

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
- 2 Numerical Solution Approach
- 3 Results Discussion

Solver	Time (s)
dop853	11092
dopri5	10069
bdf	13237

Table 1: 10 days solution

Solver	Time (s)
dop853	1120
dopri5	1025
bdf	1318

Table 2: 24 hours solution

4 Conclusion

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.

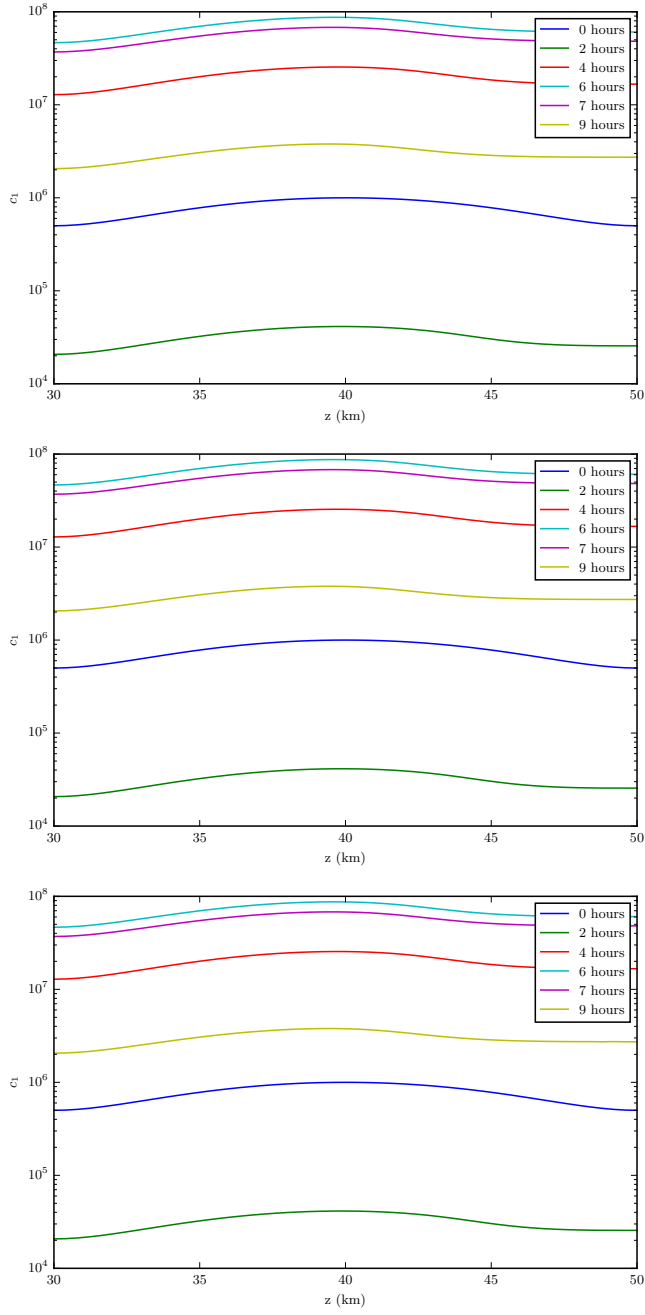


Fig. 1: Results from bdf, dop853, and dopri5 solvers of c_1 vs. z at $t = 0, 2, 4, 6, 7,$ and 9 hours

