Users Manual for RNS: version 2.0

Sharon Morsink

August 25, 2023

1 Introduction

RNS version 2.0 is a set of routines which will integrate the Einstein field equations for a rapidly rotating neutron star given a perfect fluid equation of state. This version of the code will be most useful for people who need to use the metric in another application. This manual will explain how to use the routines.

2 Metric and Coordinates

The neutron star models which we compute are assumed to be stationary, axisymmetric, uniformly rotating perfect fluid solutions of the Einstein field equations. The assumptions of stationarity and axisymmetry allow the introduction of two coordinates ϕ and t on which the space-time metric doesn't depend. The metric, $g_{\alpha\beta}$ can be written as

$$ds^2 = -e^{\gamma+\rho}dt^2 + e^{\gamma-\rho}\bar{r}^2\sin^2\theta\left(d\phi - \omega dt\right)^2 + e^{2\alpha}\left(d\bar{r}^2 + \bar{r}^2 d\theta^2\right), \tag{1}$$

where the metric potentials ρ, γ, α and ω depend only on the coordinates $\bar{r}$ and θ. The function $\frac{1}{2}(\gamma+\rho)$ is the relativistic generalization of the Newtonian gravitational potential; the time dilation factor between an observer moving with angular velocity ω and an observer at infinity is $e^{\frac{1}{2}(\gamma+\rho)}$. The coordinate $\bar{r}$ is not the same as the Schwarzschild coordinate r. In the limit of spherical symmetry, $\bar{r}$ corresponds to the isotropic Schwarzschild coordinate. Circles centred about the axis of symmetry have circumference $2\pi r$ where r is related to our coordinates $\bar{r}, \theta$ by

$$r = e^{\frac{1}{2}(\gamma-\rho)}\bar{r}\sin\theta. \tag{2}$$

The metric potential ω is the angular velocity about the symmetry axis of zero angular momentum observers (ZAMOs) and is responsible for the Lense-Thirring effect. The fourth metric potential, α specifies the geometry of the two-surfaces of constant t and ϕ. When the star is non-rotating, the exterior geometry is that of the isotropic Schwarzschild metric, with

$$e^{\frac{1}{2}(\gamma+\rho)} = \frac{1-M/2\bar{r}}{1+M/2\bar{r}},\ e^{\frac{1}{2}(\gamma-\rho)} = e^{\alpha} = \left(1+M/2\bar{r}\right)^2,\ \omega = 0. \tag{3}$$

The program uses a compactified coordinate s which is related to $\bar{r}$ by

$$\bar{r} = \bar{r}_e\left(\frac{s}{1-s}\right), \tag{4}$$

where $\bar{r}_e$ is the value of $\bar{r}$ at the star's equator. This definition of s gives

$$s = 0.5 \quad \Leftrightarrow \bar{r} = \bar{r}_e \tag{5}$$

$$s = 1.0 \quad \Leftrightarrow \bar{r} \to \infty \tag{6}$$

The angular variable μ, defined by

$$\mu = \cos\theta$$

is used by the program.

3 Creating the Numerical Grid

The user specifies the numerical grid size in the makefile. The parameter `MDIV` (*divisions of* μ) specifies the number of spokes in the angular direction, while `SDIV` (*divisions of* s) specifies the number of spokes in the radial direction. For example the following line in the makefile:

```
#STANDARD
SIZE=-DMDIV=65 -DSDIV=129
```

sets the number of angular spokes to 65 and the number of radial spokes to 129.

The function make_grid is called at the beginning of the application in order to set up the numerical grid.

3.1 make_grid

```
/*****************************************************************/
/* Create computational grid.                                    */
/* Points in the mu-direction are stored in the array mu[i].     */
/* Points in the s-direction are stored in the array s_gp[j].    */
/*****************************************************************/
void make_grid(double s_gp[SDIV+1],
               double mu[MDIV+1])
{
  int m, s;                          /* counters */

      for(s=1;s<=SDIV;s++)
         s_gp[s] = SMAX*(s-1.0)/(SDIV-1.0);

/* s_gp[1] = 0.0     corresponds to the center of the star
   s_gp[SDIV] = SMAX corresponds to infinity */

/* SMAX is defined in the file consts.h */

      for(m=1;m<=MDIV;m++)
         mu[m] = (m-1.0)/(MDIV-1.0);
```

```
/* mu[1] = 0.0    corresponds to the plane of the equator
   mu[MDIV] = 1.0 corresponds to the axis of symmetry */

/* s_gp[0] and mu[0] are not used by the program */

}
```

4 Loading the Equation of State

RNS needs a tabulated, zero-temperature equation of state (EOS), as an input, in order to run. You will have to format the EOS file in a specific way, as described here: The first line in the EOS file should contain the number of tabulated points. The remaining lines should consist of four columns - energy density (in gr/cm^3), pressure (in dynes/cm^2), enthalpy (in cm^2/s^2), and baryon number density (in cm^{-3}). The *enthalpy* is defined as

$$H(P) = \int_0^P \frac{c^2 dP}{(\epsilon + P)}, \tag{7}$$

where ϵ is the energy density, P is the pressure and c is the speed of light.

The number of points should be limited to 200. Example files (e.g. `eosC`) are supplied with the source code.

The function load_eos is called at the beginning of the application to load the specified equation of state file.

4.1 load_eos

```
/*************************************************************************/
/* Load EOS file.                                                        */
/*************************************************************************/
void load_eos( char eos_file[],
               double log_e_tab[201],
               double log_p_tab[201],
               double log_h_tab[201],
               double log_n0_tab[201],
               int *n_tab)
{
 int i;                    /* counter */

 double p,                 /* pressure */
        rho,               /* density */
        h,                 /* enthalpy */
        n0;                /* number density */
```

```
 FILE *f_eos;              /* pointer to eos_file */


    /* OPEN FILE TO READ */

    if((f_eos=fopen(eos_file,"r")) == NULL ) {
       printf("cannot open file:  %s\n",eos_file);
       exit(0);
    }


    /* READ NUMBER OF TABULATED POINTS */

    fscanf(f_eos,"%d\n",n_tab);


    /* READ EOS, H, NO AND MAKE THEM DIMENSIONLESS */

    for(i=1;i<=(*n_tab);i++) {
       fscanf(f_eos,"%lf %lf %lf %lf\n",&rho,&p,&h,&n0) ;
       log_e_tab[i]=log10(rho*C*C*KSCALE);     /* multiply by C^2 to get */
       log_p_tab[i]=log10(p*KSCALE);           /* energy density. */
       log_h_tab[i]=log10(h/(C*C));
       log_n0_tab[i]=log10(n0);
    }
}
```

5 Setting Program Defaults

There are some parameters used in the program whose default values need to be set. I will probably build their values into the routines at some unspecified later date. Until then you must include the following lines of code in your application:

```
double
   cf,  /* convergence factor */
   accuracy;

double
   e_surface,   /* value of energy density at star's surface */
   p_surface,   /* value of pressure at star's surface */
   enthalpy_min; /* minimum value of enthalpy */

  /* set program defaults */
  cf=1.0;
```

```
  accuracy=1e-5;

  if(strcmp(eos_type,"tab")==0) {
    e_surface=7.8*C*C*KSCALE;
    p_surface=1.01e8*KSCALE;
    enthalpy_min=1.0/(C*C);
  }
  else{
    e_surface=0.0;
    p_surface=0.0;
    enthalpy_min=0.0;
  }
```

6 Compute Pressure and Enthalpy at Star's Center

The user specifies the value of the star's energy density at the star's center. The application must then compute the values of pressure and enthalpy at the center of the star. This is done by calling the function make_center().

6.1 make_center()

```
void make_center(
       char eos_file[],
               double log_e_tab[201],
               double log_p_tab[201],
               double log_h_tab[201],
               double log_n0_tab[201],
               int n_tab,
       char eos_type[],
       double Gamma_P,
       double e_center,
       double *p_center,
       double *h_center)

{
 int n_nearest;

 double rho0_center;

 n_nearest=n_tab/2;

 if(strcmp(eos_type,"tab")==0) {

 /* If the EOS is tabulated, interpolate to find
```

```
    p_center and h_center given the value of e_center */

   (*p_center) = p_at_e( e_center, log_p_tab, log_e_tab, n_tab, &n_nearest);
   (*h_center) = h_at_p( (*p_center), log_h_tab, log_p_tab, n_tab, &n_nearest);

 }

 else {

 /* If the EOS is polytropic, use standard formulae to compute
    p_center and h_center */

       rho0_center = rtsec_G( e_of_rho0, Gamma_P, 0.0,e_center,DBL_EPSILON,
                              e_center );
       (*p_center) = pow(rho0_center,Gamma_P);
       (*h_center) = log((e_center+(*p_center))/rho0_center);
 }


}
```

