

Case Study # 5: Two-Species Diffusion-Diurnal Kinetics

John Karasinski

Graduate Student Researcher

Center for Human/Robotics/Vehicle Integration and Performance

Department of Mechanical and Aerospace Engineering

University of California

Davis, California 95616

Email: karasinski@ucdavis.edu

1 Problem Description

Chang et al. [1,2] have proposed approximate models to describe the chemical kinetics and transport phenomena associated with the dissociation of oxygen (O_2) into ozone (O_3) and monatomic oxygen (O) in the upper atmosphere. A one-dimensional version of such a model is considered here. The ambient oxygen concentration, c_3 , is constant, while the concentrations of the two minor species, O and O_3 , are $c_1(z, t)$ and $c_2(z, t)$, where z is the elevation above the earth's surface in km (here $30 \leq z \leq 50$) and t is time in seconds. Their transport is modeled using a reaction-diffusion equation,

$$\frac{\partial c_i}{\partial t} = \frac{\partial}{\partial z} \left[K(z) \frac{\partial c_i}{\partial z} \right] + R_i(\vec{c}, t) \quad i = 1, 2, 3. \quad (1)$$

The diffusive term is meant to represent the turbulent vertical transport with

$$K(z) = 10^{-8} \cdot \exp(z/5) \quad [\text{km}^2/\text{s}], \quad (2)$$

and the chemistry is described using the Chapman mechanism [2]. The reaction rates, $R_i(c, t)$, are given by

$$\begin{aligned} R_1(c_1, c_2, t) &= -k_1 c_1 c_3 - k_2 c_1 c_2 + 2k_3(t) c_3 + k_4(t) c_2 \\ R_2(c_1, c_2, t) &= k_1 c_1 c_3 - k_2 c_1 c_2 - k_4(t) c_2 \end{aligned} \quad (3)$$

with,

$$\begin{aligned} k_1 &= 1.63 \times 10^{16} \\ k_2 &= 4.66 \times 10^{16} \\ k_l &= \exp[a_l / \sin \omega t] \text{ if } \sin \omega t > 0, \text{ else } 0 \quad (l = 3, 4) \end{aligned}$$

and with $a_3 = 22.62$, $a_4 = 7.601$, and $\omega = \pi/43200$. This system is subject to the initial conditions,

$$\begin{aligned} c_1(z, 0) &= 10^6 \cdot \gamma(z) \\ c_2(z, 0) &= 10^{12} \cdot \gamma(z), \end{aligned} \quad (4)$$

where

$$\gamma(z) = 1 - \left(\frac{z-40}{10} \right)^2 + \frac{1}{2} \left(\frac{z-40}{10} \right)^4, \quad (5)$$

and a boundary condition of no flux at the top and bottom of the vertical atmospheric layer considered.

2 Numerical Solution Approach

To generate a system of ordinary differential equations, all the spatial derivatives in Equation 1 are replaced with centered finite differences. For the base case considered, the domain is discretized into $M = 50$ partitions, $\Delta z = 20/M$, and $z_j = 30 + j(\Delta z)$ for $0 \leq j \leq M$.

The function $c^i(z_j, t)$ can then be approximated as

$$\dot{c}_j^i = (\Delta z)^{-2} [K_{j+1/2} c_{j+1}^i - (K_{j+1/2} + K_{j-1/2}) c_j^i + K_{j-1/2} c_{j-1}^i] + R^i(\mathbf{c}, t), \quad (6)$$

where $K_{j \pm 1/2} = K(30 + [j \pm 1/2] \Delta z)$. A system of $2M$ ODEs is then specified by setting $\mathbf{y}(t) = [c_1^1(t), c_1^2(t), c_2^1(t), c_2^2(t), \dots, c_M^1(t), c_M^2(t)]^T$, with boundary conditions $c_0^i = c_2^i$ and $c_{M-1}^i = c_{M+1}^i$. The two parabolic PDEs are then reduced to a system of $2M$ ODEs of the form $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$.

Two solvers are called from scipy version 0.11.0 to solve this system of ODEs. A stiff solver, "vode", is an implicit method based on the backward differentiation formulas, and a non-stiff solver, "dopri5", is an explicit Runge-Kutta method of order (4,5) are used.

