

Project 1 code

erikalev

February 2018

1 Code

```
1 module rec_mod
2   use healpix_types
3   use params
4   use time_mod
5   use ode_solver
6   use spline_1D_mod
7   implicit none
8
9   real(dp), allocatable, dimension(:) :: dtau           ! First
10     derivative of tau: tau'
11   real(dp), allocatable, dimension(:) :: dg             ! First
12     derivative of g: g'
13   real(dp)                               :: ypl, ypn, eps, hmin
14
15   integer(i4b),                          private :: n, n_inter
16     ! Number of grid points
17   real(dp), allocatable, dimension(:), private :: x_rec, x_inter
18     ! Grid
19   real(dp), allocatable, dimension(:), private :: tau, tau2, tau22,
20     log_tau, d2_tau ! Splined tau and second derivatives
21   real(dp), allocatable, dimension(:), private :: n_e, n_e2 !
22     Splined (log of) electron density, n_e
23   real(dp), allocatable, dimension(:), private :: g, g2, g22, dg_dx
24     ! Splined visibility function
25
26 contains
27
28   subroutine initialize_rec_mod
29     implicit none
30
31     real(dp), allocatable, dimension(:) :: X_e, a_rec ! Fractional
32       electron density, n_e / n_H
33     integer(i4b) :: i, j, k
34     real(dp)      :: saha_limit, y, T_b, n_b, dx, hmin, eps, f, n_e0
35       , X_e0, xstart, xstop, step
36     real(dp)      :: C_r, constant
37
38     logical(lgt) :: use_saha, test_interpolate
```

```

32  saha_limit = 0.99d0      ! Switch from Saha to Peebles when
    X_e < 0.99
33  xstart    = log(1.d-10)  ! Start grids at a = 10^-10
34  xstop     = 0.d0        ! Stop grids at a = 1
35  n         = 1000        ! Number of grid points between
    xstart and xstopo
36  n_inter   = 50000
37  ! Spline variables
38  yp1 = 1.d30
39  ypn = 1.d30
40
41  ! Integration variables
42  eps = 1.d-10
43  hmin = 0.d0
44
45  ! Set as .true. to try the interpolated x-values
46  test_interpolate = .false.
47
48
49  ! Allocating arrays
50  allocate(x_rec(n))
51  allocate(X_e(n))
52  allocate(tau(n))
53  allocate(tau2(n))
54  allocate(tau22(n))
55  allocate(d2_tau(n))
56  allocate(log_tau(n))
57  allocate(n_e(n))
58  allocate(n_e2(n))
59  allocate(g(n))
60  allocate(g2(n))
61  allocate(g22(n))
62  allocate(dg_dx(n))
63  allocate(x_inter(n_inter))
64
65  !-----
66  ! x_(rec) grid
67  !-----
68  dx = (xstop-xstart)/(n-1)
69  x_rec(1) = xstart
70  do i=1, (n-1)
71      x_rec(i+1) = x_rec(i) + dx
72  end do
73
74  !-----
75  ! X_e and n_e at all grid times
76  !-----
77  use_saha = .true.
78  do i = 1, n
79      n_b = Omega_b0*rho_c0/(m_H*exp(3.d0*x_rec(i)))
80
81      if (use_saha) then
82          ! Saha equation
83          T_b = T_0/exp(x_rec(i))
84          constant = (m_e*k_b*T_b/(2.d0*pi*hbar**2))**1.5d0*exp(-
            epsilon_0/(k_b*T_b))/n_b      ! Constant in front of x^2 in
            quadratic formula