7 Allocate Memory

Memory for the metric functions, energy density, pressure, etc. must be allocated at the beginning of the application. Each of these functions is two-dimensional and is represented by a `SDIV` $\times$ `MDIV` matrix. The metric functions are represented by the following matrices:

$$\rho \Leftrightarrow \text{rho} \tag{8}$$

$$\gamma \Leftrightarrow \text{gama} \quad \text{Note the spelling!} \tag{9}$$

$$\omega \Leftrightarrow \text{omega} \quad \text{Note the lower case o} \tag{10}$$

$$\alpha \Leftrightarrow \text{alpha} \tag{11}$$

Declare and allocate memory for these variables by including the following lines of code in your application:

```
double
**rho,
**gama,
**omega,
**alpha,
**energy,  /* The star's energy-density */
**pressure,  /* The star's pressure */
**enthalpy, /* The star's enthalpy */
**velocity_sq,  /* The square of the fluid's velocity */
```

```
*v_plus, /* vel. of co-rot. particle wrt ZAMO */
    *v_minus; /* vel. of counter-rot. ... */

  /* ALLLOCATE MEMORY  (Numerical Recipes routines)*/

  rho = dmatrix(1,SDIV,1,MDIV);
  gama = dmatrix(1,SDIV,1,MDIV);
  alpha = dmatrix(1,SDIV,1,MDIV);
  omega = dmatrix(1,SDIV,1,MDIV);

  energy = dmatrix(1,SDIV,1,MDIV);
  pressure = dmatrix(1,SDIV,1,MDIV);
  enthalpy = dmatrix(1,SDIV,1,MDIV);
  velocity_sq = dmatrix(1,SDIV,1,MDIV);

  v_plus = dvector(1,SDIV);
  v_minus = dvector(1,SDIV);
```

8 Compute a Spherical Star

Before computing a rotating neutron star, you will need to compute the form of a spherical neutron star. The spherical star will be used as a first approximation for the rotating neutron star.

The function sphere() is called by the application to compute the metric of a spherical star. The function sphere is given information about the equation of state and the values of energy density, pressure and enthalpy at the center and surface of the star. The function computes the coordinate radius of the star r_e as well as the metric functions ρ, γ, ω and α.

8.1 sphere()

```
void sphere(double s_gp[SDIV+1],
    double log_e_tab[201],
    double log_p_tab[201],
    double log_h_tab[201],
    double log_n0_tab[201],
    int n_tab,
    char eos_type[],
    double Gamma_P,
    double e_center,
    double p_center,
    double h_center,
    double p_surface,
    double e_surface,
    double **rho,
```

```
    double **gama,
    double **alpha,
    double **omega,
    double *r_e)

{
 int s,
     m,
     n_nearest;

 double r_is_s,
        r_is_final,
        r_final,
        m_final,
        lambda_s,
        nu_s,
        r_is_gp[RDIV+1],
        lambda_gp[RDIV+1],
        nu_gp[RDIV+1],
        gama_mu_0[SDIV+1],
        rho_mu_0[SDIV+1],
        gama_eq,
        rho_eq,
        s_e=0.5;

 /* The function TOV integrates the TOV equations. The function
can be found in the file equil.c */

 TOV(1, eos_type, e_center, p_center, p_surface, e_surface, Gamma_P,
              log_e_tab, log_p_tab, log_h_tab, n_tab, r_is_gp, lambda_gp,
              nu_gp, &r_is_final, &r_final, &m_final);

 TOV(2, eos_type, e_center, p_center, p_surface, e_surface, Gamma_P,
              log_e_tab, log_p_tab, log_h_tab, n_tab, r_is_gp, lambda_gp,
              nu_gp, &r_is_final, &r_final, &m_final);

 TOV(3, eos_type, e_center, p_center, p_surface, e_surface, Gamma_P,
              log_e_tab, log_p_tab, log_h_tab, n_tab, r_is_gp, lambda_gp,
              nu_gp, &r_is_final, &r_final, &m_final);



 n_nearest=RDIV/2;
 for(s=1;s<=SDIV;s++) {
    r_is_s=r_is_final*(s_gp[s]/(1.0-s_gp[s]));
```

```
    if(r_is_s<r_is_final) {
      lambda_s=interp(r_is_gp,lambda_gp,RDIV,r_is_s,&n_nearest);
      nu_s=interp(r_is_gp,nu_gp,RDIV,r_is_s,&n_nearest);
    }
    else {
      lambda_s=2.0*log(1.0+m_final/(2.0*r_is_s));
      nu_s=log((1.0-m_final/(2.0*r_is_s))/(1.0+m_final/(2*r_is_s)));
    }

    gama[s][1]=nu_s+lambda_s;
    rho[s][1]=nu_s-lambda_s;

    for(m=1;m<=MDIV;m++) {
        gama[s][m]=gama[s][1];
        rho[s][m]=rho[s][1];
        alpha[s][m]=(gama[s][1]-rho[s][1])/2.0;
        omega[s][m]=0.0;
    }

    gama_mu_0[s]=gama[s][1];                    /* gama at \mu=0 */
    rho_mu_0[s]=rho[s][1];                      /* rho at \mu=0 */

 }

   n_nearest=SDIV/2;
   gama_eq = interp(s_gp,gama_mu_0,SDIV,s_e,&n_nearest); /* gama at equator */
   rho_eq = interp(s_gp,rho_mu_0,SDIV,s_e,&n_nearest);   /* rho at equator */

   (*r_e)= r_final*exp(0.5*(rho_eq-gama_eq));

}
```

9 Integrate a Rotating Star

Once the spherical star's metric has been computed the application is ready to integrate the Einstein field equations for a rotating neutron star. Before doing so, the application must choose a value of the ratio of the star's polar radius to the equatorial radius. This ratio is denoted

`r_ratio`

Typically, one has to try many values of the ratio before the desired value of angular velocity, mass, etc is found. Once the value of the ratio has been set, call the function spin().