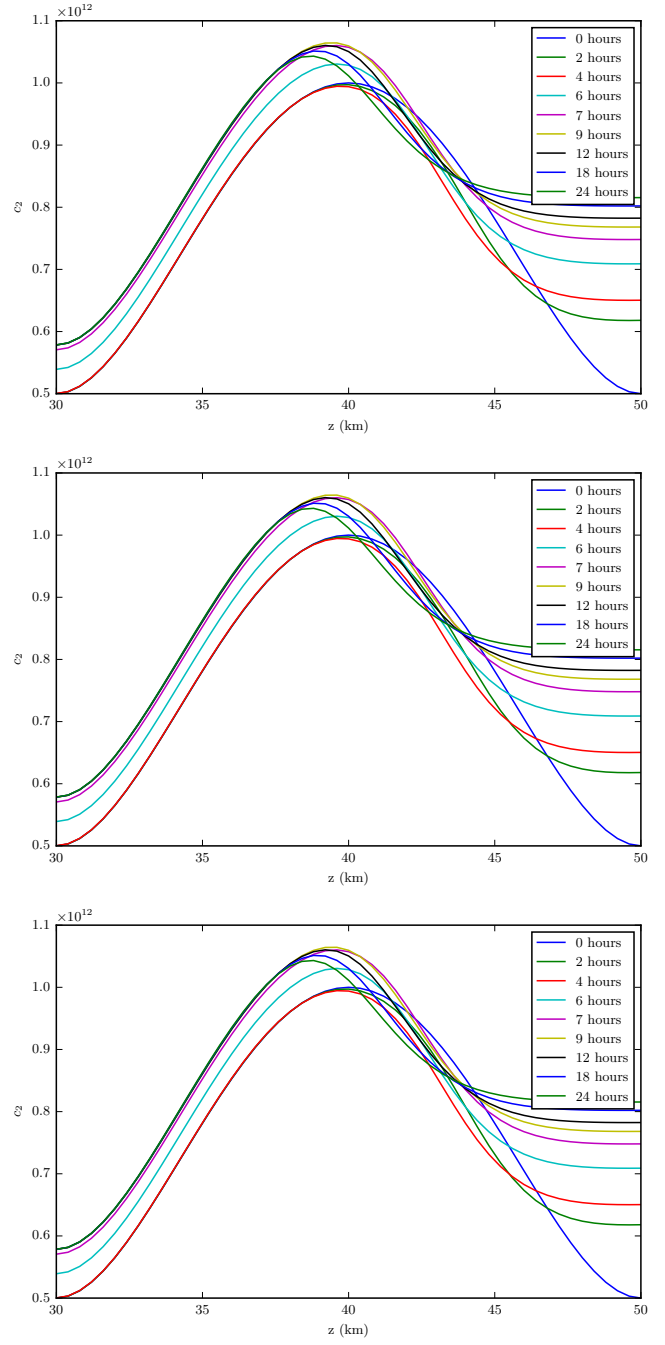


Fig. 2: Results from bdf, dop853, and dopri5 solvers of c_2 vs. z at $t = 0, 2, 4, 6, 7, 9, 12, 18,$ and 24 hours

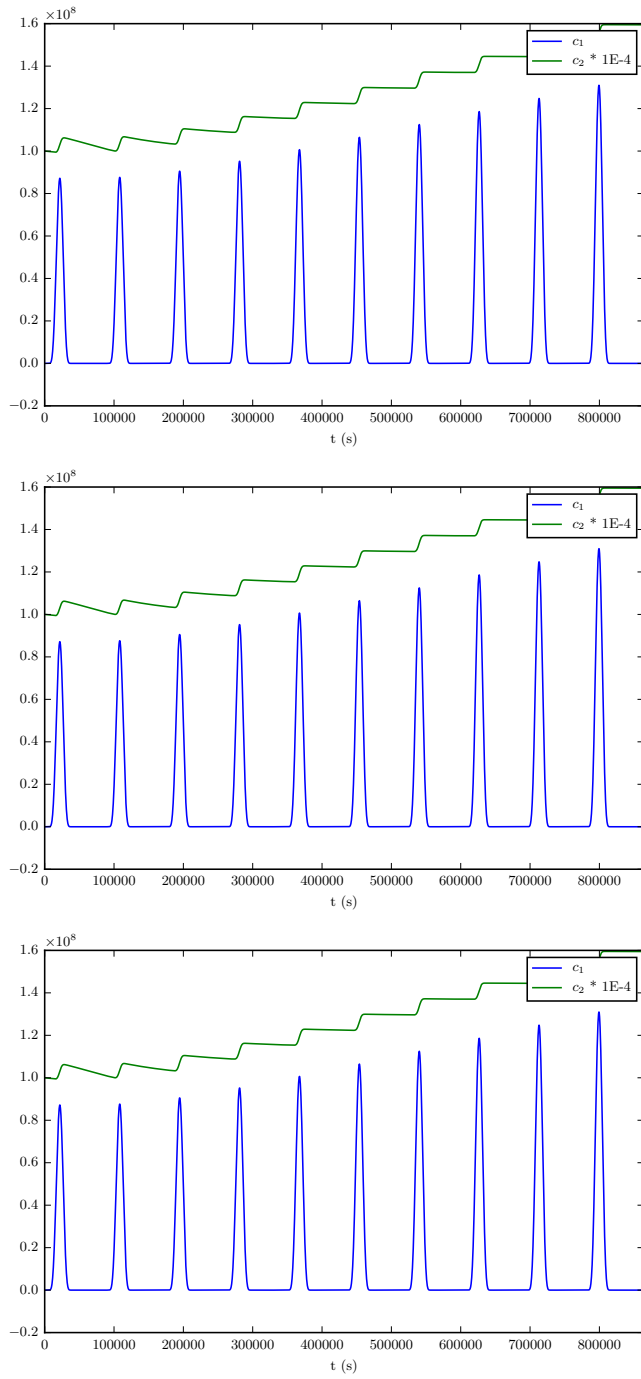


Fig. 3: Results from bdf, dop853, and dopri5 solvers of c_1 and c_2 vs. time (from 0 to 10 days) at $z = 40$ km

