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,

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

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

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

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

$$\begin{aligned} R_1(c_1,c_2,t) =& -k_1c_1c_3 - k_2c_1c_2 + 2k_3(t)c_3 + k_4(t)c_2 \\ R_2(c_1,c_2,t) =& k_1c_1c_3 - k_2c_1c_2 - k_4(t)c_2 \end{aligned} \tag{3}$$

with,

$$\begin{aligned} k_1 =& 1.63x10^{16} \\ k_2 =& 4.66x10^{16} \\ k_l =& \exp[a_l/\sin\omega t] \text{ if } \sin\omega t > 0, \text{ else } 0 \ (l = 3,4) \end{aligned}$$

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

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

where

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

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

$$\begin{aligned} \dot{c}^i_j = (\Delta z)^{-2}[&K_{j+1/2}c^i_{j+1} - (K_{j+1/2} + K_{j-1/2})c^i_j + \\ &K_{j-1/2}c^i_{j-1}] + R^i(\boldsymbol{c},t), \end{aligned} \tag{6}$$

where $K_{j\pm1/2} = K(30 + [j \pm 1/2]\Delta z)$. A system of $2M$ ODEs is then specified by setting $\boldsymbol{y}(t) = [c^1_1(t), c^2_1(t), c^1_2(t), c^2_2(t), \ldots, c^1_M(t), c^2_M(t)]^T$, with boundary conditions $c^i_0 = c^i_2$ and $c^i_{M-1} = c^i_{M+1}$. The two parabolic PDEs are then reduced to a system of 2M ODEs of the form $\dot{\boldsymbol{y}} = \boldsymbol{f}(t,\boldsymbol{y})$.

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.