9.1 spin()

```
/*****************************************************************************/
/* Main iteration cycle for computation of the rotating star's metric    */
/*****************************************************************************/
void spin(double s_gp[SDIV+1],
  double mu[MDIV+1],
  double log_e_tab[201],
  double log_p_tab[201],
  double log_h_tab[201],
  double log_n0_tab[201],
  int n_tab,
  char eos_type[],
  double Gamma_P,
  double h_center,
  double enthalpy_min,
  double **rho,
  double **gama,
  double **alpha,
  double **omega,
  double **energy,
  double **pressure,
  double **enthalpy,
  double **velocity_sq,
  int    a_check,
  double accuracy,
  double cf,
  double r_ratio,
  double *r_e_new,
  double *Omega)

 {
 int m,                       /* counter */
     s,                       /* counter */
     n,                       /* counter */
     k,                       /* counter */
     n_of_it=0,               /* number of iterations */
     n_nearest,
     print_dif = 0,
     i,
     j;

double **D2_rho,
  **D2_gama,
  **D2_omega;
```

```
float  ***f_rho,
       ***f_gama;


double   sum_rho=0.0,         /* intermediate sum in eqn for rho */
 sum_gama=0.0,        /* intermediate sum in eqn for gama */
 sum_omega=0.0,       /* intermediate sum in eqn for omega */
         r_e_old,             /* equatorial radius in previus cycle */
     dif=1.0,             /* difference | r_e_old - r_e | */
         d_gama_s,            /* derivative of gama w.r.t. s */
         d_gama_m,            /* derivative of gama w.r.t. m */
         d_rho_s,             /* derivative of rho w.r.t. s */
         d_rho_m,             /* derivative of rho w.r.t. m */
         d_omega_s,           /* derivative of omega w.r.t. s */
         d_omega_m,           /* derivative of omega w.r.t. m */
         d_gama_ss,           /* 2nd derivative of gama w.r.t. s */
         d_gama_mm,           /* 2nd derivative of gama w.r.t. m */
         d_gama_sm,           /* derivative of gama w.r.t. m and s */
         temp1,                /* temporary term in da_dm */
         temp2,
         temp3,
         temp4,
         temp5,
         temp6,
         temp7,
         temp8,
         m1,
         s1,
         s2,
         ea,
         rsm,
         gsm,
         omsm,
         esm,
         psm,
         v2sm,
         mum,
         sgp,
         s_1,
         e_gsm,
         e_rsm,
         rho0sm,
         term_in_Omega_h,
         r_p,
         s_p,
```

```
      gama_pole_h,                  /* gama^hat at pole */
      gama_center_h,                /* gama^hat at center */
      gama_equator_h,               /* gama^hat at equator */
      rho_pole_h,                   /* rho^hat at pole */
      rho_center_h,                 /* rho^hat at center */
      rho_equator_h,                /* rho^hat at equator */
      omega_equator_h,              /* omega^hat at equator */
      gama_mu_1[SDIV+1],            /* gama at \mu=1 */
      gama_mu_0[SDIV+1],            /* gama at \mu=0 */
      rho_mu_1[SDIV+1],             /* rho at \mu=1 */
      rho_mu_0[SDIV+1],             /* rho at \mu=0 */
      omega_mu_0[SDIV+1],           /* omega at \mu=0 */
      s_e=0.5,
    **da_dm,
    **dgds,
    **dgdm,
    **D1_rho,
    **D1_gama,
    **D1_omega,
    **S_gama,
    **S_rho,
    **S_omega,
    **f2n,
    **P_2n,
    **P1_2n_1,
      Omega_h,
      sin_theta[MDIV+1],
      theta[MDIV+1],
      sk,
      sj,
      sk1,
      sj1,
      r_e;


f2n = dmatrix(1,LMAX+1,1,SDIV);
f_rho = f3tensor(1,SDIV,1,LMAX+1,1,SDIV);
f_gama = f3tensor(1,SDIV,1,LMAX+1,1,SDIV);

P_2n = dmatrix(1,MDIV,1,LMAX+1);
P1_2n_1 = dmatrix(1,MDIV,1,LMAX+1);


for(n=0;n<=LMAX;n++)
   for(i=2;i<=SDIV;i++) f2n[n+1][i] = pow((1.0-s_gp[i])/s_gp[i],2.0*n);
```

```
if(SMAX!=1.0) {

 for(j=2;j<=SDIV;j++)
    for(n=1;n<=LMAX;n++)
       for(k=2;k<=SDIV;k++) {
             sk=s_gp[k];
             sj=s_gp[j];
             sk1=1.0-sk;
             sj1=1.0-sj;

             if(k<j) {
                      f_rho[j][n+1][k] = f2n[n+1][j]*sj1/(sj*
                              f2n[n+1][k]*sk1*sk1);
                      f_gama[j][n+1][k] = f2n[n+1][j]/(f2n[n+1][k]*sk*sk1);
             }else {
                      f_rho[j][n+1][k] = f2n[n+1][k]/(f2n[n+1][j]*sk*sk1);
                      f_gama[j][n+1][k] = f2n[n+1][k]*sj1*sj1*sk/(sj*sj
                                        *f2n[n+1][j]*sk1*sk1*sk1);
             }
}
 j=1;

   n=0;
   for(k=2;k<=SDIV;k++) {
      sk=s_gp[k];
      f_rho[j][n+1][k]=1.0/(sk*(1.0-sk));
   }

   n=1;
   for(k=2;k<=SDIV;k++) {
      sk=s_gp[k];
      sk1=1.0-sk;
      f_rho[j][n+1][k]=0.0;
      f_gama[j][n+1][k]=1.0/(sk*sk1);
   }

   for(n=2;n<=LMAX;n++)
      for(k=1;k<=SDIV;k++) {
         f_rho[j][n+1][k]=0.0;
         f_gama[j][n+1][k]=0.0;
      }


 k=1;

   n=0;
```

```
    for(j=1;j<=SDIV;j++)
       f_rho[j][n+1][k]=0.0;

    for(j=1;j<=SDIV;j++)
       for(n=1;n<=LMAX;n++) {
          f_rho[j][n+1][k]=0.0;
          f_gama[j][n+1][k]=0.0;
       }


  n=0;
  for(j=2;j<=SDIV;j++)
     for(k=2;k<=SDIV;k++) {
            sk=s_gp[k];
            sj=s_gp[j];
            sk1=1.0-sk;
            sj1=1.0-sj;

            if(k<j)
              f_rho[j][n+1][k] = sj1/(sj*sk1*sk1);
            else
              f_rho[j][n+1][k] = 1.0/(sk*sk1);
   }

}
else{
     for(j=2;j<=SDIV-1;j++)
        for(n=1;n<=LMAX;n++)
           for(k=2;k<=SDIV-1;k++) {
              sk=s_gp[k];
              sj=s_gp[j];
              sk1=1.0-sk;
              sj1=1.0-sj;

              if(k<j) {
                       f_rho[j][n+1][k] = f2n[n+1][j]*sj1/(sj*
                                          f2n[n+1][k]*sk1*sk1);
                       f_gama[j][n+1][k] = f2n[n+1][j]/(f2n[n+1][k]*sk*sk1);
              }else {
                       f_rho[j][n+1][k] = f2n[n+1][k]/(f2n[n+1][j]*sk*sk1);

                       f_gama[j][n+1][k] = f2n[n+1][k]*sj1*sj1*sk/(sj*sj
                                           *f2n[n+1][j]*sk1*sk1*sk1);
              }
   }
```

```
j=1;

  n=0;
  for(k=2;k<=SDIV-1;k++) {
     sk=s_gp[k];
     f_rho[j][n+1][k]=1.0/(sk*(1.0-sk));
  }

  n=1;
  for(k=2;k<=SDIV-1;k++) {
     sk=s_gp[k];
     sk1=1.0-sk;
     f_rho[j][n+1][k]=0.0;
     f_gama[j][n+1][k]=1.0/(sk*sk1);
  }

  for(n=2;n<=LMAX;n++)
     for(k=1;k<=SDIV-1;k++) {
        f_rho[j][n+1][k]=0.0;
        f_gama[j][n+1][k]=0.0;
     }

k=1;

  n=0;
  for(j=1;j<=SDIV-1;j++)
     f_rho[j][n+1][k]=0.0;

  for(j=1;j<=SDIV-1;j++)
     for(n=1;n<=LMAX;n++) {
        f_rho[j][n+1][k]=0.0;
        f_gama[j][n+1][k]=0.0;
     }


n=0;
  for(j=2;j<=SDIV-1;j++)
     for(k=2;k<=SDIV-1;k++) {
        sk=s_gp[k];
        sj=s_gp[j];
        sk1=1.0-sk;
        sj1=1.0-sj;

        if(k<j)
          f_rho[j][n+1][k] = sj1/(sj*sk1*sk1);
        else
```

```
                f_rho[j][n+1][k] = 1.0/(sk*sk1);
            }

       j=SDIV;
         for(n=1;n<=LMAX;n++)
            for(k=1;k<=SDIV;k++) {
               f_rho[j][n+1][k] = 0.0;
               f_gama[j][n+1][k] = 0.0;
            }

       k=SDIV;
         for(j=1;j<=SDIV;j++)
             for(n=1;n<=LMAX;n++) {
                f_rho[j][n+1][k] = 0.0;
                f_gama[j][n+1][k] = 0.0;
             }
 }

n=0;
 for(i=1;i<=MDIV;i++)
