


```

for (int i = 0; i < Y; ++i){

    n = pow(P/363.44, W); // <<
    rhos = 236 * pow(n, 2.54) + n*mn; // << Bethe & Johnson's EoS

    //n = pow(P * 0.012597402, 0.6);
    //rhos = 119.07216481 * pow(n, 5/3) + n*mn;
    // Non-relativistic non-interacting derived EoS

//Need to be careful about editing EoS... also change function call in EoS routine
    // ... RHOHAT()

    Phat = P/rhos;
    rhohat = 1; // <<
    mhat = 0; // <<Initial conditions
    rhat = 0; // <<

    RK4 (h, rhat, Phat, mhat, P, rhos, out2); // RK4 calls main programme loop

    cout << rhos;

    if (rhos >= 6170) exit(1);
    else P = P + 1;
    // Condition to stop programme once
    // upper-bound on range of validity of
    // EoS reached
} // end for()

out2.close(); // close output file

} // end main()

void RK4 (double h, double rhat, double Phat, double mhat, double P, double rhos,
ofstream& out2){

    double k1, k2, k3, k4, l1, l2, l3, l4, M01, R01;
    double rhat1, Phat1, mhat1, Phat2, rhat2, mhat2;
    double Phat3, rhat3, mhat3;
    double Phat4, rhat4, mhat4;
    M01 = M0(rhos);
    R01 = R0(rhos);
    Phat4 = 0;
    rhat4 = 0;
    mhat4 = 0;
    Phat3 = 0;
    rhat3 = 0;
    mhat3 = 0;
    Phat2 = 0;
    mhat2 = 0; // Initialising to zero for convenience
    rhat2 = 0;
    Phat1 = 0;
    mhat1 = 0;
    rhat1 = 0;

    for (int i = 0; i < MAX; ++i){

        Phat4 = Phat3;
        mhat4 = mhat3; // Stores "four values ago"
        rhat4 = rhat3;

        Phat3 = Phat2;
        mhat3 = mhat2; // Stores "three values ago"
        rhat3 = rhat2;
    }
}

```

```

Phat2 = Phat1;
mhat2 = mhat1; // Stores "two values ago"
rhat2 = rhat1;

Phat1 = Phat; // Store "last value"
mhat1 = mhat;
rhat1 = rhat;

//out2 <<RHOHAT(Phat, rhos)<<"\t"<<rhat*R01<<"\t"<<mhat*M01<<endl;
// Select this for a pressure profile
// of a single star, and set Y = 1 above

k1 = h * F(rhat, Phat, mhat, rhos);
l1 = h * G(rhat, Phat, mhat, rhos);
k2 = h * F(rhat + h/2, Phat + k1*0.5, mhat + l1*0.5, rhos);
l2 = h * G(rhat + h/2, Phat + k1*0.5, mhat + l1*0.5, rhos);
k3 = h * F(rhat + h/2, Phat + k2*0.5, mhat + l2*0.5, rhos);
l3 = h * G(rhat + h/2, Phat + k2*0.5, mhat + l2*0.5, rhos);
k4 = h * F(rhat + h, Phat + k3, mhat + l3, rhos);
l4 = h * G(rhat + h, Phat + k3, mhat + l3, rhos);
// Runge-Kutta 4th order steps

rhat = rhat + h;
Phat = Phat + (k1 + 2*k2 + 2*k3 + k4)/6;
mhat = mhat + (l1 + 2*l2 + 2*l3 + l4)/6;

P = Phat * rhos;

if ( !( P > 0 ) ) {

    extrapolation (Phat4, Phat3, Phat2, Phat1,
                  mhat4, mhat3, mhat2, mhat1,
                  rhat4, rhat3, rhat2, rhat1,
                  rhos, h, out2, i);
// Extrapolation routines

    break;

// Checks when edge of star reached & outputs values
} // end if()

} // end for()

} // end RK4

double F(double rhat, double Phat, double mhat, double rhos) {

    double rhohat;
    rhohat = RHOHAT(Phat, rhos);

    if (rhat == 0) return 0;

    //else return (-1)*(mhat * rhohat) / (rhat*rhat);
    // Classical equation

    else return - (Phat + rhohat)*(rhat*rhat*rhat*Phat + mhat)/(rhat*rhat -
2*mhat*rhat);

// TOV equation
} // end F()

double G(double rhat, double Phat, double mhat, double rhos){

    double rhohat;
    rhohat = RHOHAT(Phat, rhos);

