(f);
}


/**********************************************************************************
 *
 * Function: DopingDensity
 *
 * Description: Computing the doping density
 *
 *********************************************************************************/

REAL DopingDensity(y)
REAL y;
{
   REAL f;
   if (DopingType == UNIFORM_DOPING) f = Nd;       /* uniform doping */
   else if (Ynom1_doping > 1.0e30) {               /* quadratic doping */
      f = (y - Y0_doping) / Ynom_doping;
      f = -0.5 * f * f;
      f = Nd * exp(f);
   }
   else  {                                         /* piecewise doping */
      if (y <= Y0_doping) f = Nd + Nd1_doping;
      else f = Nd * exp((Y0_doping - y) / Ynom_doping)
              + Nd1_doping * exp((Y0_doping - y) / Ynom1_doping);
   }
   return(f);
}


/**********************************************************************************
 *
 * Function: FunYy
 *
 * Description: Computing the boundary of the second integration used in
 *              Double_Integral() for calculating Qg
 *
 *********************************************************************************/

FunYy(x, y1, y2)
REAL x, *y1, *y2;
{
   *y1 = 0.0;
   *y2 = a;
}


/**********************************************************************************
 *
 * Function: Wamobility
 *
 * Description: Calculating weighted average mobility
 *
 *********************************************************************************/

REAL Wamobility(x)
REAL x;
{
   REAL Smu, Somiga, Exi, Eyi, Mui, omigai, yyi, yi, f1, f2, V1dL, ChitadLV0;
   int i;
   V1dL = V1 / L;
   ChitadLV0 = ChitadL * V0;
   Smu = 0.0;
   Somiga = 0.0;
   for (i = 1; i <= M_number; i++) {
      yyi = (REAL)(i - 1) / R_M_number;
      omigai = pow(yyi, Alfa);
      yi = yyi * a;
      f1 = yi / a - 1.0;
      f1 = V1dL * (1.0 - f1 * f1);
      if (x == 0.0) {
         Exi = - f1 - ChitadLV0 * sin(Pid2a * yi);
         if (a <= yi) Eyi = 0.0;
         else {
            dx = dx_0;
            Single_Integral_Fix(yi, a, FunEy1, N, &f1);
            Eyi = - f1 / Epsromdq;
         }
      }
      else {
         Exi = - f1 - ChitadLV0 * coshV * sin(Pid2a * yi);
         if (a <= yi) Eyi = 0.0;
         else {
            dx = dx_L1;
            Single_Integral_Fix(yi, a, FunEy1, N, &f1);
            Single_Integral_Fix(yi, a, FunEy2, N, &f2);
            Eyi = - f1 / Epsromdq - ChitadLV0 * sinhV * cos(Pid2a * yi) + Gama * L_L1 * f2 / Epsromdq;
         }
      }
      Eyi = omigai * Eyi;
      Mui = Mu_E(Exi, Eyi);
      Somiga += omigai;
      Smu += omigai * Mui;
   }
   Smu /= Somiga;
   return(Smu);
}


/************************************************************************************
 *
 * Function: Mu_E
 *
 * Description: Creating the relationship of mobility and electric field for
 *              calculating the weighted average mobility
 *
 ************************************************************************************/

REAL Mu_E(Exi, Eyi)
REAL Exi, Eyi;
{
   REAL f, Ei, f2, f3, temp;
   Ei = sqrt(Exi * Exi + Eyi * Eyi);
   if (Ei < 1E-38) f3 = 0.0;
   else {
      if (Ec2 <= Ei) f = Vs / Ei;
      else f = Mu0 - Vsd4Ec2 * Ei;
      temp = Ei / aa_velocity;
      f2 = Vs / cc_velocity * bb_velocity * (( temp - cc_velocity) / (temp * temp
           * temp * temp + bb_velocity) + cc_velocity / bb_velocity);
      f3 = (1.0 - dd_velocity) * f + dd_velocity * f2 / Ei;
   }
   return(f3);
}
```

B. Spline.c

