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

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.

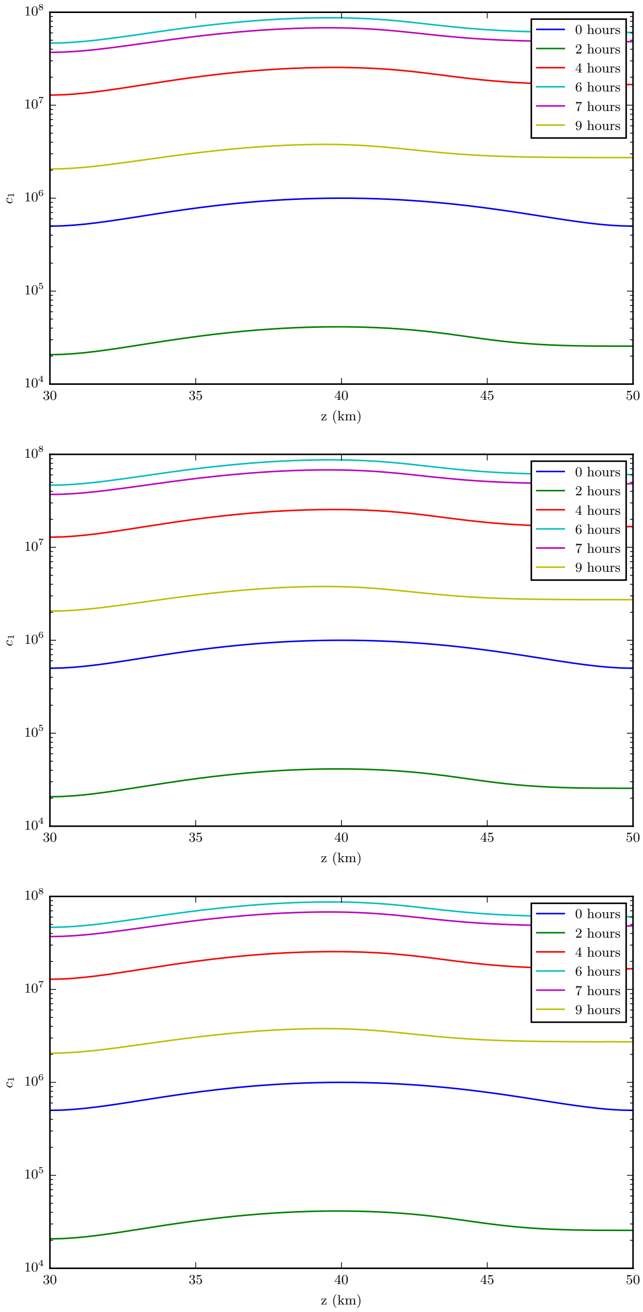

Fig. 1: c_1

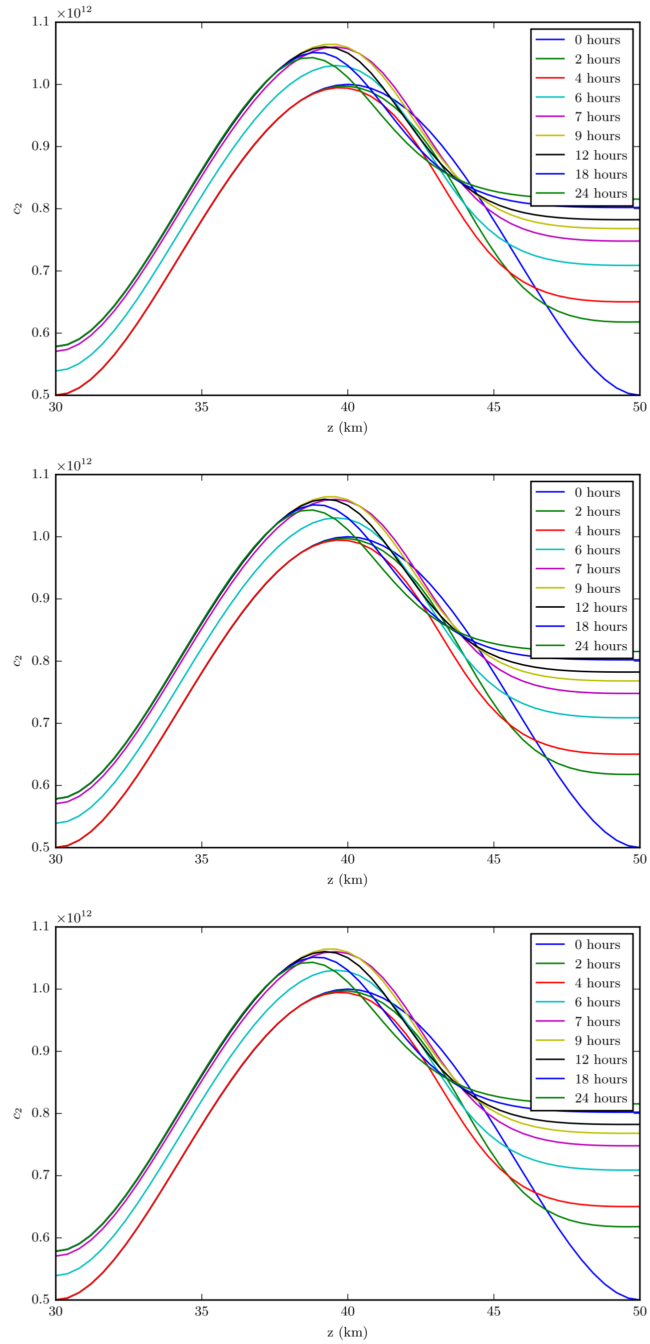