```

```

    return rhat*rhat * rhohat;
} // end G()

double R0 (double rhos){
    return 245.841157 * pow(rhos, -0.5);
} // end R0()

double M0 (double rhos){
    return 167.3431835*pow(rhos, -0.5);
} // end M0()

double RHOHAT(double Phat, double rhos){
    double P, n, rho, rhohat;
    P = Phat * rhos;

    n = pow(P/363.44, W);
    rho = 236 * pow(n, 2.54) + n*mn;

    //n = pow(P * 0.012597402, 0.6);
    //rho = 119.07216481 * pow(n, 5/3) + n*mn;
    // Non-rel non-interacting derived EoS

    rhohat = rho / rhos;
    return rhohat;
} // end RHOHAT()

void extrapolation (double Phat4, double Phat3, double Phat2, double Phat1,
    double mhat4, double mhat3, double mhat2, double mhat1,
    double rhat4, double rhat3, double rhat2, double rhat1,
    double rhos, double h, ofstream& out2, int i){

    double M01, R01, rhat, mhat, rerror, merror;
    M01 = M0(rhos);
    R01 = R0(rhos);

    rhat = find_linear_roots(rhat1, Phat1, rhat2, Phat2, rhat3, Phat3, rhat4,
    Phat4);
    mhat = find_cubic_fit_mhat_use(rhat1, mhat1, rhat2, mhat2, rhat3, mhat3, rhat4,
    mhat4, rhat);

    if(rhat != -4 && mhat != -4) {

        //rerror = (pow(rhat1-rhat2, 2) + pow(h, 4))*R01;
        //merror = M01*pow(mhat1-mhat2, 2) + R01*pow(h, 4);
        // linear error

        //rerror = (pow(rhat1-rhat2, 3) + pow(h, 4))*R01;
        //merror = M01*pow(mhat1-mhat2, 3) + R01*pow(h, 4);
        // quadratic error

        rerror = (pow(rhat1-rhat2, 4) + pow(h, 4))*R01;
        merror = M01*pow(mhat1-mhat2, 4) + R01*pow(h, 4);
        // cubic error

        out2 << rhos
        <<"\t"<<rhat*R01<<"\t"<<mhat*M01<<"\t"<<rerror<<"\t"<<merror<<endl;

```

```

    }

    // The values xi are such that x1 is closest to the 'root' point, and x4 is
    //    4 values before it

} // end extrapolation routine

double find_linear_roots(double x1,double y1,double x2,double y2,double x3,double
y3,double x4,double y4){

    double answer;
    answer = -y1/((y2-y1)/(x2-x1)) + x1;
    return answer;

} // end find_linear_roots()

double find_quadratic_roots (double x1, double y1, double x2, double y2, double x3,
double y3, double x4, double y4){
int size = 3; //Matrix size
double answer = 2*x1;
double **A = matrix_make(size); //Original Matrix

A[0][0] = pow(x1, 2);
A[0][1] = x1;
A[0][2] = A[1][2] = A[2][2] = 1;

A[1][0] = pow(x2, 2);
A[1][1] = x2;

A[2][0] = pow(x3, 2);
A[2][1] = x3;

double det = matrix_det(A,size);

if ( det == 0.0 ) {
    return answer = -4;
    // if det is zero, do nothing, dont write to file
    exit (2);
}

else {
    double **B = matrix_make(size); //Inverse
    double a, b, c, discrim, root_1, root_2, x0;
    matrix_inverse(B,A,size,det); //Get inverse matrix
    a = B[0][0]*y1 + B[0][1]*y2 + B[0][2]*y3;
    b = B[1][0]*y1 + B[1][1]*y2 + B[1][2]*y3;
    c = B[2][0]*y1 + B[2][1]*y2 + B[2][2]*y3;

    discrim = b*b - 4*a*c;
    double grad = -b/(2*a);
    x0 = (x1-x2) + x1;
    if(discrim < 0){

        if (0 < grad - x1 && grad - x1 < answer-x1){
            answer = grad;

        }

    }

}
if (discrim >= 0 ){