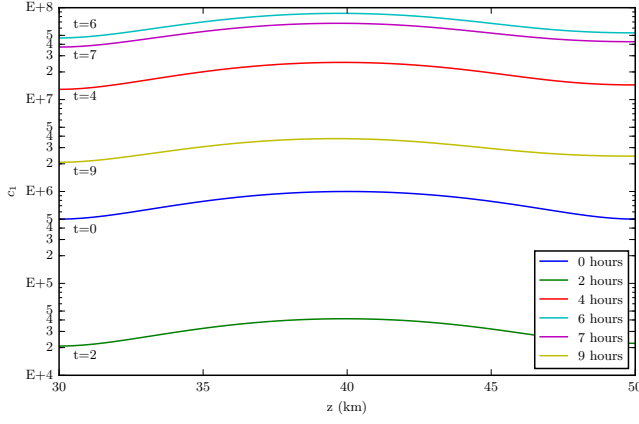


Fig. 1: c_1 vs. z at $t = 0, 2, 4, 6, 7$, and 9 hours

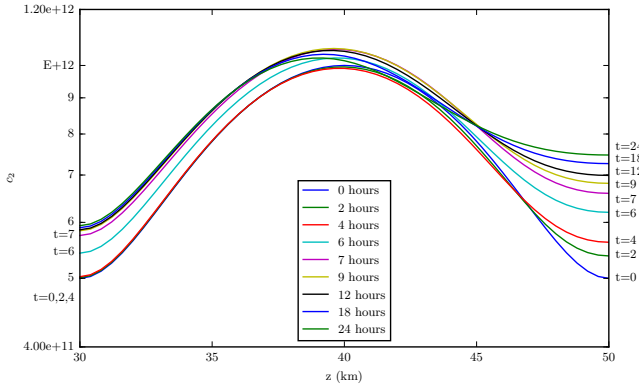


Fig. 2: c_2 vs. z at $t = 0, 2, 4, 6, 7, 9, 12, 18$, and 24 hours

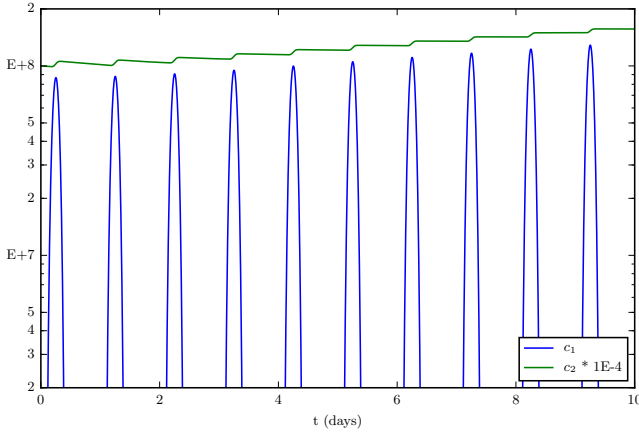


Fig. 3: c_1 and c_2 vs. time (from 0 to 10 days) at $z = 40$ km, c_2 is scaled by $1E-4$

3 Results Discussion

3.1 Comparison to Published Results

Both methods required an absolute tolerance of $1E-1$ and a relative tolerance $1E-3$ for convergence. The results found for both the bdf and dopri5 solvers are effectively the same when plotted on a logarithmic scale. The results from

Solver	24 hours	10 days
dopri5	1025	10069
bdf	1318	13237

Table 1: Wall clock time, in seconds, for each solver to solve to $t = 24$ hours and $t = 10$ days

the dopri5 solver can be seen in Figures 1, 2, and 3, and compare well with published results [1, 2].

In particular the ordering of the c_1 concentrations are the same from $t = 0$ to $t = 9$ hours. From highest to lowest concentrations, the ordering is $t = 6, 7, 4, 9, 0$, and 2 hours. The ordering of the c_2 concentrations also agree over the time range of $t = 0$ to $t = 24$ hours. From highest to lowest concentrations at the upper boundary of 50 km, the ordering is $t = 24, 18, 12, 9, 7, 6, 4, 2$, and 0 hours. Additionally, the concentrations of both species were investigated at a height of 40 km as a function of time. A peak is seen in the concentration of c_1 during each day. This peak grows slightly as time elapses. The concentration of c_2 acts as a step function, stepping up each day at approximately the same time as the c_1 concentration peaks.