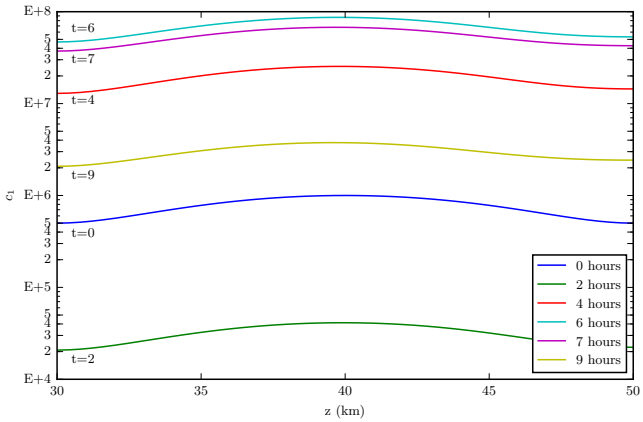

Fig. 1: c_1 vs. z at $t = 0$, 2, 4, 6, 7, and 9 hours

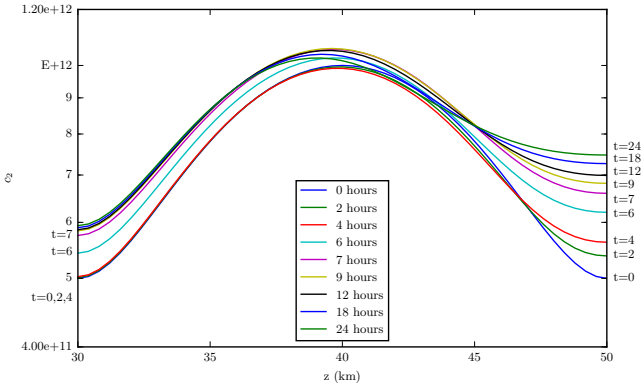

Fig. 2: c_2 vs. z at $t = 0$, 2, 4, 6, 7, 9, 12, 18, and 24 hours

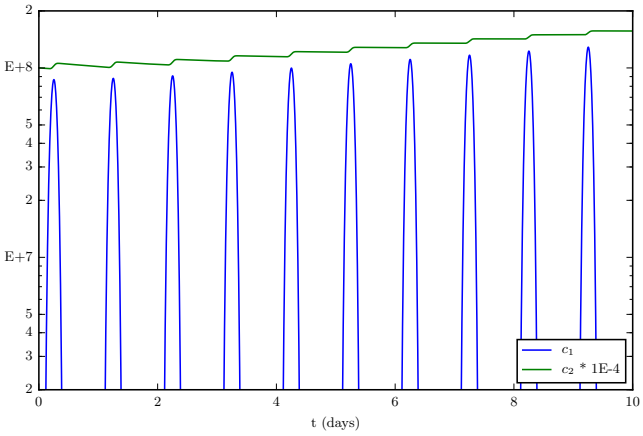

Fig. 3: c_1 and c_2 vs. time (from 0 to 10 days) at $z = 40$ km, c_2 is scaled by 1E-4

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 save when plotted on a logarithmic scale. The results from 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

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

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,

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

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 (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 (bdf solver)

and the Normalized Root Mean Square error,

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

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.

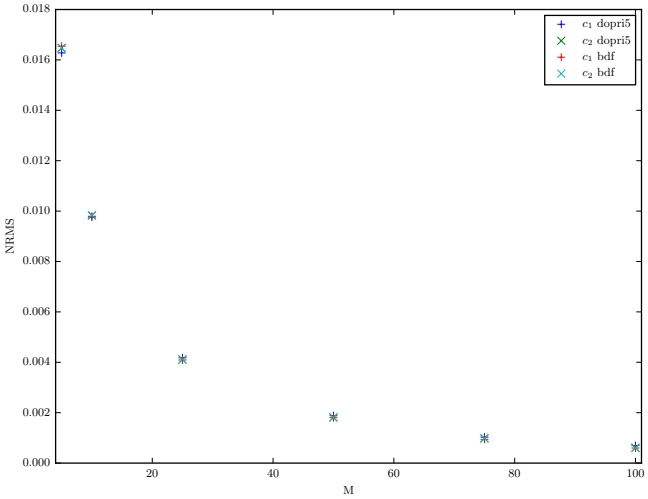

Fig. 4: NRMS for M = 5, 10, 25, 50, 75, and 100 compared against M = 200 for both solvers and c_1 and c_2

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.

Appendix A: Python Code

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


def K(j):
    return 1E-8 * np.exp((30. + j * dz) / 5.)


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


def R(y_1, y_2, 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]) + R1)
        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)

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

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

    return f


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

    start_time = clock()
    for i in range(0, len(time) - 1):
        # Initial and final time
        t_0 = time[i]
        t_f = time[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 "{0:03.2f}%".format(100. * solver.t / time[-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 save_variables(name, z, c1, c2, t, c1_40km, c2_40km):
    try:
        os.mkdir('data')
    except Exception:
        pass

    try:
        os.mkdir('data/' + name)
    except Exception:
        pass

    np.savetxt('data/' + name + '/z.csv', z)
    np.savetxt('data/' + name + '/c1.csv', c1)
    np.savetxt('data/' + name + '/c2.csv', c2)
    np.savetxt('data/' + name + '/t.csv', t)
    np.savetxt('data/' + name + '/c1_40km.csv', c1_40km)
    np.savetxt('data/' + name + '/c2_40km.csv', c2_40km)


def load_variables(name):
    z       = np.loadtxt('data/' + name + '/z.csv')
    c1      = np.loadtxt('data/' + name + '/c1.csv')
    c2      = np.loadtxt('data/' + name + '/c2.csv')
    t       = np.loadtxt('data/' + name + '/t.csv')
    c1_40km = np.loadtxt('data/' + name + '/c1_40km.csv')
    c2_40km = np.loadtxt('data/' + name + '/c2_40km.csv')

    return z, c1, c2, t, c1_40km, c2_40km


def run_trials(z, integrators, times, M):
    # Set up ODE solver
    for integrator in integrators:
        if integrator == 'dop853' or integrator == 'dopri5':
            solver = ode(system)
            solver.set_integrator(integrator, atol=1E-1, rtol=1E-3)
        elif integrator == 'bdf':
            solver = ode(system)
            solver.set_integrator('vode', method=integrator, atol=1E-1, rtol=1E-3, nsteps=2000)