Appendix A: Python Code

```
1 import numpy as np
2 from scipy.integrate import ode
3 from time import clock
4 from scipy.sparse import spdiags
5 import json
6 from PrettyPlots import *
7
8
9 def K(z):
10     return 10E-8 * np.exp(z / 5.)
11
12
13 def gamma(z):
14     return 1. - ((z - 40.) / 10.) ** 2 + (1. / 2.) * ((z - 40.) / 10.) ** 4
15
16
17 def R(y_1, y_2, t):
18     """
19     Find the reaction rates, R_1 and R_2, of the system at state c and time t.
20     """
21
22     if np.sin(w * t) > 0:
23         k_3 = np.exp(-a_3 / np.sin(w * t))
24         k_4 = np.exp(-a_4 / np.sin(w * t))
25     else:
26         k_3 = 0
27         k_4 = 0
28
29     R_1 = -k_1 * y_1 * y_3 - k_2 * y_1 * y_2 + 2. * k_3 * y_3 + k_4 * y_2
30     R_2 = +k_1 * y_1 * y_3 - k_2 * y_1 * y_2 - k_4 * y_2
31     return R_1, R_2
32
33
34 def system(t, y):
35     f = np.zeros(len(y))
36
37     R1, R2 = R(y[0], y[1], t)
38     l_p, l_m = 3. / 2., 1. / 2.
39
40     f[0] = (dz ** -2 * (K(l_p) * y[2] - (K(l_p) + K(l_m)) * y[0] + K(l_m) * y[2])) + R1)
41     f[1] = (dz ** -2 * (K(l_p) * y[3] - (K(l_p) + K(l_m)) * y[1] + K(l_m) * y[3])) + R2)
42
43     for i in range(1, M):
44         R1, R2 = R(y[2 * i], y[2 * i + 1], t)
45         l_p, l_m = i + 3. / 2., i + 1. / 2.
46
47         f[2 * i] = (dz ** -2 * (K(l_p) * y[2 * i + 2] - (K(l_p) + K(l_m)) * y[2 * i] + K(l_m) * y[2 * i - 2])) + R1
48         f[2 * i + 1] = (dz ** -2 * (K(l_p) * y[2 * i + 3] - (K(l_p) + K(l_m)) * y[2 * i + 1] + K(l_m) * y[2 * i - 1])) + R2
49
50     R1, R2 = R(y[2 * M - 2], y[2 * M - 1], t)
51     l_p, l_m = M + 1. / 2., M - 1. / 2.
52
53     f[-2] = (dz ** -2 * (K(l_p) * y[2 * M - 4] - (K(l_p) + K(l_m)) * y[2 * M - 2] + K(l_m) * y[2 * M - 4])) + R1
54     f[-1] = (dz ** -2 * (K(l_p) * y[2 * M - 3] - (K(l_p) + K(l_m)) * y[2 * M - 1] + K(l_m) * y[2 * M - 3])) + R2
55
56     return f
57
58
59 def jacobian(t, y):
60     main = np.zeros(len(y))
61     sub_1, sub_2 = np.zeros(len(y)), np.zeros(len(y))
62     sup_1, sup_2 = np.zeros(len(y)), np.zeros(len(y))
63
64     if np.sin(w * t) > 0:
65         k_4 = np.exp(-a_4 / np.sin(w * t))
66     else:
67         k_4 = 0
```

```

68 sup_2[-2] = dz ** -2 * (K(M + 1./2.) + K(M - 1./2.))
69 sup_2[-1] = sup_2[-2]
70 for i in range(2, 2 * M):
71     sup_2[i] = dz ** -2 * K(i + 1. + 1./2.)
72
73
74 for i in range(1, M + 1):
75     sup_1[2 * i] = -k_2 * y[2 * i] + k_4
76     sup_1[2 * i + 1] = 0
77
78 for i in range(0, M + 1):
79     main[2 * i] = -dz ** -2 * (K(i + 1. + 1./2.) + K(i + 1 - 1./2.)) - k_1 * y_3 - k_2 * y[2 * i + 1]
80     main[2 * i + 1] = -k_2 * y[2 * i] - k_4
81
82 for i in range(0, M):
83     sub_1[2 * i] = k_1 * y_3 - k_2 * y[2 * i + 1]
84     sub_1[2 * i + 1] = 0
85
86 sub_2[0:2] = dz ** -2 * K(1. - 1./2.)
87 for i in range(2, 2 * M):
88     sub_2[i] = dz ** -2 * (K(i + 1. + 1./2.) + K(i + 1 - 1./2.))
89
90 diag_rows = np.array([sup_2, sup_1, main, sub_1, sub_2])
91 positions = [2, 1, 0, -1, -2]
92 jac = spdiags(diag_rows, positions, len(y), len(y)).todense()
93
94 return jac
95
96
97 def solve(solver, c, integrator):
98     # Create result arrays
99     c1, c2, c1_40km, c2_40km, t = [], [], [], [], []
100
101     start_time = clock()
102     for i in range(0, len(times) - 1):
103         # Initial and final time
104         t_0 = times[i]
105         t_f = times[i + 1]
106
107         # Solver setup
108         sol = []
109         solver.set_initial_value(c, t_0)
110         while solver.successful() and solver.t < t_f:
111             solver.integrate(solver.t + dt)
112             sol.append(solver.y)
113
114         # keep time history for 40km point
115         one, two = sol[-1][0::2], sol[-1][1::2]
116         mid_one, mid_two = one[M / 2], two[M / 2]
117         c1_40km.append(mid_one), c2_40km.append(mid_two)
118         t.append(solver.t)
119
120         print "{0:3.2f}%".format(clock(), 100. * t[-1] / times[-1])
121
122         # Save c1, c2 solutions
123         c1.append(one), c2.append(two)
124
125         #Update initial conditions for next iteration
126         c = sol[-1]
127
128     elapsed_time = clock() - start_time
129     print(elapsed_time, "seconds process time")
130
131     output = [c1, c2, c1_40km, c2_40km, t]
132     return output
133
134
135 def run_trials():
136     number_of_solvers = 4