Fig. 2: c_2

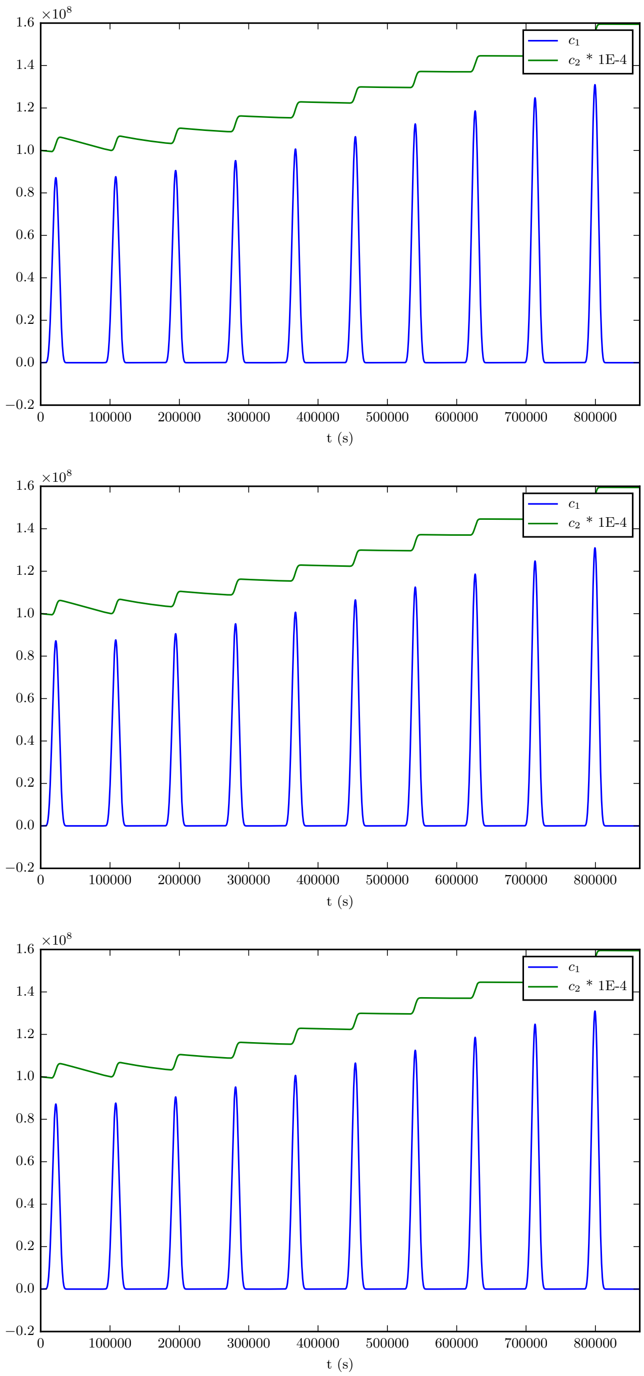
×10^8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
c1
c2 * 1E-4
0
100000
200000
300000
400000
500000
600000
700000
800000
t (s)