    P_2n[i][n+1]=legendre(2*n,mu[i]);

 for(i=1;i<=MDIV;i++)
   for(n=1;n<=LMAX;n++) {
    P_2n[i][n+1]=legendre(2*n,mu[i]);
    P1_2n_1[i][n+1] = plgndr(2*n-1 ,1,mu[i]);
  }

free_dmatrix(f2n,1,LMAX+1,1,SDIV);


for(m=1;m<=MDIV;m++) {
   sin_theta[m] = sqrt(1.0-mu[m]*mu[m]);
   theta[m] = asin(sin_theta[m]);
}


r_e = (*r_e_new);

while(dif> accuracy || n_of_it<2) {

    if(print_dif!=0)
       printf("%4.3e\n",dif);


    /* Rescale potentials and construct arrays with the potentials along
```

```
 | the equatorial and polar directions.
*/

for(s=1;s<=SDIV;s++) {
   for(m=1;m<=MDIV;m++) {
      rho[s][m] /= SQ(r_e);
      gama[s][m] /= SQ(r_e);
      alpha[s][m] /= SQ(r_e);
      omega[s][m] *= r_e;
   }
   rho_mu_0[s]=rho[s][1];
   gama_mu_0[s]=gama[s][1];
   omega_mu_0[s]=omega[s][1];
   rho_mu_1[s]=rho[s][MDIV];
   gama_mu_1[s]=gama[s][MDIV];
}

/* Compute new r_e. */

r_e_old=r_e;
r_p=r_ratio*r_e;
s_p=r_p/(r_p+r_e);

n_nearest= SDIV/2;
gama_pole_h=interp(s_gp,gama_mu_1,SDIV,s_p,&n_nearest);
gama_equator_h=interp(s_gp,gama_mu_0,SDIV,s_e,&n_nearest);
gama_center_h=gama[1][1];

rho_pole_h=interp(s_gp,rho_mu_1,SDIV,s_p,&n_nearest);
rho_equator_h=interp(s_gp,rho_mu_0,SDIV,s_e,&n_nearest);
rho_center_h=rho[1][1];

r_e=sqrt(2*h_center/(gama_pole_h+rho_pole_h-gama_center_h-rho_center_h));


/* Compute angular velocity Omega. */

if(r_ratio==1.0) {
  Omega_h=0.0;
  omega_equator_h=0.0;
}
else {
      omega_equator_h=interp(s_gp,omega_mu_0,SDIV,s_e, &n_nearest);
      term_in_Omega_h=1.0-exp(SQ(r_e)*(gama_pole_h+rho_pole_h
                                       -gama_equator_h-rho_equator_h));
      if(term_in_Omega_h>=0.0)
```

```
          Omega_h = omega_equator_h + exp(SQ(r_e)*rho_equator_h)
                                        *sqrt(term_in_Omega_h);
        else {
            Omega_h=0.0;
}
  }


  /* Compute velocity, energy density and pressure. */

  n_nearest=n_tab/2;

  for(s=1;s<=SDIV;s++) {
     sgp=s_gp[s];

     for(m=1;m<=MDIV;m++) {
        rsm=rho[s][m];

        if(r_ratio==1.0)
            velocity_sq[s][m]=0.0;
        else
            velocity_sq[s][m]=SQ((Omega_h-omega[s][m])*(sgp/(1.0-sgp))
                              *sin_theta[m]*exp(-rsm*SQ(r_e)));

        if(velocity_sq[s][m]>=1.0)
          velocity_sq[s][m]=0.0;

        enthalpy[s][m]=enthalpy_min + 0.5*(SQ(r_e)*(gama_pole_h+rho_pole_h
                       -gama[s][m]-rsm)-log(1.0-velocity_sq[s][m]));

        if((enthalpy[s][m]<=enthalpy_min) || (sgp>s_e)) {
              pressure[s][m]=0.0;
              energy[s][m]=0.0;
}
        else { if(strcmp(eos_type,"tab")==0) {
                  pressure[s][m]=p_at_h(enthalpy[s][m], log_p_tab,
                                        log_h_tab, n_tab, &n_nearest);
                  energy[s][m]=e_at_p(pressure[s][m], log_e_tab,
                                      log_p_tab, n_tab, &n_nearest, eos_type,
                                       Gamma_P);
       }
                 else {
                       rho0sm=pow(((Gamma_P-1.0)/Gamma_P)
                              *(exp(enthalpy[s][m])-1.0),1.0/(Gamma_P-1.0));

                       pressure[s][m]=pow(rho0sm,Gamma_P);
```

```
                        energy[s][m]=pressure[s][m]/(Gamma_P-1.0)+rho0sm;
                }
        }

        /* Rescale back metric potentials (except omega) */

        rho[s][m] *= SQ(r_e);
        gama[s][m] *= SQ(r_e);
        alpha[s][m] *= SQ(r_e);
}
    }

    /* Compute metric potentials */

    S_gama = dmatrix(1,SDIV,1,MDIV);
    S_rho = dmatrix(1,SDIV,1,MDIV);
    S_omega = dmatrix(1,SDIV,1,MDIV);

    for(s=1;s<=SDIV;s++)
       for(m=1;m<=MDIV;m++) {
          rsm=rho[s][m];
          gsm=gama[s][m];
          omsm=omega[s][m];
          esm=energy[s][m];
          psm=pressure[s][m];
          e_gsm=exp(0.5*gsm);
          e_rsm=exp(-rsm);
          v2sm=velocity_sq[s][m];
          mum=mu[m];
          m1=1.0-SQ(mum);
          sgp=s_gp[s];
          s_1=1.0-sgp;
          s1=sgp*s_1;
          s2=SQ(sgp/s_1);

          ea=16.0*PI*exp(2.0*alpha[s][m])*SQ(r_e);

          if(s==1) {
            d_gama_s=0.0;
            d_gama_m=0.0;
            d_rho_s=0.0;
            d_rho_m=0.0;
            d_omega_s=0.0;
            d_omega_m=0.0;
          }else{
```

```
                d_gama_s=deriv_s(gama,s,m);
                d_gama_m=deriv_m(gama,s,m);
                d_rho_s=deriv_s(rho,s,m);
                d_rho_m=deriv_m(rho,s,m);
                d_omega_s=deriv_s(omega,s,m);
                d_omega_m=deriv_m(omega,s,m);
    }

            S_rho[s][m] = e_gsm*(0.5*ea*(esm + psm)*s2*(1.0+v2sm)/(1.0-v2sm)

                        + s2*m1*SQ(e_rsm)*(SQ(s1*d_omega_s)

                        + m1*SQ(d_omega_m))

                        + s1*d_gama_s - mum*d_gama_m + 0.5*rsm*(ea*psm*s2

                        - s1*d_gama_s*(0.5*s1*d_gama_s+1.0)

                        - d_gama_m*(0.5*m1*d_gama_m-mum)));

            S_gama[s][m] = e_gsm*(ea*psm*s2 + 0.5*gsm*(ea*psm*s2 - 0.5*SQ(s1

                         *d_gama_s) - 0.5*m1*SQ(d_gama_m)));

            S_omega[s][m]=e_gsm*e_rsm*( -ea*(Omega_h-omsm)*(esm+psm)

                        *s2/(1.0-v2sm) + omsm*( -0.5*ea*(((1.0+v2sm)*esm

                        + 2.0*v2sm*psm)/(1.0-v2sm))*s2

                        - s1*(2*d_rho_s+0.5*d_gama_s)

                        + mum*(2*d_rho_m+0.5*d_gama_m) + 0.25*SQ(s1)*(4

                        *SQ(d_rho_s)-SQ(d_gama_s)) + 0.25*m1*(4*SQ(d_rho_m)

                        - SQ(d_gama_m)) - m1*SQ(e_rsm)*(SQ(SQ(sgp)*d_omega_s)

                        + s2*m1*SQ(d_omega_m))));
}


     /* ANGULAR INTEGRATION */

     D1_rho = dmatrix(1,LMAX+1,1,SDIV);
```

```
      D1_gama = dmatrix(1,LMAX+1,1,SDIV);
      D1_omega = dmatrix(1,LMAX+1,1,SDIV);

      n=0;
      for(k=1;k<=SDIV;k++) {

         for(m=1;m<=MDIV-2;m+=2) {
               sum_rho += (DM/3.0)*(P_2n[m][n+1]*S_rho[k][m]
                         + 4.0*P_2n[m+1][n+1]*S_rho[k][m+1]
                         + P_2n[m+2][n+1]*S_rho[k][m+2]);
}

         D1_rho[n+1][k]=sum_rho;
         D1_gama[n+1][k]=0.0;
         D1_omega[n+1][k]=0.0;
         sum_rho=0.0;

      }

      for(n=1;n<=LMAX;n++)
         for(k=1;k<=SDIV;k++) {
            for(m=1;m<=MDIV-2;m+=2) {

               sum_rho += (DM/3.0)*(P_2n[m][n+1]*S_rho[k][m]
                         + 4.0*P_2n[m+1][n+1]*S_rho[k][m+1]
                         + P_2n[m+2][n+1]*S_rho[k][m+2]);

               sum_gama += (DM/3.0)*(sin((2.0*n-1.0)*theta[m])*S_gama[k][m]
                           +4.0*sin((2.0*n-1.0)*theta[m+1])*S_gama[k][m+1]
                           +sin((2.0*n-1.0)*theta[m+2])*S_gama[k][m+2]);

               sum_omega += (DM/3.0)*(sin_theta[m]*P1_2n_1[m][n+1]*S_omega[k][m]
                            +4.0*sin_theta[m+1]*P1_2n_1[m+1][n+1]*S_omega[k][m+1]
                            +sin_theta[m+2]*P1_2n_1[m+2][n+1]*S_omega[k][m+2]);
   }
            D1_rho[n+1][k]=sum_rho;
            D1_gama[n+1][k]=sum_gama;
            D1_omega[n+1][k]=sum_omega;
            sum_rho=0.0;
            sum_gama=0.0;
            sum_omega=0.0;
}


      free_dmatrix(S_gama,1,SDIV,1,MDIV);
      free_dmatrix(S_rho,1,SDIV,1,MDIV);
```

```
      free_dmatrix(S_omega,1,SDIV,1,MDIV);



      /* RADIAL INTEGRATION */