        name = integrator + ' ' + str(times[-1]) + ' ' + str(M)
        try:
```

```
            z, c1, c2, t, c1_40km, c2_40km = load_variables(name)
            print "Loaded data for: " + name
        except:
            print "Starting solver: ", integrator, "with times", times
            c1, c2, c1_40km, c2_40km, t = solve(solver, c, times, integrator)
            save_variables(name, z, c1, c2, t, c1_40km, c2_40km)

        # And plot some things
        if times[-1] == 86400.0:
            labels = [str(int(time / 3600.)) + " hours" for time in times[1:]]
            plot_c1(z, c, c1, labels, name)
            plot_c2(z, c, c2, labels, name)
        elif times[-1] == 864000.0:
            plot_40km(t, c1_40km, c2_40km, name)


def sensitivity_analysis(integrators, times, meshes):
    plt.figure()
    for integrator in integrators:
        z_M, c1_M, c2_M = [], [], []

        for M in meshes:
            name = integrator + ' ' + str(times[-1]) + ' ' + str(M)
            try:
                z, c1, c2, _, _, _ = load_variables(name)
            except Exception:
                print Exception
            z_M.append(list(z))
            c1_M.append(list(c1[-1]))
            c2_M.append(list(c2[-1]))

        best_z = z_M[-1]
        best_c1, best_c2 = c1_M[-1], c2_M[-1]
        NRMS1, NRMS2 = [], []
        for j, mesh in enumerate(z_M):
            if j + 1 == len(z_M): break  # RMS with yourself is silly
            best1, best2, curr1, curr2 = [], [], [], []
            for i, element in enumerate(best_z):
                if element in mesh:
                    best1.append(best_c1[i])
                    best2.append(best_c2[i])
                    curr1.append(c1_M[j][mesh.index(element)])
                    curr2.append(c2_M[j][mesh.index(element)])

            best1, best2 = np.array(best1), np.array(best2)
            curr1, curr2 = np.array(curr1), np.array(curr2)

            err1, err2 = curr1 - best1, curr2 - best2
            NRMS1.append(np.sqrt(np.mean(np.square(err1)))/(max(best1) - min(best1)))
            NRMS2.append(np.sqrt(np.mean(np.square(err2)))/(max(best2) - min(best2)))

        x = [mesh for mesh in meshes][0:-1]
        plt.plot(x, NRMS1, '+', label='$c_1$ ' + integrator)
        plt.plot(x, NRMS2, 'x', label='$c_2$ ' + integrator)

    plt.ylabel('NRMS')
    plt.xlabel('M')
    plt.xlim([meshes[0] - 1, meshes[-2] + 1])
    plt.legend()
    save_name = str(meshes) + '.pdf'
    save_plot(save_name)


# Basic problem parameters
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]
dt = 60.              # seconds

# Base Case
M = 50                # Number of sections
dz = 20. / M          # 20km divided by M subsections

# This generates the initial conditions
c = np.zeros(2 * (M + 1))
z = np.zeros(M + 1)
for j in range(0, M + 1):
    z[j] = 30. + j * dz
    c[2 * j]     = 1E6  * gamma(z[j])
    c[2 * j + 1] = 1E12 * gamma(z[j])

# Run the trials
integrators = ['dopri5', 'bdf']
times = 3600. * np.array([0., 2., 4., 6., 7., 9., 12., 18., 24.])
run_trials(z, integrators, times, M)

# integrators = ['dopri5', 'bdf']
# times = 3600. * np.array([0., 2., 4., 6., 7., 9., 12., 18., 240.])
# run_trials(z, integrators, times, M)

# # Mesh Analysis
# meshes = [5, 10, 25, 50, 75, 100, 200]
# for M in meshes:
#     dz = 20. / M         # 20km divided by M subsections