Fig. 3: 10 days

Appendix A: Python Code

```
import numpy as np
from scipy.integrate import ode
from time import clock
from scipy.sparse import spdiags
import json
from PrettyPlots import *


def K(z):
    return 10E-8 * np.exp(z / 5.)


def gamma(z):
    return 1. - ((z - 40.) / 10.) ** 2 + (1. / 2.) * ((z - 40.) / 10.) ** 4


def R(y_1, y_2, t):
    '''
    Find the reaction rates, R_1 and R_2, of the system at state c and time t.
    '''

    if np.sin(w * t) > 0:
        k_3 = np.exp(-a_3 / np.sin(w * t))
        k_4 = np.exp(-a_4 / np.sin(w * t))
    else:
        k_3 = 0
        k_4 = 0

    R_1 = -k_1 * y_1 * y_3 - k_2 * y_1 * y_2 + 2. * k_3 * y_3 + k_4 * y_2
    R_2 = +k_1 * y_1 * y_3 - k_2 * y_1 * y_2 - k_4 * y_2
    return R_1, R_2


def system(t, y):
    f = np.zeros(len(y))

    R1, R2 = R(y[0], y[1], t)
    l_p, l_m = 3. / 2., 1. / 2.

    f[0] = (dz ** -2 * (K(l_p) * y[2] - (K(l_p) + K(l_m)) * y[0] + K(l_m) * y[2]) + R1)
    f[1] = (dz ** -2 * (K(l_p) * y[3] - (K(l_p) + K(l_m)) * y[1] + K(l_m) * y[3]) + R2)

    for i in range(1, M):
        R1, R2 = R(y[2 * i], y[2 * i + 1], t)
        l_p, l_m = i + 3. / 2., i + 1. / 2.

        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])
        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])

    R1, R2 = R(y[2 * M - 2], y[2 * M - 1], t)
    l_p, l_m = M + 1. / 2., M - 1. / 2.

    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)
    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)

    return f


def jacobian(t, y):
    main = np.zeros(len(y))
    sub_1, sub_2 = np.zeros(len(y)), np.zeros(len(y))
    sup_1, sup_2 = np.zeros(len(y)), np.zeros(len(y))

    if np.sin(w * t) > 0:
        k_4 = np.exp(-a_4 / np.sin(w * t))
    else:
        k_4 = 0
```

```
    sup_2[-2] = dz ** -2 * (K(M + 1./2.) + K(M - 1./2.))
    sup_2[-1] = sup_2[-2]
    for i in range(2, 2 * M):
        sup_2[i] = dz ** -2 * K(i + 1. + 1./2.)

    for i in range(1, M + 1):
        sup_1[2 * i]     = -k_2 * y[2 * i] + k_4
        sup_1[2 * i + 1] = 0

    for i in range(0, M + 1):
        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]
        main[2 * i + 1] =                                                       - k_2 * y[2 * i]     - k_4

    for i in range(0, M):
        sub_1[2 * i]     = k_1 * y_3 - k_2 * y[2 * i + 1]
        sub_1[2 * i + 1] = 0

    sub_2[0:2] = dz ** -2 * K(1. - 1./2.)
    for i in range(2, 2 * M):
        sub_2[i] = dz ** -2 * (K(i + 1. + 1./2.) + K(i + 1 - 1./2.))

    diag_rows = np.array([sup_2, sup_1, main, sub_1, sub_2])
    positions = [2, 1, 0, -1, -2]
    jac = spdiags(diag_rows, positions, len(y), len(y)).todense()

    return jac


def solve(solver, c, integrator):
    # Create result arrays
    c1, c2, c1_40km, c2_40km, t = [], [], [], [], []

    start_time = clock()
    for i in range(0, len(times) - 1):
        # Initial and final time
        t_0 = times[i]
        t_f = times[i + 1]

        # Solver setup
        sol = []
        solver.set_initial_value(c, t_0)
        while solver.successful() and solver.t < t_f:
            solver.integrate(solver.t + dt)
            sol.append(solver.y)

            # keep time history for 40km point
            one, two = sol[-1][0::2], sol[-1][1::2]
            mid_one, mid_two = one[M / 2], two[M / 2]
            c1_40km.append(mid_one), c2_40km.append(mid_two)
            t.append(solver.t)

            print "{0:3.2f}%".format(clock(), 100. * t[-1] / times[-1])

        # Save c1, c2 solutions
        c1.append(one), c2.append(two)

        #Update initial conditions for next iteration
        c = sol[-1]

    elapsed_time = clock() - start_time
    print(elapsed_time, "seconds process time")

    output = [c1, c2, c1_40km, c2_40km, t]
    return output


def run_trials():
    number_of_solvers = 4
```

```
    for trial in range(number_of_solvers):
        # Set up ODE solver
        if trial == 0:
            solver = ode(system)
            integrator = 'dop853'
            solver.set_integrator(integrator, atol=1E-1, rtol=1E-3)
        elif trial == 1:
            solver = ode(system)
            integrator = 'dopri5'
            solver.set_integrator(integrator, atol=1E-1, rtol=1E-3)
        elif trial == 2:
            solver = ode(system)
            integrator = 'bdf'
            solver.set_integrator('vode', method=integrator, atol=1E-1, rtol=1E-3, nsteps=1000)
        elif trial == 3:
            solver = ode(system, jacobian)
            integrator = 'bdf Jacobian'
            solver.set_integrator('vode', method=integrator.split(' ')[0], atol=1E-1, rtol=1E-3, nsteps=1000, with_ja

        print("Starting solver: ", integrator)
        c1, c2, c1_40km, c2_40km, t = solve(solver, c, integrator)

        # And plot some things
        plot_c1(z, c, c1, labels, integrator)
        plot_c2(z, c, c2, labels, integrator)
        plot_40km(t, c1_40km, c2_40km, integrator)

# Basic problem parameters
M = 50              # Number of sections
dz = 20. / M        # 20km divided by M subsections
z = [30. + j * dz for j in range(M + 1)]

y_3 = 3.7E16        # Concentration of O_2 (constant)
k_1 = 1.63E-16      # Reaction rate [O + O_2 ->     O_3]
k_2 = 4.66E-16      # Reaction rate [O + O_3 -> 2 * O_2]
a_3 = 22.62         # Constant used in calculation of k_3
a_4 = 7.601         # Constant used in calculation of k_4
w = np.pi / 43200.  # Cycle (half a day) [1/sec]

# This generates the initial conditions
c = np.zeros(len(2 * z))
for i, _ in enumerate(z):
    c[2 * i]     = 1E6  * gamma(z[i])
    c[2 * i + 1] = 1E12 * gamma(z[i])

# Time array
times = 3600. * np.array([0., 2., 4., 6., 7., 9., 12., 18., 24., 240.])
dt = 60.

labels = [str(int(x / 3600.)) + " hours" for x in times[1:]]

run_trials()
```