      D2_rho = dmatrix(1,SDIV,1,LMAX+1);
      D2_gama = dmatrix(1,SDIV,1,LMAX+1);
      D2_omega = dmatrix(1,SDIV,1,LMAX+1);



      n=0;
      for(s=1;s<=SDIV;s++) {
            for(k=1;k<=SDIV-2;k+=2) {
               sum_rho += (DS/3.0)*( f_rho[s][n+1][k]*D1_rho[n+1][k]
                          + 4.0*f_rho[s][n+1][k+1]*D1_rho[n+1][k+1]
                          + f_rho[s][n+1][k+2]*D1_rho[n+1][k+2]);
      }
   D2_rho[s][n+1]=sum_rho;
   D2_gama[s][n+1]=0.0;
   D2_omega[s][n+1]=0.0;
            sum_rho=0.0;
}


      for(s=1;s<=SDIV;s++)
         for(n=1;n<=LMAX;n++) {
            for(k=1;k<=SDIV-2;k+=2) {
               sum_rho += (DS/3.0)*( f_rho[s][n+1][k]*D1_rho[n+1][k]
                          + 4.0*f_rho[s][n+1][k+1]*D1_rho[n+1][k+1]
                          + f_rho[s][n+1][k+2]*D1_rho[n+1][k+2]);

               sum_gama += (DS/3.0)*( f_gama[s][n+1][k]*D1_gama[n+1][k]
                           + 4.0*f_gama[s][n+1][k+1]*D1_gama[n+1][k+1]
                           + f_gama[s][n+1][k+2]*D1_gama[n+1][k+2]);

               if(k<s && k+2<=s)
                 sum_omega += (DS/3.0)*( f_rho[s][n+1][k]*D1_omega[n+1][k]
                              + 4.0*f_rho[s][n+1][k+1]*D1_omega[n+1][k+1]
                              + f_rho[s][n+1][k+2]*D1_omega[n+1][k+2]);
               else {
                 if(k>=s)
                   sum_omega += (DS/3.0)*( f_gama[s][n+1][k]*D1_omega[n+1][k]
                                + 4.0*f_gama[s][n+1][k+1]*D1_omega[n+1][k+1]
                                + f_gama[s][n+1][k+2]*D1_omega[n+1][k+2]);
```

```
                else
                  sum_omega += (DS/3.0)*( f_rho[s][n+1][k]*D1_omega[n+1][k]
                              + 4.0*f_rho[s][n+1][k+1]*D1_omega[n+1][k+1]
                              + f_gama[s][n+1][k+2]*D1_omega[n+1][k+2]);
              }
   }
   D2_rho[s][n+1]=sum_rho;
   D2_gama[s][n+1]=sum_gama;
   D2_omega[s][n+1]=sum_omega;
           sum_rho=0.0;
           sum_gama=0.0;
           sum_omega=0.0;
}

     free_dmatrix(D1_rho,1,LMAX+1,1,SDIV);
     free_dmatrix(D1_gama,1,LMAX+1,1,SDIV);
     free_dmatrix(D1_omega,1,LMAX+1,1,SDIV);


     /* SUMMATION OF COEFFICIENTS */

     for(s=1;s<=SDIV;s++)
        for(m=1;m<=MDIV;m++) {

           gsm=gama[s][m];
           rsm=rho[s][m];
           omsm=omega[s][m];
           e_gsm=exp(-0.5*gsm);
           e_rsm=exp(rsm);
           temp1=sin_theta[m];

           sum_rho += -e_gsm*P_2n[m][0+1]*D2_rho[s][0+1];

           for(n=1;n<=LMAX;n++) {

              sum_rho += -e_gsm*P_2n[m][n+1]*D2_rho[s][n+1];

              if(m==MDIV) {
                sum_omega += 0.5*e_rsm*e_gsm*D2_omega[s][n+1];
                sum_gama += -(2.0/PI)*e_gsm*D2_gama[s][n+1];
       }
              else {
                    sum_omega += -e_rsm*e_gsm*(P1_2n_1[m][n+1]/(2.0*n
                                 *(2.0*n-1.0)*temp1))*D2_omega[s][n+1];

                    sum_gama += -(2.0/PI)*e_gsm*(sin((2.0*n-1.0)*theta[m])
```

```
                                /((2.0*n-1.0)*temp1))*D2_gama[s][n+1];
     }
  }

          rho[s][m]=rsm + cf*(sum_rho-rsm);
          gama[s][m]=gsm + cf*(sum_gama-gsm);
          omega[s][m]=omsm + cf*(sum_omega-omsm);

          sum_omega=0.0;
          sum_rho=0.0;
          sum_gama=0.0;
}

    free_dmatrix(D2_rho,1,SDIV,1,LMAX+1);
    free_dmatrix(D2_gama,1,SDIV,1,LMAX+1);
    free_dmatrix(D2_omega,1,SDIV,1,LMAX+1);


    /* CHECK FOR DIVERGENCE */

    if(fabs(omega[2][1])>100.0 || fabs(rho[2][1])>100.0
       || fabs(gama[2][1])>300.0) {
       a_check=200;
       break;
    }


    /* TREAT SPHERICAL CASE */

    if(r_ratio==1.0) {
      for(s=1;s<=SDIV;s++)
         for(m=1;m<=MDIV;m++) {
            rho[s][m]=rho[s][1];
            gama[s][m]=gama[s][1];
            omega[s][m]=0.0;
 }
    }


    /* TREAT INFINITY WHEN SMAX=1.0 */

    if(SMAX==1.0) {
       for(m=1;m<=MDIV;m++) {
          rho[SDIV][m]=0.0;
          gama[SDIV][m]=0.0;
          omega[SDIV][m]=0.0;
```

```
}
     }


     /* COMPUTE FIRST ORDER DERIVATIVES OF GAMA */


     da_dm = dmatrix(1,SDIV,1,MDIV);
     dgds = dmatrix(1,SDIV,1,MDIV);
     dgdm = dmatrix(1,SDIV,1,MDIV);

     for(s=1;s<=SDIV;s++)
        for(m=1;m<=MDIV;m++) {
           dgds[s][m]=deriv_s(gama,s,m);
           dgdm[s][m]=deriv_m(gama,s,m);
}



     /* ALPHA */

     if(r_ratio==1.0) {
       for(s=1;s<=SDIV;s++)
          for(m=1;m<=MDIV;m++)
             da_dm[s][m]=0.0;
     }
     else {
           for(s=2;s<=SDIV;s++)
              for(m=1;m<=MDIV;m++) {

                 da_dm[1][m]=0.0;

                 sgp=s_gp[s];
                 s1=sgp*(1.0-sgp);
                 mum=mu[m];
                 m1=1.0-SQ(mum);

                 d_gama_s=dgds[s][m];
                 d_gama_m=dgdm[s][m];
                 d_rho_s=deriv_s(rho,s,m);
                 d_rho_m=deriv_m(rho,s,m);
                 d_omega_s=deriv_s(omega,s,m);
                 d_omega_m=deriv_m(omega,s,m);
                 d_gama_ss=s1*deriv_s(dgds,s,m)+(1.0-2.0*sgp)
                                               *d_gama_s;
                 d_gama_mm=m1*deriv_m(dgdm,s,m)-2.0*mum*d_gama_m;
```

```
                d_gama_sm=deriv_sm(gama,s,m);

          temp1=2.0*SQ(sgp)*(sgp/(1.0-sgp))*m1*d_omega_s*d_omega_m

               *(1.0+s1*d_gama_s) - (SQ(SQ(sgp)*d_omega_s) -

               SQ(sgp*d_omega_m/(1.0-sgp))*m1)*(-mum+m1*d_gama_m);

          temp2=1.0/(m1 *SQ(1.0+s1*d_gama_s) + SQ(-mum+m1*d_gama_m));

          temp3=s1*d_gama_ss + SQ(s1*d_gama_s);

          temp4=d_gama_m*(-mum+m1*d_gama_m);

          temp5=(SQ(s1*(d_rho_s+d_gama_s)) - m1*SQ(d_rho_m+d_gama_m))

                *(-mum+m1*d_gama_m);

          temp6=s1*m1*(0.5*(d_rho_s+d_gama_s)* (d_rho_m+d_gama_m)

               + d_gama_sm + d_gama_s*d_gama_m)*(1.0 + s1*d_gama_s);

          temp7=s1*mum*d_gama_s*(1.0+s1*d_gama_s);

          temp8=m1*exp(-2*rho[s][m]);

         da_dm[s][m] = -0.5*(d_rho_m+d_gama_m) - temp2*(0.5*(temp3 -

           d_gama_mm - temp4)*(-mum+m1*d_gama_m) + 0.25*temp5

           - temp6 +temp7 + 0.25*temp8*temp1);
     }
}

   for(s=1;s<=SDIV;s++) {
      alpha[s][1]=0.0;
      for(m=1;m<=MDIV-1;m++)
         alpha[s][m+1]=alpha[s][m]+0.5*DM*(da_dm[s][m+1]+
                       da_dm[s][m]);
   }


free_dmatrix(da_dm,1,SDIV,1,MDIV);
free_dmatrix(dgds,1,SDIV,1,MDIV);
free_dmatrix(dgdm,1,SDIV,1,MDIV);
```

```
      for(s=1;s<=SDIV;s++)
         for(m=1;m<=MDIV;m++) {
            alpha[s][m] += -alpha[s][MDIV]+0.5*(gama[s][MDIV]-rho[s][MDIV]);

            if(alpha[s][m]>=300.0) {
              a_check=200;
              break;
            }
            omega[s][m] /= r_e;
         }

      if(SMAX==1.0) {
         for(m=1;m<=MDIV;m++)
            alpha[SDIV][m] = 0.0;
      }

      if(a_check==200)
        break;

      dif=fabs(r_e_old-r_e)/r_e;
      n_of_it++;

 }   /* end while */



     /* COMPUTE OMEGA */

     if(strcmp(eos_type,"tab")==0)
         (*Omega) = Omega_h*C/(r_e*sqrt(KAPPA));
     else
         (*Omega) = Omega_h/r_e;


    /* UPDATE r_e_new */

    (*r_e_new) = r_e;


    free_f3tensor(f_rho, 1,SDIV,1,LMAX+1,1,SDIV);
    free_f3tensor(f_gama,1,SDIV,1,LMAX+1,1,SDIV);
    free_dmatrix(P_2n,   1,MDIV,1,LMAX+1);
    free_dmatrix(P1_2n_1,1,MDIV,1,LMAX+1);
}
```