```

```

    root_1 = (-b + sqrt(discrim))/(2*a);
    root_2 = (-b - sqrt(discrim))/(2*a);

    if (0 < root_1 - x1 && root_1 - x1 < answer-x1){
        answer = root_1;
    }
    if ( 0 < root_2 - x1 && root_2 - x1 < answer-x1){
        answer = root_2;
    }

    if (answer > x0 ){
        if (0 < grad-x1 && grad-x1 < answer-x1){
            answer = grad;
        }
    }

}
if (answer > x0 ){
    answer = -4;
    // This condition checks if a viable root has been found.
}
}
return answer;
} // end find_quadratic_roots()

double find_cubic_roots (double x1, double y1, double x2, double y2, double x3,
double y3, double x4, double y4){

double Coeff_1, Coeff_2, Coeff_3, Coeff_4;

        // This routine send 4 values to have a cubic fitted, then
        // solves the cubic, and returns the closest value to 'x1'

int size = 4; //Matrix size
double answer = 2*x1;
double **A = matrix_make(size); //Original Matrix

A[0][0] = pow(x1, 3);
A[0][1] = pow(x1, 2);
A[0][2] = x1;
A[0][3] = A[1][3] = A[2][3] = A[3][3] = 1;

A[1][0] = pow(x2, 3);
A[1][1] = pow(x2, 2);
A[1][2] = x2;

A[2][0] = pow(x3, 3);
A[2][1] = pow(x3, 2);
A[2][2] = x3;

A[3][0] = pow(x4, 3);
A[3][1] = pow(x4, 2);
A[3][2] = x4;

double det = matrix_det(A,size);

if ( det == 0.0 ) {
    return answer = -4;
    // If det is zero, do nothing, dont write to file
    exit (2);
}

```

```

}

else {
    double **B = matrix_make(size); //Inverse

    matrix_inverse(B,A,size,det); //Get inverse matrix

    Coeff_1 = B[0][0]*y1 + B[0][1]*y2 + B[0][2]*y3 + B[0][3]*y4;
    Coeff_2 = B[1][0]*y1 + B[1][1]*y2 + B[1][2]*y3 + B[1][3]*y4;
    Coeff_3 = B[2][0]*y1 + B[2][1]*y2 + B[2][2]*y3 + B[2][3]*y4;
    Coeff_4 = B[3][0]*y1 + B[3][1]*y2 + B[3][2]*y3 + B[3][3]*y4;
    // finds coefficients of cubic Coeff_i <=> x^i
    matrix_free(B,size);

} //End of if/else statement

matrix_free(A,size);

double x1re;
double x1im;
double x2re;
double x2im;
double x3re;
double x3im;

Coeff_2 /= Coeff_1;
Coeff_3 /= Coeff_1;
Coeff_4 /= Coeff_1;

double discrim;
double q;
double r;
double dum1;
double s;
double t;
double term1;
double r13;

q = (3.0*Coeff_3 - (Coeff_2*Coeff_2))/9.0;
r = -(27.0*Coeff_4) + Coeff_2*(9.0*Coeff_3 - 2.0*(Coeff_2*Coeff_2));
r /= 54.0;

discrim = q*q*q + r*r;
x1im = 0;
term1 = (Coeff_2/3.0);

if (discrim > 0) {
    s = r + sqrt(discrim);
    s = ((s < 0) ? -pow(-s, (1.0/3.0)) : pow(s, (1.0/3.0)));
    t = r - sqrt(discrim);
    t = ((t < 0) ? -pow(-t, (1.0/3.0)) : pow(t, (1.0/3.0)));
    x1re = -term1 + s + t;
    term1 += (s + t)/2.0;
    x3re = x2re = -term1;
    term1 = sqrt(3.0)*(-t + s)/2;
    x2im = term1;
    x3im = -term1;

    if (0 < x1re-x1 && x1re-x1 < answer-x1 && x1im == 0){
        answer = x1re;
    }
    if (0 < x2re-x1 && x2re-x1 < answer-x1 && x2im == 0){