```

```

85
86      ! Alternatice quadratic formula to assure numerical
presicion
87      X_e(i) = 2.d0/(sqrt(1.d0 + 4.d0/constant) + 1.d0)
88
89      if (X_e(i) < saha_limit) then
90          use_saha = .false.
91      end if
92  else
93      ! Peeble's equation
94      X_e(i) = X_e(i-1)
95      ! Using the last equated value of Xe
call odeint(X_e(i:i), x_rec(i-1), x_rec(i), eps, dx, hmin
, dXe_dx, bsstep, output) ! Updating Xe through ODEINT
96  end if
97      n_e(i) = X_e(i)*n_b
! Computing n_e
98  end do
99  n_e = log(n_e)
100
101  !-----
102  ! Splined (log of) electron density function
103  !-----
104  call spline(x_rec, n_e, yp1, ypn, n_e2)
105
106  !-----
107  ! Splined optical depth at all grid points
108  !-----
109  tau(n) = 0.d0
110  log_tau(n) = -18.7d0
111  do i = n-1, 1, -1
112      tau(i) = tau(i+1)
113      call odeint(tau(i:i), x_rec(i+1), x_rec(i), eps, dx, hmin,
dtau_dx, bsstep, output)
114      log_tau(i) = log(tau(i))
115  end do
116
117  !-----
118  ! Splined (log of) optical depth
119  !-----
120  call spline(x_rec, log_tau, yp1, ypn, tau2)
121
122  !-----
123  ! Splined second derivative of (log of) optical depth
124  !-----
125  call spline(x_rec, tau2, yp1, ypn, tau22)
126
127  !-----
128  ! Splined vivisility function at all grid points
129  !-----
130  do i=1,n
131      g(i) = -get_dtau(x_rec(i))*exp(-tau(i))
132  end do
133
134  !-----
135
136  !-----

```

```

137 ! Splined visibility function and second derivative
138 !
139 call spline(x_rec, g, yp1, ypn, g2)
140 call spline(x_rec, g2, yp1, ypn, g22)
141
142 !
143 ! Printing x_rec and X_e to file
144 !
145 open(50, file = "X_e.dat")
146 do i=1,n
147     write(50, '(2(E17.8E3))') x_rec(i), X_e(i)
148 end do
149 close(50)
150 if (test_interpolate) then
151     ! x-range set manually
152     xstart = -7.5
153     xstop = 6.0
154
155     x_inter(1) = xstart
156     dx = (xstop-xstart)/(n_inter-1)
157     do i=1, n_inter-1
158         x_inter(i+1) = x_inter(i) + dx
159     end do
160
161     ! write to file
162     open(51, file = "data.dat")
163     do i=1,n
164         write(51, '(7(E17.8E3))') x_inter(i), get_tau(x_inter(i)),
165         , get_dtau(x_inter(i)), get_ddtau(x_inter(i)), &
166         get_g(x_inter(i)), get_dg(x_inter(i)), get_ddg(x_inter(i))
167     end do
168     close(51)
169
170 else
171     ! write to file
172     open(51, file = "data.dat")
173     do i=1,n
174         write(51, '(7(E17.8E3))') x_rec(i), get_tau(x_rec(i)),
175         get_dtau(x_rec(i)), get_ddtau(x_rec(i)), get_g(x_rec(i)), &
176         get_dg(x_rec(i)), get_ddg(x_rec(i))
177     end do
178     close(51)
179
180 end if
181 end subroutine initialize_rec_mod
182
183 ! ----- Saha equation for integration
184
185 subroutine dXe_dx(x, X_e, deriv)
186 use healpix_types
187 implicit none
188 real(dp), intent(in) :: x
189 real(dp), dimension(:), intent(in) :: X_e
190 real(dp), dimension(:), intent(out) :: deriv
191 real(dp) :: lambda_21, lambda_a, Cr, beta, alpha2, n1s, beta2,
192 phi2, H, n_b, T_b