#     # This generates the initial conditions
#     c = np.zeros(2 * (M + 1))
#     z = np.zeros(M + 1)
#     for i in range(0, M + 1):
#         z[i] = 30. + i * dz
#         c[2 * i]     = 1E6  * gamma(z[i])
#         c[2 * i + 1] = 1E12 * gamma(z[i])

#     # Time array
#     dt = 60.

#     integrators = ['dopri5', 'bdf']
#     times = 3600. * np.array([0., 2., 4.])
#     run_trials(z, integrators, times, M)

# sensitivity_analysis(integrators, times, meshes)
```

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.ylim([1E4, 1E8])
    plt.yticks([1E4, 2E4, 3E4, 4E4, 5E4,
                1E5, 2E5, 3E5, 4E5, 5E5,
                1E6, 2E6, 3E6, 4E6, 5E6,
                1E7, 2E7, 3E7, 4E7, 5E7, 1E8],
               ['E+4', '2', '3', '4', '5',
                'E+5', '2', '3', '4', '5',
                'E+6', '2', '3', '4', '5',
                'E+7', '2', '3', '4', '5', 'E+8'])
    plt.legend(loc='lower right')

    plt.text(30.5, 1.5e+4, 't=2', fontsize=9, family='serif')
    plt.text(30.5, 3.5e+5, 't=0', fontsize=9, family='serif')
    plt.text(30.5, 1.5e+6, 't=9', fontsize=9, family='serif')
    plt.text(30.5, 1.e+7,  't=4', fontsize=9, family='serif')
    plt.text(30.5, 2.8e+7, 't=7', fontsize=9, family='serif')
    plt.text(30.5, 6.e+7,  't=6', fontsize=9, family='serif')

    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.yscale('log')
    plt.ylim([4.E11, 1.2E12])
    plt.yticks([4E11, 5E11, 6E11, 7E11, 8E11, 9E11, 1E12, 1.2E12],
               ['4.00e+11', '5', '6', '7', '8', '9', 'E+12', '1.20e+12'])
    plt.legend(loc='best')

    # Left side text
    plt.text(28.25, 4.65e+11, 't=0,2,4', fontsize=9, family='serif')
    plt.text(29, 5.4e+11,     't=6',     fontsize=9, family='serif')
    plt.text(29, 5.7e+11,     't=7',     fontsize=9, family='serif')

    # Right side text
    plt.text(50.25, 4.95e+11, 't=0',  fontsize=9, family='serif')
    plt.text(50.25, 5.35e+11, 't=2',  fontsize=9, family='serif')
    plt.text(50.25, 5.6e+11,  't=4',  fontsize=9, family='serif')
    plt.text(50.25, 6.1e+11,  't=6',  fontsize=9, family='serif')
    plt.text(50.25, 6.4e+11,  't=7',  fontsize=9, family='serif')
    plt.text(50.25, 6.7e+11,  't=9',  fontsize=9, family='serif')
    plt.text(50.25, 7.0e+11,  't=12', fontsize=9, family='serif')
    plt.text(50.25, 7.3e+11,  't=18', fontsize=9, family='serif')
```

```
    plt.text(50.25, 7.6e+11,  't=24', fontsize=9, family='serif')

    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]
    days = [val / 86400. for val in t]

    plt.figure(figsize=fig_dims)
    plt.plot(days, c1_40km, label='$c_1$')
    plt.plot(days, c2_40km_scaled, label='$c_2$ * 1E-4')
    plt.xlabel('t (days)')
    plt.yscale('log')
    plt.ylim([2.E6, 2E8])
    plt.yticks([2E6, 3E6, 4E6, 5E6, 1E7, 2E7, 3E7, 4E7, 5E7, 1E8, 2E8],
               ['2', '3', '4', '5', 'E+7', '2', '3', '4', '5', 'E+8', '2'])
    plt.xlim([0, days[-1]])
    plt.legend(loc='lower right')
    save_name = integrator + ' time.pdf'
    save_plot(save_name)
```

Listing 2: Code to generate pretty plots