```

```

        answer = x2re;
    }
    if ( 0 < x3re-x1 && x3re-x1 < answer-x1 && x3im == 0){
        answer = x3re;
    }

    double x0;
    x0= (x1-x2) + x1 ;
    if (answer > x0 ){

        double grad_1, grad_2;
        grad_1 = (-2*Coeff_2 + pow((2*Coeff_2*2*Coeff_2 -
4*3*Coeff_1*Coeff_3),0.5) )/(2*3*Coeff_1);
        grad_2 = (-2*Coeff_2 - pow((2*Coeff_2*2*Coeff_2 -
4*3*Coeff_1*Coeff_3),0.5) )/(2*3*Coeff_1);

        if (0 < grad_1-x1 && grad_1-x1 < answer-x1){
            answer = grad_1;
        }
        if (0 < grad_2-x1 && grad_2-x1 < answer-x1){
            answer = grad_2;
        }
    }
    if (answer > x0 ){
        answer = -4;
// This condition checks if a viable root has been found.
    } // if not, then nothing will be written to file.

    return answer;
}

x3im = x2im = 0;

if (discrim == 0){
    r13 = ((r < 0) ? -pow(-r,(1.0/3.0)) : pow(r,(1.0/3.0)));
    x1re = -term1 + 2.0*r13;
    x3re = x2re = -(r13 + term1);

    if (0 < x1re-x1 && x1re-x1 < answer-x1 && x1im == 0){
        answer = x1re;
    }
    if ( 0 < x2re-x1 && x2re-x1 < answer-x1 && x2im == 0){
        answer = x2re;
    }
    if ( 0 < x3re-x1 && x3re-x1 < answer-x1 && x3im == 0){
        answer = x3re;
    }

    double x0;
    x0= (x1-x2) + x1 ;
    if (answer > x0 ){

        double grad_1, grad_2;
        grad_1 = (-2*Coeff_2 + pow((2*Coeff_2*2*Coeff_2 -
4*3*Coeff_1*Coeff_3),0.5) )/(2*3*Coeff_1);
        grad_2 = (-2*Coeff_2 - pow((2*Coeff_2*2*Coeff_2 -
4*3*Coeff_1*Coeff_3),0.5) )/(2*3*Coeff_1);

        if (0 < grad_1-x1 && grad_1-x1 < answer-x1){
            answer = grad_1;
        }
        if (0 < grad_2-x1 && grad_2-x1 < answer-x1){

```

```

        answer = grad_2;
    }
    }
    if (answer > x0 ){
        answer = -4;
// This condition checks if a viable root has been found.
    } // if not, then nothing will be written to file.

    return answer;
}

q = -q;
dum1 = q*q*q;
dum1 = acos(r/sqrt(dum1));
r13 = 2.0*sqrt(q);
xlre = -term1 + r13*cos(dum1/3.0);
x2re = -term1 + r13*cos((dum1 + 2.0*pi)/3.0);
x3re = -term1 + r13*cos((dum1 + 4.0*pi)/3.0);

if (0 < xlre-x1 && xlre-x1 < answer-x1 && xlim == 0){
    answer = xlre;
}
if ( 0 < x2re-x1 && x2re-x1 < answer-x1 && x2im == 0){
    answer = x2re;
}
if ( 0 < x3re-x1 && x3re-x1 < answer-x1 && x3im == 0){
    answer = x3re;
}

double x0;

x0= (x1-x2) + x1;

if (answer > x0 ){

    double grad_1, grad_2;
    grad_1 = (-2*Coeff_2 + pow((2*Coeff_2*2*Coeff_2 -
4*3*Coeff_1*Coeff_3),0.5) )/(2*3*Coeff_1);
    grad_2 = (-2*Coeff_2 - pow((2*Coeff_2*2*Coeff_2 -
4*3*Coeff_1*Coeff_3),0.5) )/(2*3*Coeff_1);

    if (0 < grad_1-x1 && grad_1-x1 < answer-x1){
        answer = grad_1;
    }
    if (0 < grad_2-x1 && grad_2-x1 < answer-x1){
        answer = grad_2;
    }
}

if (answer > x0 ){
    answer = -4;
// This condition checks if a viable root has been found.
} // if not, then nothing will be written to file.

return answer;
} // end find_cubic_roots()

double** matrix_make(const int d) {

    double **M = new double *[d];
    for(int i = 0; i < d; i++)