The results for concentrations of c_1 and c_2 as a function of altitude, and the results for the concentrations of c_1 and c_2 at 40 km as a function of time agree with the results in both reference papers.

3.2 Difference in convergence time

For stiff problems such as this, stiff solvers should take longer to convergence than non-stiff solvers. Table 1 shows the wall clock time, in seconds, to converge for the 24 hour and 10 day cases. The bdf solver takes about 31% longer to converge than the dopri5 solver, suggesting that this problem is indeed stiff.

3.3 Sensitivity to Mesh Density

The effect of varying the mesh size was also investigated. The solution was found for the concentrations of both species for both solvers at $t = 4$ hours using meshes resultant from $M = 5, 10, 25, 50, 75, 100$, and 200 . The solution with the mesh resulting from $M = 200$ was taken to be effectively equivalent to the analytical solution, and was used as the basis for comparison between the solutions found with the other meshes.

The Root Mean Square error,

$$RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^N [c_i - c_i^*]^2}, \quad (7)$$

and the Normalized Root Mean Square error,

$$NRMS = \frac{RMSE}{\max(c_i^*) - \min(c_i^*)}, \quad (8)$$

M	c_1	c_2
5	1.628E-2	1.648E-2
10	9.766E-3	9.829E-3
25	4.178E-3	4.110E-3
50	1.879E-3	1.811E-3
75	1.031E-3	9.806E-4
100	6.923E-4	6.098E-4

Table 2: NRMS of results at $t = 4$ hours compared to results at $M = 200$ for the dopri5 solver

M	c_1	c_2
5	1.654E-2	1.648E-2
10	9.796E-3	9.829E-3
25	4.091E-3	4.110E-3
50	1.802E-3	1.811E-3
75	9.436E-4	9.806E-4
100	5.982E-4	6.098E-4

Table 3: NRMS of results at $t = 4$ hours compared to results at $M = 200$ for the bdf solver

can be calculated. Here c_i is the result for the the concentration of each species i at each point on the 1-D domain generated from each value for M , c_i^* is the solution from the $M = 200$ simulation, and N is the number of points on the 1-D domain. The NRMS for each case is expressed as a percentage, where lower values indicate a result closer to the analytic solution. For the complete NRMS results for each solver and mesh, see Tables 2 and 3.

These results show that the difference between the meshes decreases rapidly for larger M . At $M = 50$, the NRMS is around $1.8E-3$ for both concentrations using both solvers. This relatively low value indicates that using a mesh from $M = 50$ leads to relatively good results with the benefit of a reduced solver time. A plot of these results can be seen as Figure 4.

4 Conclusion

A previously developed model to describe the chemical kinetics and transport phenomena associated with the dissociation of oxygen in the upper atmosphere was investigated and the results were compared to previously published results. A numerical solution of the system was discretized using the method of lines and central differencing for the diffusion term on a uniform mesh with M nodes.

A Python script was created and scipy was used to call both a stiff and non-stiff ODE solver to investigate the concentrations of two species of oxygen as a function of altitude

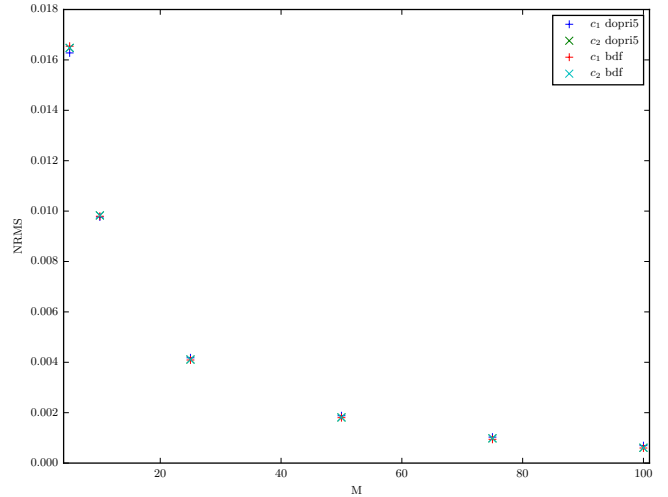


Fig. 4: NRMS for $M = 5, 10, 25, 50, 75$, and 100 compared against $M = 200$ for both solvers and concentrations

and time. This script can be seen as Appendix 1. The results of these two solvers both compared well to previously published results, though the stiff solver took approximately 31% longer to converge. A mesh sensitivity study was also performed, and confirmed that the base mesh resulting from $M = 50$ was adequate for the values investigated here.