```
/************************************************************************************
 *
 * File: spline.c
 *
 * Description:  Cubic spline interpolation
 *
 ************************************************************************************/
```

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

#define REAL  float
#define MAX     100

#include "spline.h"
#include "integral.h"


/************************************************************************************
 *
 * Module: CubicSplineWithDeriv
 *
 * Description: Compute the cubic spline interpolation with specified
 *              derivative endpoint conditions
 *
 ************************************************************************************/

CubicSplineWithDeriv(Ndata, Xdata, Fdata, Ideriv, Lderiv, Rderiv, Coef)
int Ndata, Ideriv;
REAL *Xdata, *Fdata, Coef[][N_SPLINE_POINTS];
REAL Lderiv, Rderiv;
{
   int i, J1;
   REAL Deriv2[MAX+1];
   REAL H, R;

   if (Ideriv == 0) {
      Deriv2[0] = 0.0;
      Deriv2[Ndata] = 0.0;
   }

   else {
      Deriv2[0] = Lderiv;
      Deriv2[Ndata] = Rderiv;
   }

   switch (Ideriv) {
      case 0: CubicSpline1(Ndata, Xdata, Fdata, Deriv2);
              break;
      case 1: CubicSpline3(Ndata, Xdata, Fdata, Deriv2);
              break;
      case 2: CubicSpline1(Ndata, Xdata, Fdata, Deriv2);
              break;
      case 3: CubicSpline2(Ndata, Xdata, Fdata, Deriv2);
              break;
      default: printf("Ideriv out of range: 0 <= Ideriv <= 3\n");
               exit(1);
   }

   for (i = 0; i < Ndata; i++) {
      J1 = i + 1;
      H = Xdata[J1] - Xdata[i];
      R = Fdata[J1] -Fdata[i];
      Coef[0][i] = (Deriv2[J1] - Deriv2[i])/(6.0*H);
      Coef[1][i] = Deriv2[i]/2.0;
      Coef[2][i] = R/H - H*(Deriv2[J1] + 2.0*Deriv2[i])/6.0;
      Coef[3][i] = Fdata[i];
   }
}


/************************************************************************************
 *
 * Module: CubicSpline1
 *
 * Description: Calculating cubic spline coefficients given the values of the
 *              second derivative f"(Xo) and f"(Xn) in the first and last points
 *              Xo and Xn. Return the second derivative values at breakpoints
 *
 ************************************************************************************/

CubicSpline1(Ndata, Xdata, Fdata, Deriv2)
int Ndata;
REAL  *Xdata, *Fdata, *Deriv2;
{
   int i, N1, J1, J2;
   REAL H, H1, H2, R1, R2, Z;
   REAL F[MAX+1], G[MAX+1];

   N1 = Ndata - 1;
   F[0] = 0.0;
   G[0] = 0.0;

   for (i = 0; i <= N1; i++){
      J2 = i + 1;
      H2 = Xdata[J2] - Xdata[i];
      R2 = (Fdata[J2] - Fdata[i]) / H2;
      if (i > 0) {
         Z = 1 / (2*(H1 + H2) - H1*G[J1]);
         G[i] = Z*H2;
         H = 6*(R2 - R1);
         if (i == 1) H = H - H1*Deriv2[0];
         if (i == N1) H = H - H2*Deriv2[Ndata];
         F[i] = Z*(H - H1*F[J1]);
      }
      J1 = i;
      H1 = H2;
      R1 = R2;
   }

   Deriv2[N1] = F[N1];
   if (N1 > 1) {
      for (i = 1; i < N1; i++) {
         J2 = Ndata - i;
         Deriv2[J2] = F[J2] - G[J2]*Deriv2[J2+1];
      }
   }
}


/*************************************************************************************
 *
 * Module: CubicSpline2
 *
 * Description: Calculating cubic spline coefficients given the second derivative
 *              relationship of the form f"(Xo) = u*f"(X1) and f"(Xn) = v*f"[X(n-1)].
 *              Return the second derivative values at breakpoints
 *
 *************************************************************************************/

CubicSpline2(Ndata, Xdata, Fdata, Deriv2)
int Ndata;
REAL *Xdata, *Fdata, *Deriv2;
{
   int i, N1, J1, J2;
   REAL H1, H2, R1, R2, Y1, Y2, U, V, Z;
   REAL F[MAX+1], G[MAX+1];

   N1 = Ndata - 1;
   G[0] = 0.0;
   F[0] = 0.0;
   U = Fdata[0];
   V = Fdata[Ndata];
   Y2 = 2.0;

   for (i = 0; i <= N1; i++){
      J2 = i + 1;
      H2 = Xdata[J2] - Xdata[i];
      R2 = (Fdata[J2] - Fdata[i])/H2;
      if (i > 0) {
         Y1 = 2.0;
         if (i == 1) Y1 = 2.0 + U;