Listing 1: Code to create solutions

```
import numpy as np
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import os

# Configure figures for production
WIDTH = 495.0  # the number latex spits out
FACTOR = 1.0   # the fraction of the width the figure should occupy
fig_width_pt = WIDTH * FACTOR

inches_per_pt = 1.0 / 72.27
golden_ratio = (np.sqrt(5) - 1.0) / 2.0      # because it looks good
fig_width_in = fig_width_pt * inches_per_pt  # figure width in inches
```

```
fig_height_in = fig_width_in * golden_ratio   # figure height in inches
fig_dims = [fig_width_in, fig_height_in]  # fig dims as a list


def save_plot(save_name):
    # Save plots
    try:
        os.mkdir('figures')
    except Exception:
        pass

    plt.savefig('figures/' + save_name, bbox_inches='tight')
    plt.close()


def plot_c1(z, initial, c1, labels, integrator):
    plt.figure(figsize=fig_dims)
    plt.plot(z, initial[0::2], label='0 hours')
    for solution, label in zip(c1, labels):
        if "12" in label:
            break
        plt.plot(z, solution, label=label)
    plt.ylabel('$c_1$')
    plt.xlabel('z (km)')
    plt.yscale('log')
    plt.legend()
    save_name = integrator + ' c1.pdf'
    save_plot(save_name)


def plot_c2(z, initial, c2, labels, integrator):
    plt.figure(figsize=fig_dims)
    plt.plot(z, initial[1::2], label='0 hours')
    for solution, label in zip(c2, labels):
        if "240" in label:
            break
        plt.plot(z, solution, label=label)
    plt.ylabel('$c_2$')
    plt.xlabel('z (km)')
    plt.legend()
    save_name = integrator + ' c2.pdf'
    save_plot(save_name)


def plot_40km(t, c1_40km, c2_40km, integrator):
    c2_40km_scaled = [1E-4 * val for val in c2_40km]

    plt.figure(figsize=fig_dims)
    plt.plot(t, c1_40km, label='$c_1$')
    plt.plot(t, c2_40km_scaled, label='$c_2$ * 1E-4')
    plt.xlabel('t (s)')
    plt.legend()
    plt.xlim([0, t[-1]])

    save_name = integrator + ' time.pdf'
    save_plot(save_name)
```

Listing 2: Code to generate pretty plots