10 Compute Equilibrium Quantities

Once the star's metric has been computed, the application can compute various equilibrium quantities, such as the star's gravitational mass, baryonic mass, angular momentum, radius, etc. This is done by calling the function mass_radius().

10.1 mass_radius()

```
/*****************************************************************/
/* Computes the gravitational mass, equatorial radius, angular momentum
 * of the star
 *  and the velocity of co- and counter-rotating particles
 * with respect to a ZAMO                                        */
/*****************************************************************/
void mass_radius(
 double s_gp[SDIV+1],
 double mu[MDIV+1],
 double log_e_tab[201],
 double log_p_tab[201],
 double log_h_tab[201],
 double log_n0_tab[201],
 int n_tab,
 char eos_type[],
 double Gamma_P,
 double **rho,
 double **gama,
 double **alpha,
 double **omega,
 double **energy,
 double **pressure,
 double **enthalpy,
 double **velocity_sq,
                 double r_ratio,
 double e_surface,
                 double r_e,
                 double Omega,
                 double *Mass,
 double *Mass_0,
 double *ang_mom,
                 double *R_e,
 double *v_plus,
 double *v_minus,
 double *Omega_K)

{
```

```
int s,
    m,
    n_nearest;


double
  **rho_0, /*rest mass density*/
  **velocity,
  gama_equator,              /* gama at equator */
  rho_equator,               /* rho at equator */
  omega_equator,             /* omega at equator */
  s1,
  s_1,
  d_gama_s,
  d_rho_s,
  d_omega_s,
  sqrt_v,
  D_m[SDIV+1],               /* int. quantity for M */
  D_m_0[SDIV+1],             /* int. quantity for M_0 */
  D_J[SDIV+1],               /* int. quantity for J */
  s_e,
  d_o_e[SDIV+1],
  d_g_e[SDIV+1],
  d_r_e[SDIV+1],
  d_v_e[SDIV+1],
  doe,
  dge,
  dre,
  dve,
  vek,
  gama_mu_0[SDIV+1],
  rho_mu_0[SDIV+1],
  J,
  r_p,
  s_p;


  r_p= r_ratio*r_e;                              /* radius at pole */
  s_p= r_p/(r_p+r_e);                            /* s-coordinate at pole */
  s_e=0.5;

  rho_0 = dmatrix(1,SDIV,1,MDIV);
  velocity = dmatrix(1,SDIV,1,MDIV);

  for(s=1;s<=SDIV;s++) {
     gama_mu_0[s]=gama[s][1];
```

```
      rho_mu_0[s]=rho[s][1];
   }

   n_nearest= SDIV/2;
   gama_equator=interp(s_gp,gama_mu_0,SDIV,s_e, &n_nearest);
   rho_equator=interp(s_gp,rho_mu_0,SDIV,s_e, &n_nearest);


/* Circumferential radius */

   if(strcmp(eos_type,"tab")==0)
     (*R_e) = sqrt(KAPPA)*r_e*exp((gama_equator-rho_equator)/2.0);
   else
     (*R_e) = r_e*exp((gama_equator-rho_equator)/2.0);

 /* Masses and angular momentum */

   (*Mass) = 0.0;              /* initialize */
   (*Mass_0) = 0.0;
   J=0.0;

   /* CALCULATE THE REST MASS DENSITY */
 if(strcmp(eos_type,"tab")==0) {
   n_nearest=n_tab/2;
   for(s=1;s<=SDIV;s++)
      for(m=1;m<=MDIV;m++) {
           if(energy[s][m]>e_surface)
             rho_0[s][m]=n0_at_e(energy[s][m], log_n0_tab, log_e_tab, n_tab,
                                          &n_nearest)*MB*KSCALE*SQ(C);
           else
             rho_0[s][m]=0.0;
      }
 }else {
        for(s=1;s<=SDIV;s++)
           for(m=1;m<=MDIV;m++)
                rho_0[s][m]=(energy[s][m]+pressure[s][m])*exp(-enthalpy[s][m]);
  }

   for(s=1;s<=SDIV;s++) {
    D_m[s]=0.0;           /* initialize */
    D_m_0[s]=0.0;
    D_J[s]=0.0;

    for(m=1;m<=MDIV-2;m+=2) {
     D_m[s] += (1.0/(3.0*(MDIV-1)))*( exp(2.0*alpha[s][m]+gama[s][m])*
               (((energy[s][m]+pressure[s][m])/(1.0-velocity_sq[s][m]))*
```

```
            (1.0+velocity_sq[s][m]+(2.0*s_gp[s]*sqrt(velocity_sq[s][m])/
            (1.0-s_gp[s]))*sqrt(1.0-mu[m]*mu[m])*r_e*omega[s][m]*
            exp(-rho[s][m])) + 2.0*pressure[s][m])

          + 4.0*exp(2.0*alpha[s][m+1]+gama[s][m+1])*
            (((energy[s][m+1]+pressure[s][m+1])/(1.0-velocity_sq[s][m+1]))*
            (1.0+velocity_sq[s][m+1]+(2.0*s_gp[s]*sqrt(velocity_sq[s][m+1])/
            (1.0-s_gp[s]))*sqrt(1.0-mu[m+1]*mu[m+1])*r_e*omega[s][m+1]*
            exp(-rho[s][m+1])) + 2.0*pressure[s][m+1])

          + exp(2.0*alpha[s][m+2]+gama[s][m+2])*
            (((energy[s][m+2]+pressure[s][m+2])/(1.0-velocity_sq[s][m+2]))*
            (1.0+velocity_sq[s][m+2]+(2.0*s_gp[s]*sqrt(velocity_sq[s][m+2])/
            (1.0-s_gp[s]))*sqrt(1.0-mu[m+2]*mu[m+2])*r_e*omega[s][m+2]*
            exp(-rho[s][m+2])) + 2.0*pressure[s][m+2]));

  D_m_0[s] += (1.0/(3.0*(MDIV-1)))*( exp(2.0*alpha[s][m]+(gama[s][m]
           -rho[s][m])/2.0)*rho_0[s][m]/sqrt(1.0-velocity_sq[s][m])

           + 4.0* exp(2.0*alpha[s][m+1]+(gama[s][m+1]
           -rho[s][m+1])/2.0)*rho_0[s][m+1]/sqrt(1.0-velocity_sq[s][m+1])

           + exp(2.0*alpha[s][m+2]+(gama[s][m+2]
           -rho[s][m+2])/2.0)*rho_0[s][m+2]/sqrt(1.0-velocity_sq[s][m+2]));


  D_J[s] += (1.0/(3.0*(MDIV-1)))*( sqrt(1.0-mu[m]*mu[m])*
            exp(2.0*alpha[s][m]+gama[s][m]-rho[s][m])*(energy[s][m]
            +pressure[s][m])*sqrt(velocity_sq[s][m])/(1.0-velocity_sq[s][m])

            +4.0*sqrt(1.0-mu[m+1]*mu[m+1])*
            exp(2.0*alpha[s][m+1]+gama[s][m+1]-rho[s][m+1])*(energy[s][m+1]
            +pressure[s][m+1])*sqrt(velocity_sq[s][m+1])/
            (1.0-velocity_sq[s][m+1])

            + sqrt(1.0-mu[m+2]*mu[m+2])*
            exp(2.0*alpha[s][m+2]+gama[s][m+2]-rho[s][m+2])*(energy[s][m+2]
            +pressure[s][m+2])*sqrt(velocity_sq[s][m+2])/
            (1.0-velocity_sq[s][m+2]));
 }
}

 for(s=1;s<=SDIV-2;s+=2) {
  (*Mass) += (SMAX/(3.0*(SDIV-1)))*(pow(sqrt(s_gp[s])/(1.0-s_gp[s]),4.0)*
       D_m[s]+4.0*pow(sqrt(s_gp[s+1])/(1.0-s_gp[s+1]),4.0)*D_m[s+1]
       +pow(sqrt(s_gp[s+2])/(1.0-s_gp[s+2]),4.0)*D_m[s+2]);
```

```
     (*Mass_0) += (SMAX/(3.0*(SDIV-1)))*(pow(sqrt(s_gp[s])/(1.0-s_gp[s]),4.0)*
          D_m_0[s]+4.0*pow(sqrt(s_gp[s+1])/(1.0-s_gp[s+1]),4.0)*D_m_0[s+1]
          +pow(sqrt(s_gp[s+2])/(1.0-s_gp[s+2]),4.0)*D_m_0[s+2]);

     J += (SMAX/(3.0*(SDIV-1)))*((pow(s_gp[s],3.0)/pow(1.0-s_gp[s],5.0))*
          D_J[s]+ 4.0*(pow(s_gp[s+1],3.0)/pow(1.0-s_gp[s+1],5.0))*
          D_J[s+1] + (pow(s_gp[s+2],3.0)/pow(1.0-s_gp[s+2],5.0))*
          D_J[s+2]);

    }

    if(strcmp(eos_type,"tab")==0) {
      (*Mass) *= 4*PI*sqrt(KAPPA)*C*C*pow(r_e,3.0)/G;
      (*Mass_0) *= 4*PI*sqrt(KAPPA)*C*C*pow(r_e,3.0)/G;
    }
    else {
          (*Mass) *= 4*PI*pow(r_e,3.0);
          (*Mass_0) *= 4*PI*pow(r_e,3.0);
    }

    if(r_ratio==1.0)
         J=0.0;
    else {
          if(strcmp(eos_type,"tab")==0)
              J *= 4*PI*KAPPA*C*C*C*pow(r_e,4.0)/G;
          else
              J *= 4*PI*pow(r_e,4.0);
    }

    (*ang_mom) = J;


  /* Compute the velocities of co-rotating and counter-rotating particles
with respect to a ZAMO  */

  for(s=1+(SDIV-1)/2;s<=SDIV;s++) {
    s1= s_gp[s]*(1.0-s_gp[s]);
    s_1=1.0-s_gp[s];

    d_gama_s=deriv_s(gama,s,1);
    d_rho_s=deriv_s(rho,s,1);
    d_omega_s=deriv_s(omega,s,1);

    sqrt_v= exp(-2.0*rho[s][1])*r_e*r_e*pow(s_gp[s],4.0)*pow(d_omega_s,2.0)
            + 2*s1*(d_gama_s+d_rho_s)+s1*s1*(d_gama_s*d_gama_s-d_rho_s*d_rho_s);
```

```
    if(sqrt_v>0.0) sqrt_v= sqrt(sqrt_v);
     else {
      sqrt_v=0.0;
     }

    v_plus[s]=(exp(-rho[s][1])*r_e*s_gp[s]*s_gp[s]*d_omega_s + sqrt_v)/
              (2.0+s1*(d_gama_s-d_rho_s));

    v_minus[s]=(exp(-rho[s][1])*r_e*s_gp[s]*s_gp[s]*d_omega_s - sqrt_v)/
               (2.0+s1*(d_gama_s-d_rho_s));
  }


/* Kepler angular velocity */

   for(s=1;s<=SDIV;s++) {
     d_o_e[s]=deriv_s(omega,s,1);
     d_g_e[s]=deriv_s(gama,s,1);
     d_r_e[s]=deriv_s(rho,s,1);
     d_v_e[s]=deriv_s(velocity,s,1);
   }

   n_nearest=SDIV/2;
   doe=interp(s_gp,d_o_e,SDIV,s_e, &n_nearest);
   dge=interp(s_gp,d_g_e,SDIV,s_e, &n_nearest);
   dre=interp(s_gp,d_r_e,SDIV,s_e, &n_nearest);
   dve=interp(s_gp,d_v_e,SDIV,s_e, &n_nearest);

  vek=(doe/(8.0+dge-dre))*r_e*exp(-rho_equator) + sqrt(((dge+dre)/(8.0+dge
        -dre)) + pow((doe/(8.0+dge-dre))*r_e*exp(-rho_equator),2.0));

  if(strcmp(eos_type,"tab")==0)
    (*Omega_K) = (C/sqrt(KAPPA))*(omega_equator+vek*exp(rho_equator)/r_e);
  else
    (*Omega_K) = omega_equator + vek*exp(rho_equator)/r_e;


   free_dmatrix(velocity,1,SDIV,1,MDIV);
   free_dmatrix(rho_0,1,SDIV,1,MDIV);

}
```