         if (i == N1) Y2 = 2.0 + V;
         Z = 1.0/((Y1 - G[J1])*H1 + Y2*H2);
         G[i] = Z*H2;
         F[i] = Z*(6.0*(R2 - R1) - H1*F[J1]);
      }
      J1 = i;
      H1 = H2;
      R1 = R2;
   }
   Deriv2[N1] = F[N1];
   if (N1 > 1) {
      for (i = 1; i < N1; i++){
         J2 = Ndata - i;
         Deriv2[J2] = F[J2] - G[J2]*Deriv2[J2+1];
      }
   }
   Deriv2[0] = U*Deriv2[1];
   Deriv2[Ndata] = V*Deriv2[N1];
}


/*************************************************************************************
 *
 * Module: CubicSpline3
 *
 * Description: Calculating cubic spline coefficients given the values of first
 *              derivative f'(Xo) and f'(Xn) at the first and last points Xo and
 *              Xn. Return the second derivative values at breakpoints
 *
 *************************************************************************************/

CubicSpline3(Ndata, Xdata, Fdata, Deriv2)
int Ndata;
REAL *Xdata, *Fdata, *Deriv2;
{
   int i, N1, J1, J2;
   REAL H1, H2, R1, R2, Z;
   REAL F[MAX+1], G[MAX+1];
   N1 = Ndata - 1;
   J1 = 0;
   H1 = 0.0;
   F[0] = 0.0;
   G[0] = 0.0;
   R1 = Deriv2[0];

   for (i = 0; i <= Ndata; i++){
      if (i > N1) {
         H2 = 0.0;
         R2 = Deriv2[Ndata];
      }
      else {
         J2 = i + 1;
         H2 = Xdata[J2] - Xdata[i];
         R2 = (Fdata[J2] - Fdata[i])/H2;
      }
      Z = 1.0/(2.0*(H1 + H2) - H1*G[J1]);
      G[i] = Z*H2;
      F[i] = Z*(6.0*(R2 - R1) - H1*F[J1]);
      J1 = i;
      H1 = H2;
      R1 = R2;
   }
   Deriv2[Ndata] = F[Ndata];
   for (i = 0; i <= N1; i++){
      J2 = Ndata - i;
      Deriv2[J2] = F[J2] - G[J2]*Deriv2[J2+1];
   }
}


/*************************************************************************************
 *
 * Function: CubicSplineValue
 *
 * Description: Calculating the value of cubic spline at point X
 *
 *************************************************************************************/

REAL CubicSplineValue(X, Ndata, Xdata, Coef)
int Ndata;
REAL X, *Xdata, Coef[][N_SPLINE_POINTS];
{
   int Low, Up, Center, i;
   REAL H, Fval;

   if (X < Xdata[0] ) X = Xdata[0];
   else if ( X > Xdata[Ndata]) X = Xdata[Ndata];

   /**  Find the interval including the point X using binary search  **/

   Low = 0;
   Up = Ndata;

   while (Up - Low > 1) {
      Center = (Up + Low)/2;
      if (X < Xdata[Center]) Up = Center;
      else Low = Center;
   }

   /**  Evaluate the value of cubic spline at X  **/

   i = Low;
   H = X - Xdata[i];
   Fval = Coef[0][i]*H*H*H + Coef[1][i]*H*H + Coef[2][i]*H + Coef[3][i];
   return (Fval);
}


/***************************************************************************************
 *
 * Function: CubicSplineDrive
 *
 * Description: Calculating the derivative value of cubic spline at point X
 *
 ***************************************************************************************/

REAL CubicSplineDeriv(X, Ndata, Xdata, Coef, Order)
int Ndata, Order;
REAL X, *Xdata, Coef[][N_SPLINE_POINTS];
{
   int Low, Up, Center, i;
   REAL H, Dval;

   if (X < Xdata[0] ) X = Xdata[0];
   else if ( X > Xdata[Ndata]) X = Xdata[Ndata];

   /* Check for Order >= 0 */

   if (Order < 0) {
      printf("Order outside the range Order >= 0\n");
      exit(1);
   }

   /**   Order > 3, derivative value Dval = 0   **/

   if (Order > 3) return(0);

   /**  Find the interval including the point x using binary search  **/

   Low = 0;
   Up = Ndata;

   while (Up - Low > 1) {
      Center = (Up + Low)/2;
      if (X < Xdata[Center]) Up = Center;
      else Low = Center;
   }

   /**  Evaluate the derivative value  **/

   i = Low;
   H = X - Xdata[i];

   switch (Order) {
      case 0: Dval = Coef[0][i]*H*H*H + Coef[1][i]*H*H + Coef[2][i]*H + Coef[3][i];
              break;
      case 1: Dval = 3*Coef[0][i]*H*H + 2*Coef[1][i]*H + Coef[2][i];
              break;
      case 2: Dval = 6*Coef[0][i]*H + 2*Coef[1][i];
              break;
      case 3: Dval = 6*Coef[0][i];
   }
   return(Dval);
}
```