References

- [1] Chang, J., Hindmarsh, A., and Madsen, N., 1974. "Simulation of chemical kinetics transport in the stratosphere". In *Stiff differential systems*. Springer, pp. 51–65.
- [2] Byrne, G. D., and Hindmarsh, A. C., 1987. "Stiff ode solvers: A review of current and coming attractions". *Journal of Computational Physics*, **70**(1), pp. 1–62.

Appendix A: Python Code

```
1 import numpy as np
2 from scipy.integrate import ode
3 from time import clock
4 from PrettyPlots import *
5
6
7 def K(j):
8     return 1E-8 * np.exp((30. + j * dz) / 5.)
9
10
11 def gamma(z):
12     return 1. - ((z - 40.) / 10.) ** 2 + (1. / 2.) * ((z - 40.) / 10.) ** 4
13
14
15 def R(y_1, y_2, t):
16     if np.sin(w * t) > 0.:
17         k_3 = np.exp(-a_3 / np.sin(w * t))
18         k_4 = np.exp(-a_4 / np.sin(w * t))
19     else:
20         k_3 = 0.
21         k_4 = 0.
22
23     R_1 = -k_1 * y_1 * y_3 - k_2 * y_1 * y_2 + 2. * k_3 * y_3 + k_4 * y_2
24     R_2 = +k_1 * y_1 * y_3 - k_2 * y_1 * y_2 - k_4 * y_2
25     return R_1, R_2
26
27
28 def system(t, y):
29     f = np.zeros(len(y))
30
31     R1, R2 = R(y[0], y[1], t)
32     l_p, l_m = 3. / 2., 1. / 2.
33
34     f[0] = (dz ** -2 * (K(l_p) * y[2] - (K(l_p) + K(l_m)) * y[0] + K(l_m) * y[2])) + R1)
35     f[1] = (dz ** -2 * (K(l_p) * y[3] - (K(l_p) + K(l_m)) * y[1] + K(l_m) * y[3])) + R2)
36
37     for i in range(1, M):
38         R1, R2 = R(y[2 * i], y[2 * i + 1], t)
39         l_p, l_m = i + 3. / 2., i + 1. / 2.
40
41         f[2 * i] = (dz ** -2 * (K(l_p) * y[2 * i + 2] -
42                                (K(l_p) + K(l_m)) * y[2 * i] + K(l_m) * y[2 * i - 2])) + R1)
43         f[2 * i + 1] = (dz ** -2 * (K(l_p) * y[2 * i + 3] -
44                                   (K(l_p) + K(l_m)) * y[2 * i + 1] + K(l_m) * y[2 * i - 1])) + R2)
45
46     R1, R2 = R(y[2 * M], y[2 * M + 1], t)
47     l_p, l_m = M + 1. / 2., M - 1. / 2.
48
49     f[2 * M] = (dz ** -2 * (K(l_p) * y[2 * M - 2] -
50                           (K(l_p) + K(l_m)) * y[2 * M] + K(l_m) * y[2 * M - 2])) + R1)
51     f[2 * M + 1] = (dz ** -2 * (K(l_p) * y[2 * M - 1] -
52                               (K(l_p) + K(l_m)) * y[2 * M + 1] + K(l_m) * y[2 * M - 1])) + R2)
53
54     return f
55
56
57 def solve(solver, c, time, integrator):
58     # Create result arrays
59     c1, c2, c1_40km, c2_40km, t = [], [], [], [], []
60
61     start_time = clock()
62     for i in range(0, len(time) - 1):
63         # Initial and final time
64         t_0 = time[i]
65         t_f = time[i + 1]
66
67         # Solver setup
```

```

68     sol = []
69     solver.set_initial_value(c, t_0)
70     while solver.successful() and solver.t < t_f:
71         solver.integrate(solver.t + dt)
72         sol.append(solver.y)
73
74         # Keep time history for 40km point
75         one, two = sol[-1][0::2], sol[-1][1::2]
76         mid_one, mid_two = one[M / 2], two[M / 2]
77         c1_40km.append(mid_one), c2_40km.append(mid_two)
78         t.append(solver.t)
79
80         print "{0:03.2f}%".format(100. * solver.t / time[-1])
81
82         # Save c1, c2 solutions
83         c1.append(one), c2.append(two)
84
85         # Update initial conditions for next iteration
86         c = sol[-1]
87
88     elapsed_time = clock() - start_time
89     print(elapsed_time, "seconds process time")
90
91     output = [c1, c2, c1_40km, c2_40km, t]
92     return output
93
94
95 def save_variables(name, z, c1, c2, t, c1_40km, c2_40km):
96     try:
97         os.mkdir('data')
98     except Exception:
99         pass
100
101     try:
102         os.mkdir('data/' + name)
103     except Exception:
104         pass
105
106     np.savetxt('data/' + name + '/z.csv', z)
107     np.savetxt('data/' + name + '/c1.csv', c1)
108     np.savetxt('data/' + name + '/c2.csv', c2)
109     np.savetxt('data/' + name + '/t.csv', t)
110     np.savetxt('data/' + name + '/c1_40km.csv', c1_40km)
111     np.savetxt('data/' + name + '/c2_40km.csv', c2_40km)
112
113
114 def load_variables(name):
115     z = np.loadtxt('data/' + name + '/z.csv')
116     c1 = np.loadtxt('data/' + name + '/c1.csv')
117     c2 = np.loadtxt('data/' + name + '/c2.csv')
118     t = np.loadtxt('data/' + name + '/t.csv')
119     c1_40km = np.loadtxt('data/' + name + '/c1_40km.csv')
120     c2_40km = np.loadtxt('data/' + name + '/c2_40km.csv')
121
122     return z, c1, c2, t, c1_40km, c2_40km
123
124
125 def run_trials(z, integrators, times, M):
126     # Set up ODE solver
127     for integrator in integrators:
128         if integrator == 'dop853' or integrator == 'dopri5':
129             solver = ode(system)
130             solver.set_integrator(integrator, atol=1E-1, rtol=1E-3)
131         elif integrator == 'bdf':
132             solver = ode(system)
133             solver.set_integrator('vode', method=integrator, atol=1E-1, rtol=1E-3, nsteps=2000)
134
135         name = integrator + ' ' + str(times[-1]) + ' ' + str(M)
136         try:

```

```

137     z, c1, c2, t, c1_40km, c2_40km = load_variables(name)
138     print "Loaded data for: " + name
139 except:
140     print "Starting solver: ", integrator, "with times", times
141     c1, c2, c1_40km, c2_40km, t = solve(solver, c, times, integrator)
142     save_variables(name, z, c1, c2, t, c1_40km, c2_40km)
143
144     # And plot some things
145     if times[-1] == 86400.0:
146         labels = [str(int(time / 3600.)) + " hours" for time in times[1:]]
147         plot_c1(z, c, c1, labels, name)
148         plot_c2(z, c, c2, labels, name)
149     elif times[-1] == 864000.0:
150         plot_40km(t, c1_40km, c2_40km, name)
151
152
153 def sensitivity_analysis(integrators, times, meshes):
154     plt.figure()
155     for integrator in integrators:
156         z_M, c1_M, c2_M = [], [], []
157
158         for M in meshes:
159             name = integrator + ' ' + str(times[-1]) + ' ' + str(M)
160             try:
161                 z, c1, c2, _, _, _ = load_variables(name)
162             except Exception:
163                 print Exception
164             z_M.append(list(z))
165             c1_M.append(list(c1[-1]))
166             c2_M.append(list(c2[-1]))
167
168             best_z = z_M[-1]
169             best_c1, best_c2 = c1_M[-1], c2_M[-1]
170             NRMS1, NRMS2 = [], []
171             for j, mesh in enumerate(z_M):
172                 if j + 1 == len(z_M): break # RMS with yourself is silly
173                 best1, best2, curr1, curr2 = [], [], [], []
174                 for i, element in enumerate(best_z):
175                     if element in mesh:
176                         best1.append(best_c1[i])
177                         best2.append(best_c2[i])
178                         curr1.append(c1_M[j][mesh.index(element)])
179                         curr2.append(c2_M[j][mesh.index(element)])
180
181                 best1, best2 = np.array(best1), np.array(best2)
182                 curr1, curr2 = np.array(curr1), np.array(curr2)
183
184                 err1, err2 = curr1 - best1, curr2 - best2
185                 NRMS1.append(np.sqrt(np.mean(np.square(err1)))/(max(best1) - min(best1)))
186                 NRMS2.append(np.sqrt(np.mean(np.square(err2)))/(max(best2) - min(best2)))
187
188             x = [mesh for mesh in meshes][0:-1]
189             plt.plot(x, NRMS1, '+', label='$c_1$ ' + integrator)
190             plt.plot(x, NRMS2, 'x', label='$c_2$ ' + integrator)
191
192             plt.ylabel('NRMS')
193             plt.xlabel('M')
194             plt.xlim([meshes[0] - 1, meshes[-2] + 1])
195             plt.legend()
196             save_name = str(meshes) + '.pdf'
197             save_plot(save_name)
198
199
200 # Basic problem parameters
201 y_3 = 3.7E16 # Concentration of O_2 (constant)
202 k_1 = 1.63E-16 # Reaction rate [O + O_2 -> O_3]
203 k_2 = 4.66E-16 # Reaction rate [O + O_3 -> 2 * O_2]
204 a_3 = 22.62 # Constant used in calculation of k_3
205 a_4 = 7.601 # Constant used in calculation of k_4