```

```

137 for trial in range(number_of_solvers):
138     # Set up ODE solver
139     if trial == 0:
140         solver = ode(system)
141         integrator = 'dop853'
142         solver.set_integrator(integrator, atol=1E-1, rtol=1E-3)
143     elif trial == 1:
144         solver = ode(system)
145         integrator = 'dopri5'
146         solver.set_integrator(integrator, atol=1E-1, rtol=1E-3)
147     elif trial == 2:
148         solver = ode(system)
149         integrator = 'bdf'
150         solver.set_integrator('vode', method=integrator, atol=1E-1, rtol=1E-3, nsteps=1000)
151     elif trial == 3:
152         solver = ode(system, jacobian)
153         integrator = 'bdf Jacobian'
154         solver.set_integrator('vode', method=integrator.split(' ')[0], atol=1E-1, rtol=1E-3, nsteps=1000, with_ja
155
156     print("Starting solver: ", integrator)
157     c1, c2, c1_40km, c2_40km, t = solve(solver, c, integrator)
158
159     # And plot some things
160     plot_c1(z, c, c1, labels, integrator)
161     plot_c2(z, c, c2, labels, integrator)
162     plot_40km(t, c1_40km, c2_40km, integrator)
163
164 # Basic problem parameters
165 M = 50 # Number of sections
166 dz = 20. / M # 20km divided by M subsections
167 z = [30. + j * dz for j in range(M + 1)]
168
169 y_3 = 3.7E16 # Concentration of O_2 (constant)
170 k_1 = 1.63E-16 # Reaction rate [O + O_2 -> O_3]
171 k_2 = 4.66E-16 # Reaction rate [O + O_3 -> 2 * O_2]
172 a_3 = 22.62 # Constant used in calculation of k_3
173 a_4 = 7.601 # Constant used in calculation of k_4
174 w = np.pi / 43200. # Cycle (half a day) [1/sec]
175
176 # This generates the initial conditions
177 c = np.zeros(len(2 * z))
178 for i, _ in enumerate(z):
179     c[2 * i] = 1E6 * gamma(z[i])
180     c[2 * i + 1] = 1E12 * gamma(z[i])
181
182 # Time array
183 times = 3600. * np.array([0., 2., 4., 6., 7., 9., 12., 18., 24., 240.])
184 dt = 60.
185
186 labels = [str(int(x / 3600.)) + " hours" for x in times[1:]]
187
188 run_trials()

```

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.legend()
41     save_name = integrator + ' c1.pdf'
42     save_plot(save_name)
43
44
45 def plot_c2(z, initial, c2, labels, integrator):
46     plt.figure(figsize=fig_dims)
47     plt.plot(z, initial[1::2], label='0 hours')
48     for solution, label in zip(c2, labels):
49         if "240" in label:
50             break
51     plt.plot(z, solution, label=label)
52     plt.ylabel('$c_2$')
53     plt.xlabel('z (km)')
54     plt.legend()
55     save_name = integrator + ' c2.pdf'
56     save_plot(save_name)
57
58
59 def plot_40km(t, c1_40km, c2_40km, integrator):
60     c2_40km_scaled = [1E-4 * val for val in c2_40km]
61
62     plt.figure(figsize=fig_dims)
63     plt.plot(t, c1_40km, label='$c_1$')
64     plt.plot(t, c2_40km_scaled, label='$c_2$ * 1E-4')
65     plt.xlabel('t (s)')
66     plt.legend()
67     plt.xlim([0, t[-1]])
68
69     save_name = integrator + ' time.pdf'
70     save_plot(save_name)

```

Listing 2: Code to generate pretty plots