C. integral.c

```
/*************************************************************************************
 *
 * File:  integral.c
 *
 * Description:  Evaluating the double or single integration
 *
 *************************************************************************************/

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

#define REAL float
#define MAX_N_STEPS    256

#include "integral.h"
#include "trew.h"


/*************************************************************************************
 *
 * Module:  Double_Integral
 *
 * Description:  Calculating the double integral
 *
 *************************************************************************************/

/***********  INPUT  ***********
 * Lower ---- lower bound of the first integral (REAL)
 * Upper ---- upper bound of the first integral (REAL)
 * Eps ------ permit error of the integral (REAL)
 * Fun ------ a pointer to the function to be integrated
 * FunY ----- a pointer to the function for calculating the lower boundary
 *            and upper boundary of the second integral
 * K -------- an integer to control the step
 *
 **********  OUTPUT  ***********
 * Sum ------ a pointer to the resulting value of the integral (REAL)
 */

Double_Integral(Lower, Upper, Eps, Fun, FunY, K, Sum)
REAL Lower, Upper, Eps, *Sum;
REAL (*Fun)();
int (*FunY)();
int K;
{
   int n, n1, n2, n3, Kflag, i;
   REAL center1, center2, ss1, ss2, tb1, tb2, step1, step2, x, s, sum0, sum1;

   n2 = 1;
   Kflag = 0;
   center2 = fabs(Lower) + fabs(Upper);
   step2 = 0.5 * (Upper - Lower);
   Single_Integral_1(Lower, Eps, &Kflag, &n1, &ss1, K, Fun, FunY);
   Single_Integral_1(Upper, Eps, &Kflag, &n3, &ss2, K, Fun, FunY);
   n = n1 + n3 + 2;
   tb1 = step2 * (ss1 + ss2);
   do {
      x = Lower - step2;
      tb2 = 0.5 * tb1;
      for (i = 1; i <= n2; i++){
         x = x + 2.0 * step2;
         Single_Integral_1(x, Eps, &Kflag, &n1, &s, K, Fun, FunY);
         n = n + n1 + 1;
         tb2 = tb2 + step2 * s;
      }
      sum1 = (4.0*tb2 - tb1)/3.0;
      n2 = n2 + n2;
      if (n2 > K) {
         if (fabs(sum1 - sum0) < Eps*(fabs(sum1) + 1.0)) goto END2;
      }
      sum0 = sum1;
      tb1 = tb2;
      step2 = 0.5*step2;
   } while(n2 <= MAX_N_STEPS);

END2:;
   *Sum = sum1;
   return(1);
}


/*************************************************************************************
 *
 * Module: Single_Integral_1
 *
 * Description: Calculating the single integral. This routine is called by Double_Integral()
 *
 *************************************************************************************/

Single_Integral_1(x, Eps, Kflag, nn, s, K, Fun, FunY)
REAL x, Eps, *s, (*Fun)();
int (*FunY)();
int *Kflag, *nn, K;
{
   REAL step1, center1, t1, t2, y1, y2, y, s0, s1;
   int i, n;

   n = 1;
   (*FunY)(x, &y1, &y2);
   step1 = 0.5*(y2 - y1);
   center1 = fabs(y1) + fabs(y2);
   t1 = step1 * ( (*Fun)(x, y1) + (*Fun)(x, y2) );

   do {
      y = y1 - step1;
      t2 = 0.5 * t1;
      for (i = 1; i <= n; i++) {
         y = y + 2.0*step1;
         t2 = t2 + step1*(*Fun)(x, y);
      }
      s1 = (4.0*t2 - t1)/3.0;
      n = n + n;
      if (n > K) {
         if (fabs(s1 - s0) < Eps*(fabs(s1) + 1.0)) goto END1;
      }
      s0 = s1;
      t1 = t2;
      step1 = 0.5*step1;
   } while(n <= MAX_N_STEPS);

END1:;
   *s = s1;
   *nn = n;
   return(1);
}


/*************************************************************************************
 *
 * Module: Single_Integral
 *
 * Description: Calculating the single integral of a function with upper and lower