```

```

        M[i] = new double[d];
    return M;
} // end make_matrix()

void matrix_free(double **M, const int d) {

    for(int i = 0; i < d; i++)
        delete [] M[i]; //Free matrix memory
    delete [] M;

} // end matrix_free()

double* array_make(const int d) {

    double *A = new double [d];
    return A; //Create array size d

} // end array_make()

void array_free(double *A, const int d) {

    delete [] A; //Free array memory

} // end array_free()

double matrix_det(double **M, const int d) {

    //Calculate determinant from top row of matrix

    if ( d <= 1 ) { //If matrix is 1x1, the determinant is its element
        return M[0][0];
    } else {
        double determinant = 0.0;
        double *c = array_make(d); //Array for top row of cofactor matrix

        matrix_cofactor_row(c,M,d); //Get cofactor row

        for ( int j = 0; j < d; j++ ) {
            determinant += M[0][j] * c[j];
        } //Multiply each row element by its cofactor
        array_free(c,d);
        return determinant;
    }

} // end matrix_det()

void matrix_minor(double **M2, double **M1, const int row, const int column, const
int d){

    int a = 0, b = 0;

    //delete row and column
    for ( int i = 0; i < d; i++ ) {
        if ( i != row ) {
            b = 0;
            for ( int j = 0; j < d; j++ ) {
                if ( j != column ) M2[a][b++] = M1[i][j];
            }
            a++;
        }
    }
}

```

```

} // end matrix_minor()

void matrix_cofactor_row(double *M2, double **M1, const int d) {

    double **temp = matrix_make(d-1);
    for ( int j = 0; j < d; j++ ) {
        matrix_minor(temp, M1, 0, j, d);
        M2[j] = matrix_det(temp,d-1) * pow(-1.0,j);
    }

} //end matrix_cofactor_row()

void matrix_cofactor(double **M2, double **M1, const int d) {

    double **temp = matrix_make(d-1);
    for ( int i = 0; i < d; i++ ) {
        for ( int j = 0; j < d; j++ ) {
            matrix_minor(temp, M1, i, j, d);
            M2[i][j] = matrix_det(temp,d-1) * pow(-1.0,i+j);
        }
    }

} //end matrix_cofactor()

void matrix_inverse(double **M2, double **M1, const int d, const double det){

    double **c = matrix_make(d);

    matrix_cofactor(c,M1,d); //Get cofactor matrix

    for ( int i = 0; i < d; i++ ) {
        for ( int j = 0; j < d; j++ ) {
            M2[i][j] = c[j][i] / det;
        }
    }
    matrix_free(c,d);

} // end matrix_inverse()

double find_cubic_fit_mhat_use (double x1, double y1, double x2, double y2, double
x3, double y3, double x4, double y4, double rhat){
double a, b, c, d;

        // This routine receives 4 values to have a cubic fitted,
        // and will return the value of the fitted cubic at a point

    int size = 4; //Matrix size
    double answer = 2*x1;
    double **A = matrix_make(size); //Original Matrix

    A[0][0] = pow(x1, 3);
    A[0][1] = pow(x1, 2);
    A[0][2] = x1;
    A[0][3] = A[1][3] = A[2][3] = A[3][3] = 1;

    A[1][0] = pow(x2, 3);
    A[1][1] = pow(x2, 2);
    A[1][2] = x2;

    A[2][0] = pow(x3, 3);
    A[2][1] = pow(x3, 2);
    A[2][2] = x3;

```

```

A[3][0] = pow(x4, 3);
A[3][1] = pow(x4, 2);
A[3][2] = x4;

double det = matrix_det(A,size);

if ( det == 0.0 ) {
    return answer = -4;
// If det is zero, do nothing, dont write to file
    exit (2);
}
else {
    double **B = matrix_make(size); //Inverse

    matrix_inverse(B,A,size,det); //Get inverse matrix

    a = B[0][0]*y1 + B[0][1]*y2 + B[0][2]*y3 + B[0][3]*y4;
    b = B[1][0]*y1 + B[1][1]*y2 + B[1][2]*y3 + B[1][3]*y4;
    c = B[2][0]*y1 + B[2][1]*y2 + B[2][2]*y3 + B[2][3]*y4;
    d = B[3][0]*y1 + B[3][1]*y2 + B[3][2]*y3 + B[3][3]*y4;
        // ax^3 + bx^2 + cx + d = y(x)

    matrix_free(B,size);

} //End of if statement

return a*pow(rhat, 3) + b*pow(rhat, 2) + c*rhat + d;
} // end cubic_roots2

// This code was written for a "Physics of Neutron Stars" theory project, in
// Jan-April 2007, by J.Pearson & S.Pike.

// With thanks to C.Welshman for allowing use of his nxn matrix inversion code!

```