11 Example Program

```
/*****************************************************************************
*                              MAIN.C                                        *
*                                                                            *
*       This is a sample program which uses the routines
*       written by Nikolaos Stergioulas.
*
* This sample program is meant merely to serve as an
* example of how to use the functions!
*
* In this example, the user specifies the star's equation of state,
* and the central energy density. The program computes models with
* increasing angular velocity until the star is spinning with
* the same angular velocity as a particle orbiting the star
* at its equator. The last star computed will be spinning with
* the maximum allowed angular velocity for a star with the given
* central energy density.
*
* Note that there is no guarantee that any of the intermediate
* stars are stable to radial perturbations. Also there is no
* guarantee that given any energy density, there will be a
* stable rotating solution. As a rule of thumb, the highest
* energy density you should use is the value which gives the
* maximum mass nonrotating star.
*
* It would be a good idea to read some of the papers on rapidly
* rotating neutron stars (such as given on the rns homepage)
* before embarking on a study of rotating neutron stars
* using these routines.
*
* For example, a reasonable model would use the file eosA,
* central energy density = 10^{15} g/cm^3
* To specify these parameters, run the executable:

kepler -f eosA -e 1e15

* This program (compiled on the "standard" setting)
* requires about 2.7 MBytes of RAM and took about 2 minutes to run.
*                                                                            *
*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stddef.h>
```

```
#include <stdlib.h>
#include "consts.h"
#include "nrutil.h"
#include "equil.h"

/*****************************************************************************
 * printing routine
 *****************************************************************************/

void print(double e_center, double Mass, double Mass_0, double R_e,
   double Omega, double Omega_K, double J
   )
{

  double I_45;

  if( Omega == 0.0) I_45 = 0.0;
  else I_45 = J/(Omega*1.0e45);

  printf(
 "%4.4f \t%4.2f \t%4.2f \t%4.1f \t%4.1f \t%4.1f \t%4.1f \t%4.3f \n",
 e_center,
 Mass/MSUN,
 Mass_0/MSUN,
 R_e/1.0e5,
 Omega,
 Omega_K,
 I_45,
 ( C*J/(G*Mass*Mass)))
    ;
}

/*****************************************************************************/
/* Main program.                                                             */
/*****************************************************************************/
int main(int argc,                    /* Number of command line arguments */
         char **argv)                 /* Command line arguments */
{


 /* EQUILIBRIUM VARIABLES */

 int    n_tab,                        /* Number of points in EOS file */
        print_dif=0;                    /* if =1, monitor convergence */

 int i,
```

```
    j,
    k,
    s,
    m
  ;

double log_e_tab[201],              /* energy density/c^2 in tabulated EOS */
  log_p_tab[201],               /* pressure in tabulated EOS */
  log_h_tab[201],               /* enthalpy in EOS file */
  log_n0_tab[201],              /* number density in EOS file */
  e_center,                     /* central en. density */
  p_center,                     /* central pressure */
  h_center,                     /* central enthalpy */
  n_P,                          /* Polytropic index N */
  Gamma_P,                      /* Gamma for polytropic EOS */
  r_ratio,                      /* axis ratio */
  s_gp[SDIV+1],                 /* s grid points */
  mu[MDIV+1],                   /* \mu grid points */
  **rho,                          /* potential \rho */
  **gama,                         /* potential \gamma */
  **omega,                        /* potential \omega */
  **alpha,                        /* potential \alpha */
  Mass,                         /* Gravitational mass */
  e_surface,                    /* surface en. density */
  p_surface,                    /* surface pressure */
  enthalpy_min,                 /* minimum enthalpy in EOS */
  **energy,                       /* energy density \epsilon */
  **pressure,                     /* pressure */
  **enthalpy,                     /* enthalpy */
  **velocity_sq,                  /* square of velocity */
  Mass_0,                         /* Baryon Mass */
  Omega,    /* Angular Velocity */
  J, /* Angular Momentum */
  R_e,                          /* Circumferential radius at equator */
  *v_plus, /* vel. of co-rot. particle wrt ZAMO */
  *v_minus, /* vel. of counter-rot. ... */
  Omega_K, /* Keplerian velocity of particle orbiting at equator */
  r_e                         /* coord. radius at equator  */
  ;

double
  cf=1, /* convergence factor */
  accuracy;

/* Definitions used to find the interval containing the
   correct spin frequency */
```

```
 double dr,
   diff_Omega,
   old_diff_Omega,
   a_check;

 /* Definitions used in the Ridder zero finding method */

 float ans, fh,fl,fm,fnew,sroot,xh,xl,xm,xnew,xacc,x1,x2;


  char eos_file[80] = "no EOS file specified";   /* EOS file name */
  char eos_type[80] = "tab";                    /* EOS type (poly or tab) */


  /* READ IN THE COMMAND LINE OPTIONS */

  for(i=1;i<argc;i++)
    if(argv[i][0]=='-'){
      switch(argv[i][1]){

      case 'q':
/* CHOOSE THE EOS TYPE: EITHER "tab" or "poly"
   (default is tab) */
sscanf(argv[i+1],"%s",eos_type);
break;

      case 'N':
/* IF A POLYTROPIC EOS WAS CHOSEN, CHOOSE THE
   POLYTROPIC INDEX "N" */
sscanf(argv[i+1],"%lf",&n_P);
Gamma_P=1.0+1.0/n_P;
break;

      case 'f':
/* IF A TABULATED EOS WAS CHOSEN, CHOOSE THE
   NAME OF THE FILE */
sscanf(argv[i+1],"%s",eos_file);
break;

      case 'e':
/* CHOOSE THE CENTRAL ENERGY DENSITY OF THE
   NEUTRON STAR (IN g/cm^3) */
sscanf(argv[i+1],"%lf",&e_center);
if(strcmp(eos_type,"tab")==0)
  e_center *= C*C*KSCALE;
```

```
break;

      case 'h':
fprintf(stderr,"\n");
fprintf(stderr,"Quick help:\n");
fprintf(stderr,"\n");
fprintf(stderr,"  -q EOS type (tab)\n");
fprintf(stderr,"     tab : tabulated \n");
fprintf(stderr,"     poly : analytic polytropic \n");
fprintf(stderr,"  -N polytropic index (P=K*e^(1+1/N))\n");
fprintf(stderr,"  -f EOS file \n");
fprintf(stderr,"  -e central energy density in gr/cm^3\n");
fprintf(stderr,"  -h this menu\n");
fprintf(stderr,"\n");
exit(1);
break;

      }
    }

  /* PRINT THE HEADER */

  printf("%s,  MDIVxSDIV=%dx%d\n",eos_file,MDIV,SDIV);