 *              boundary conditions using variable step Simpson's numerical method.
 *
 *************************************************************************************/

/**********  INPUT  ***********
 *  Lower ---- lower bound (REAL)
 *  Upper ---- Upper bound (REAL)
 *  Eps ------ permit error for the integral (REAL)
 *  Fun ------ a pointer to the function to be integrated
 *  K -------- an integer to determine the initial step
 *
 **********  OUTPUT  **********
 *  Sum ------ a pointer to the resulting value of integral
 */
```

```
Single_Integral(Lower, Upper, Eps, Fun, K, Sum)
REAL Lower, Upper, Eps, *Sum;
REAL (*Fun)();
int K;
{
   REAL center, step, s0, s1, s2, x;
   int i, n;

   n = 1;
   center = fabs(Lower) + fabs(Upper);
   step = 0.5 * (Upper - Lower);
   s1 = step * ((*Fun)(Lower) + (*Fun)(Upper));
   do {
      x = Lower - step;
      s2 = 0.5 * s1;
      for (i = 1; i <= n; i++) {
         x = x + 2.0 * step;
         s2 = s2 + step * (*Fun)(x);
      }
      *Sum = (4.0 * s2 - s1) / 3.0;
      n = n + n;
      if ( n > K ) {
         if ( fabs(*Sum - s0) < (1.0 + fabs(*Sum))*Eps ) goto END;
      }
      s0 = *Sum;
      s1 = s2;
      step = 0.5 * step;
   } while (n <= MAX_N_STEPS);

END:;
}


/*****************************************************************************************
 *
 * Module: Single_Integral_Fix
 *
 * Description: Calculating single integration with fixed step Simpson method
 *
 *****************************************************************************************/

/**********  INPUT  ***********
 * Lower ---- Lower bound of the integral
 * Upper ---- Upper bound of the integral
 * N ---- integer to determine the step
 * Fun ---- a pointer to the integrated function
 *
 **********  OUTPUT  **********
 * Sum ---- a pointer to the result value of the integral
 */

Single_Integral_Fix(Lower, Upper, Fun, N, Sum)
REAL Lower, Upper, *Sum;
REAL (*Fun)();
int N;
{
   REAL H, f1, f2;
   int i;
   H = (Upper - Lower) / (REAL)(2 * N);
   f1 = 0.5 * (Fun(Lower) - Fun(Upper));
   for (i = 1; i <= N; i++)
      f1 = f1 + 2.0 * Fun(Lower + (REAL)(2 * i - 1) * H)
           + Fun(Lower + 2 * H * (REAL)i);
   *Sum = (Upper - Lower) * f1 / (REAL)(3 * N);
}
```

D. Header Files

```
/*****************************************************************************************
 * File: trew.h
 *
 * Description:  header file for trew.c
 *
 *****************************************************************************************/
```

```
  REAL DopingDensity(), ElectricField_X(), Transition();
  REAL Fun(), FunQg1(), FunQg2(), FunQg3(), FreeElectronDensity();
  REAL FunIs(), FunId(), FunQd(), FunQs();
  REAL Wamobility(), Mu_E();
  REAL ElectricField_Y(), FunEy1(), FunEy2();
  int FunY(), FunYy();
  int initrew();
  void ctrew();


/*************************************************************************************
 * File: spline.h
 *
 * Description:  header file for spline.c
 *
 *************************************************************************************/

  REAL CubicSplineValue(), CubicSplineDeriv();
  int CubicSplineWithDeriv();
  int CubicSpline1(), CubicSpline2(),  CubicSpline3();

  #define N_SPLINE_POINTS 40


/*************************************************************************************
 * File: integral.h
 *
 * Description:  header file for integral.c
 *
 *************************************************************************************/

  int Double_Integral(), Single_Integral_1(), Single_Integral();
  int Single_Integral_Fix();
```