```

```

206 w = np.pi / 43200.    # Cycle (half a day) [1/sec]
207 dt = 60.              # seconds
208
209 # Base Case
210 M = 50                 # Number of sections
211 dz = 20. / M           # 20km divided by M subsections
212
213 # This generates the initial conditions
214 c = np.zeros(2 * (M + 1))
215 z = np.zeros(M + 1)
216 for i in range(0, M + 1):
217     z[i] = 30. + i * dz
218     c[2 * i] = 1E6 * gamma(z[i])
219     c[2 * i + 1] = 1E12 * gamma(z[i])
220
221 # Run the trials
222 integrators = ['dopri5', 'bdf']
223 times = 3600. * np.array([0., 2., 4., 6., 7., 9., 12., 18., 24.])
224 run_trials(z, integrators, times, M)
225
226 integrators = ['dopri5', 'bdf']
227 times = 3600. * np.array([0., 2., 4., 6., 7., 9., 12., 18., 240.])
228 run_trials(z, integrators, times, M)
229
230 # Mesh Analysis
231 meshes = [5, 10, 25, 50, 75, 100, 200]
232 for M in meshes:
233     dz = 20. / M        # 20km divided by M subsections
234
235     # This generates the initial conditions
236     c = np.zeros(2 * (M + 1))
237     z = np.zeros(M + 1)
238     for i in range(0, M + 1):
239         z[i] = 30. + i * dz
240         c[2 * i] = 1E6 * gamma(z[i])
241         c[2 * i + 1] = 1E12 * gamma(z[i])
242
243     # Time array
244     dt = 60.
245
246     integrators = ['dopri5', 'bdf']
247     times = 3600. * np.array([0., 2., 4.])
248     run_trials(z, integrators, times, M)
249
250 sensitivity_analysis(integrators, times, meshes)

```

Listing 1: Code to create solutions

```

1 import numpy as np
2 import matplotlib
3 matplotlib.use('TkAgg')
4 import matplotlib.pyplot as plt
5 import os
6
7 # Configure figures for production
8 WIDTH = 495.0 # the number latex spits out
9 FACTOR = 1.0 # the fraction of the width the figure should occupy
10 fig_width_pt = WIDTH * FACTOR
11
12 inches_per_pt = 1.0 / 72.27
13 golden_ratio = (np.sqrt(5) - 1.0) / 2.0 # because it looks good
14 fig_width_in = fig_width_pt * inches_per_pt # figure width in inches
15 fig_height_in = fig_width_in * golden_ratio # figure height in inches
16 fig_dims = [fig_width_in, fig_height_in] # fig dims as a list
17
18
19 def save_plot(save_name):
20     # Save plots
21     try:

```

```

22     os.mkdir('figures')
23 except Exception:
24     pass
25
26 plt.savefig('figures/' + save_name, bbox_inches='tight')
27 plt.close()
28
29
30 def plot_c1(z, initial, c1, labels, integrator):
31     plt.figure(figsize=fig_dims)
32     plt.plot(z, initial[0::2], label='0 hours')
33     for solution, label in zip(c1, labels):
34         if "12" in label:
35             break
36     plt.plot(z, solution, label=label)
37     plt.ylabel('$c_1$')
38     plt.xlabel('z (km)')
39     plt.yscale('log')
40     plt.ylim([1E4, 1E8])
41     plt.yticks([1E4, 2E4, 3E4, 4E4, 5E4,
42                1E5, 2E5, 3E5, 4E5, 5E5,
43                1E6, 2E6, 3E6, 4E6, 5E6,
44                1E7, 2E7, 3E7, 4E7, 5E7, 1E8],
45               ['E+4', '2', '3', '4', '5',
46                'E+5', '2', '3', '4', '5',
47                'E+6', '2', '3', '4', '5',
48                'E+7', '2', '3', '4', '5', 'E+8'])
49     plt.legend(loc='lower right')
50
51     plt.text(30.5, 1.5e+4, 't=2', fontsize=9, family='serif')
52     plt.text(30.5, 3.5e+5, 't=0', fontsize=9, family='serif')
53     plt.text(30.5, 1.5e+6, 't=9', fontsize=9, family='serif')
54     plt.text(30.5, 1.e+7, 't=4', fontsize=9, family='serif')
55     plt.text(30.5, 2.8e+7, 't=7', fontsize=9, family='serif')
56     plt.text(30.5, 6.e+7, 't=6', fontsize=9, family='serif')
57
58     save_name = integrator + ' c1.pdf'
59     save_plot(save_name)
60
61
62 def plot_c2(z, initial, c2, labels, integrator):
63     plt.figure(figsize=fig_dims)
64     plt.plot(z, initial[1::2], label='0 hours')
65     for solution, label in zip(c2, labels):
66         if "240" in label:
67             break
68     plt.plot(z, solution, label=label)
69     plt.ylabel('$c_2$')
70     plt.xlabel('z (km)')
71     plt.yscale('log')
72     plt.ylim([4.E11, 1.2E12])
73     plt.yticks([4E11, 5E11, 6E11, 7E11, 8E11, 9E11, 1E12, 1.2E12],
74               ['4.00e+11', '5', '6', '7', '8', '9', 'E+12', '1.20e+12'])
75     plt.legend(loc='best')
76
77     # Left side text
78     plt.text(28.25, 4.65e+11, 't=0,2,4', fontsize=9, family='serif')
79     plt.text(29, 5.4e+11, 't=6', fontsize=9, family='serif')
80     plt.text(29, 5.7e+11, 't=7', fontsize=9, family='serif')
81
82     # Right side text
83     plt.text(50.25, 4.95e+11, 't=0', fontsize=9, family='serif')
84     plt.text(50.25, 5.35e+11, 't=2', fontsize=9, family='serif')
85     plt.text(50.25, 5.6e+11, 't=4', fontsize=9, family='serif')
86     plt.text(50.25, 6.1e+11, 't=6', fontsize=9, family='serif')
87     plt.text(50.25, 6.4e+11, 't=7', fontsize=9, family='serif')
88     plt.text(50.25, 6.7e+11, 't=9', fontsize=9, family='serif')
89     plt.text(50.25, 7.0e+11, 't=12', fontsize=9, family='serif')
90     plt.text(50.25, 7.3e+11, 't=18', fontsize=9, family='serif')

```



```

91 plt.text(50.25, 7.6e+11, 't=24', fontsize=9, family='serif')
92
93 save_name = integrator + ' c2.pdf'
94 save_plot(save_name)
95
96
97 def plot_40km(t, c1_40km, c2_40km, integrator):
98     c2_40km_scaled = [1E-4 * val for val in c2_40km]
99     days = [val / 86400. for val in t]
100
101     plt.figure(figsize=fig_dims)
102     plt.plot(days, c1_40km, label='$c_1$')
103     plt.plot(days, c2_40km_scaled, label='$c_2$ * 1E-4')
104     plt.xlabel('t (days)')
105     plt.yscale('log')
106     plt.ylim([2.E6, 2E8])
107     plt.yticks([2E6, 3E6, 4E6, 5E6, 1E7, 2E7, 3E7, 4E7, 5E7, 1E8, 2E8],
108               ['2', '3', '4', '5', 'E+7', '2', '3', '4', '5', 'E+8', '2'])
109     plt.xlim([0, days[-1]])
110     plt.legend(loc='lower right')
111     save_name = integrator + ' time.pdf'
112     save_plot(save_name)

```

Listing 2: Code to generate pretty plots