  printf("e_15\tM\tM_0\tr_star\tspin\tOmega_K\tI/M\tJ/M^2\n");
  printf("g/cm^3\tsun\tsun\tkm\ts-1\ts-1\tg cm^2\t\n");
  printf("\n");


  /* LOAD TABULATED EOS */

  if(strcmp(eos_type,"tab")==0)
    load_eos( eos_file, log_e_tab, log_p_tab, log_h_tab, log_n0_tab,
                                                    &n_tab );
  /* SET UP GRID */

  make_grid(s_gp, mu);

  /* ALLLOCATE MEMORY */

  rho = dmatrix(1,SDIV,1,MDIV);
  gama = dmatrix(1,SDIV,1,MDIV);
  alpha = dmatrix(1,SDIV,1,MDIV);
  omega = dmatrix(1,SDIV,1,MDIV);

  energy = dmatrix(1,SDIV,1,MDIV);
  pressure = dmatrix(1,SDIV,1,MDIV);
```

```
enthalpy = dmatrix(1,SDIV,1,MDIV);
velocity_sq = dmatrix(1,SDIV,1,MDIV);

v_plus = dvector(1,SDIV);
v_minus = dvector(1,SDIV);

/* set program defaults */
cf=1.0;
accuracy=1e-5;
xacc = 1e-4;

if(strcmp(eos_type,"tab")==0) {
  e_surface=7.8*C*C*KSCALE;
  p_surface=1.01e8*KSCALE;
  enthalpy_min=1.0/(C*C);
}
else{
  e_surface=0.0;
  p_surface=0.0;
  enthalpy_min=0.0;
}

/* CALCULATE THE PRESSURE AND ENTHALPY AT THE CENTRE OF THE STAR*/

make_center(eos_file, log_e_tab, log_p_tab,
    log_h_tab, log_n0_tab, n_tab,eos_type, Gamma_P,
    e_center, &p_center, &h_center);

/* COMPUTE A SPHERICAL STAR AS A FIRST GUESS FOR THE ROTATING STAR */

sphere( s_gp, log_e_tab, log_p_tab,
log_h_tab, log_n0_tab, n_tab, eos_type, Gamma_P,
e_center, p_center, h_center, p_surface,e_surface,
rho, gama, alpha, omega, &r_e);

r_ratio = 1.0;

/* THE PROCEDURE SPIN() WILL COMPUTE THE METRIC OF A STAR WITH
   GIVEN OBLATENESS. THE OBLATENESS IS SPECIFIED BY GIVING
   THE RATIO OF THE LENGTH OF THE AXIS CONNECTING THE CENTRE OF THE STAR
   TO ONE OF THE POLES TO THE RADIUS OF THE STAR'S EQUATOR.
   THIS RATIO IS NAMED r_ratio.
   WHEN r_ratio = 1.0, THE STAR IS SPHERICAL */

spin(s_gp, mu, log_e_tab, log_p_tab, log_h_tab, log_n0_tab,
     n_tab, eos_type, Gamma_P,
```

```
     h_center, enthalpy_min,
     rho, gama, alpha, omega, energy, pressure, enthalpy, velocity_sq,
     a_check, accuracy, cf,
     r_ratio, &r_e, &Omega);

/* THE METRIC FUNCTIONS ARE STORED IN THE FUNCTIONS
   alpha, rho, gama, omega (see user's manual for the definition
   of the metric */


/* COMPUTE THE VALUES OF VARIOUS EQUILIBRIUM QUANTITIES, SUCH
   AS MASS (Mass), RADIUS (R_e), BARYON MASS(Mass_0),
   ANGULAR MOMENTUM (J),
   KEPLERIAN ANGULAR VELOCITY OF PARTICLE ORBITING AT
   THE EQUATOR,
   VELOCITIES OF CO-ROTATING PARTICLES (v_plus),
   AND COUNTER-ROTATING PARTICLES (v_minus) */

mass_radius( s_gp, mu, log_e_tab, log_p_tab,
     log_h_tab, log_n0_tab, n_tab, eos_type, Gamma_P,
     rho, gama, alpha, omega,
     energy, pressure, enthalpy, velocity_sq,
     r_ratio, e_surface, r_e, Omega,
     &Mass, &Mass_0, &J, &R_e, v_plus, v_minus, &Omega_K);


/* PRINT OUT INFORMATION ABOUT THE STELLAR MODEL */

print(e_center, Mass, Mass_0, R_e, Omega, Omega_K, J);

dr=0.05;


/* THIS LOOP STARTS WITH A NON-ROTATING STAR AND INCREASES
   THE STAR'S OBLATENESS (BY DECREASING R_RATIO) AND
   THEN CALCULATES THE STAR'S ANGULAR VELOCITY. ONCE THE
   COMPUTED VALUE OF ANGULAR VELOCITY IS LARGER THAN
   THE ANGULAR VELOCITY OF A PARTICLE ORBITING THE STAR
   AT THE EQUATOR, (Omega_K), THE LOOP STOPS */

diff_Omega = Omega_K - Omega;
old_diff_Omega = diff_Omega;

while( diff_Omega >0){
  /* Find the interval of r_ratio where the star has the
     correct angular velocity */
```

```
   r_ratio -= dr;

   /* Compute the star with the specified value of r_ratio */

   spin(s_gp, mu, log_e_tab, log_p_tab, log_h_tab, log_n0_tab,
n_tab, eos_type, Gamma_P,
h_center, enthalpy_min,
rho, gama, alpha, omega, energy, pressure, enthalpy, velocity_sq,
a_check, accuracy, cf,
r_ratio, &r_e, &Omega);

   mass_radius( s_gp, mu, log_e_tab, log_p_tab,
log_h_tab, log_n0_tab, n_tab, eos_type, Gamma_P,
rho, gama, alpha, omega,
energy, pressure, enthalpy, velocity_sq,
r_ratio, e_surface, r_e, Omega,
&Mass, &Mass_0, &J, &R_e, v_plus, v_minus, &Omega_K);

   print(e_center, Mass, Mass_0, R_e, Omega, Omega_K, J);

   old_diff_Omega = diff_Omega;
   diff_Omega = Omega_K - Omega;

 }

 /* The correct star lies between r_ratio and r_ratio + dr */
 xl = r_ratio;
 xh = r_ratio + dr;
 fl = diff_Omega;
 fh = old_diff_Omega;

 /* Use Ridder's method to find the correct star (Taken from
    Numerical Recipes) */


 if ((fl > 0.0 && fh < 0.0) || (fl < 0.0 && fh > 0.0)) {
   ans=-1.11e30;
   for (j=1;j<=60;j++) {
     xm=0.5*(xl+xh);
     r_ratio = xm;

     spin(s_gp, mu, log_e_tab, log_p_tab, log_h_tab, log_n0_tab,
  n_tab, eos_type, Gamma_P,
  h_center, enthalpy_min,
  rho, gama, alpha, omega, energy, pressure, enthalpy, velocity_sq,
  a_check, accuracy, cf,
```

```
   r_ratio, &r_e, &Omega);

      mass_radius( s_gp, mu, log_e_tab, log_p_tab,
   log_h_tab, log_n0_tab, n_tab, eos_type, Gamma_P,
   rho, gama, alpha, omega,
   energy, pressure, enthalpy, velocity_sq,
   r_ratio, e_surface, r_e, Omega,
   &Mass, &Mass_0, &J, &R_e, v_plus, v_minus, &Omega_K);

      fm= Omega_K - Omega;
      sroot=sqrt(fm*fm-fl*fh);
      if (sroot == 0.0) {
r_ratio = ans;
break;
      }

      xnew=xm+(xm-xl)*((fl >= fh ? 1.0 : -1.0)*fm/sroot);
      if (fabs(xnew-ans) <= xacc) {
r_ratio = ans;
break;
      }
      ans=xnew;
      r_ratio = ans;

      spin(s_gp, mu, log_e_tab, log_p_tab, log_h_tab, log_n0_tab,
   n_tab, eos_type, Gamma_P,
   h_center, enthalpy_min,
   rho, gama, alpha, omega, energy, pressure, enthalpy, velocity_sq,
   a_check, accuracy, cf,
   r_ratio, &r_e, &Omega);

      mass_radius( s_gp, mu, log_e_tab, log_p_tab,
   log_h_tab, log_n0_tab, n_tab, eos_type, Gamma_P,
   rho, gama, alpha, omega,
   energy, pressure, enthalpy, velocity_sq,
   r_ratio, e_surface, r_e, Omega,
   &Mass, &Mass_0, &J, &R_e, v_plus, v_minus, &Omega_K);


      print(e_center, Mass, Mass_0, R_e, Omega, Omega_K, J);



      fnew =  Omega_K - Omega;
      if (fnew == 0.0){
r_ratio = ans;
```

```
break;
      }

      if (SIGN(fm,fnew) != fm) {
xl=xm;
fl=fm;
xh=ans;
fh=fnew;
      } else if (SIGN(fl,fnew) != fl) {
xh=ans;
fh=fnew;
      } else if (SIGN(fh,fnew) != fh) {
xl=ans;
fl=fnew;
      } else nrerror("never get here.");
      if (fabs(xh-xl) <= xacc){
r_ratio = ans;
break;
      }
    }
  }
  else {
    if (fh == 0.0){
      r_ratio +=dr;
    }
    nrerror("root must be bracketed in zriddr.");
  }

  /* THE RIDDER ZERO-FINDING ROUTINE HAS FOUND THE VALUE
     OF R_RATIO WHICH GIVES THE DESIRED STAR. */

}
```

12 References

Komatsu H., Eriguchi Y. & Hachisu I., 1989, *MNRAS*, **237**, 355
Cook G.B., Shapiro S.L. & Teukolsky S.A., 1994, *APJ*, **422**, 227
Stergioulas N. & Friedman J.L., 1995, *ApJ*, **444**, 306