```

```

189 T_b = T_0/exp(x)
190 n_b = Omega_b0*rho_c0/(m_H*exp(3.d0*x))
191 H = get_H(x)
192 nls = n_b*(1.d0 - X_e(1))
193 phi2 = 0.448d0*log(epsilon_0/(k_b*T_b))
194 lambda_a = H*(3.d0*epsilon_0)**3.d0/((8.d0*pi)*(8.d0*pi)*nls*(c
* hbar*c*hbar*c*hbar)) ! [s-1]
195
196 alpha2 = 64.d0*pi / sqrt(27.d0*pi) * (alpha/m_e)**2.d0 * sqrt(
epsilon_0/(k_b*T_b)) * phi2 * hbar*hbar/c
197 beta = alpha2 * (m_e*k_b*T_b / (2.d0*pi*hbar*hbar))**1.5d0 *
exp(-epsilon_0/(k_b*T_b))
198
199 ! combined the betas to avoid infinities
200 beta2 = alpha2 * (m_e*k_b*T_b / (2.d0*pi*hbar*hbar))**1.5d0 *
exp((-epsilon_0)/(4.d0*k_b*T_b))
201
202 Cr = (lambda_21 + lambda_a)/(lambda_21 + lambda_a + beta2)
203 deriv = Cr/H*(beta*(1.d0 - X_e) - n_b*alpha2*X_e*X_e)
204 end subroutine dXe_dx
205
206
207 ! Routine for computing n_e at arbitrary x, using precomputed
information
208 function get_n_e(x)
209 implicit none
210
211 real(dp), intent(in) :: x
212 real(dp) :: get_n_e
213 get_n_e = exp(splint(x_rec, n_e, n_e2, x))
214 end function get_n_e
215
216 ! Routine for computing tau at arbitrary x, using precomputed
information
217 function get_tau(x)
218 implicit none
219 real(dp), intent(in) :: x
220 real(dp) :: get_tau
221 get_tau = exp(splint(x_rec, log_tau, tau2, x))
222 end function get_tau
223
224 ! Routine for the derivative of tau used in ODEINT
225 subroutine dtau_dx(x,tau, deriv)
226 use healpix_types
227 implicit none
228 real(dp), intent(in) :: x
229 real(dp), dimension(:), intent(in) :: tau
230 real(dp), dimension(:), intent(out) :: deriv
231
232 deriv = -get_n_e(x)*sigma_T*exp(x)*c/get_H_p(x)
233 end subroutine dtau_dx
234
235 ! Routine for computing d_tau at arbitrary x, using precomputed
information
236 function get_dtau(x)
237 implicit none
238

```

```

239     real(dp), intent(in) :: x
240     real(dp)              :: get_dtau
241     get_dtau = splint_deriv(x_rec, log_tau, tau2, x)
242     get_dtau = get_dtau*get_tau(x)
243 end function get_dtau
244
245 ! Routine for computing dd_tau at arbitrary x, using precomputed
    information
246 function get_ddtau(x)
247     implicit none
248
249     real(dp), intent(in) :: x
250     real(dp)              :: get_ddtau
251     get_ddtau = splint(x_rec, tau2, tau22, x)
252     get_ddtau = get_ddtau*get_tau(x) + get_dtau(x)**2/get_tau(x)
253 end function get_ddtau
254
255 ! Routine for computing g at arbitrary x, using precomputed
    information
256 function get_g(x)
257     implicit none
258
259     real(dp), intent(in) :: x
260     real(dp)              :: get_g
261     get_g = splint(x_rec, g, g2, x)
262 end function get_g
263
264 ! Routine for computing the derivative of the visibility function
    , g, at arbitray x
265 function get_dg(x)
266     implicit none
267
268     real(dp), intent(in) :: x
269     real(dp)              :: get_dg
270     get_dg = splint_deriv(x_rec, g, g2, x)
271 end function get_dg
272
273 ! Task: Complete routine for computing the second derivative of
    the visibility function, g, at arbitray x
274 function get_ddg(x)
275     implicit none
276
277     real(dp), intent(in) :: x
278     real(dp)              :: get_ddg
279     get_ddg = splint(x_rec, g2, g22, x)
280 end function get_ddg
281
282 end module rec_mod

```