Compact Notation Derivation for Simple Elementary Reactions & their Numerical Solution

Raunak Chakraborty

November 6, 2021

Contents

Chapter 1

Compact Notation Derivation

1.1 Introduction

Usually, we represent Chemical Reactions as One-Step Processes, however, in reality, the Reactants may go through a large number of Intermediary Reactions & thus producing a large number of Intermediary Species. The specific path that the reaction will follow depend on the Initial Conditions (Temparature, Pressure etc) of the Reactants, as well as the time for which the Reaction is allowed to take place. This is often referred to as a Reaction Mechanism & study of such Mechanisms is termed as Chemical Kinetics. In this project, we shall consider a simple multi-step Reaction Mechanism & try to convert them into a System of Ordinary Differential Equations, which will later be solved using Numerical Integration.

1.2 Indexing all Species & Reactions

We consider here the following multi-step Reaction Mechanism :

$$\begin{aligned} CO + O_2 &\longleftrightarrow CO_2 + O \\ O + H_2O &\longleftrightarrow OH + OH \\ CO + OH &\longleftrightarrow CO_2 + H \\ H + O_2 &\longleftrightarrow OH + O \end{aligned}$$

The first step will be to assign a unique index to each of the

species as well as reactions. Here, i represents the indices for reactions, whereas j represents the indices for species.

i	Reactions
1	R1
2	R2
3	R3
4	R4

(a) Reaction Indices

j	Species
1	CO
2	O_2
3	CO_2
4	O
5	H_2O
6	OH
7	H

(b) Species Indices

Table 1.1: Assigning Indices to Reactions & Species

1.3 Stoichiometric Coefficient Matrices for Reactants & Products

Now we must create Stoichiometric Coefficient Matrices for both reactants & products, which will contain information on number of moles of every species in every reaction. They will help keep track of the concentration of different species throughout the computation.

v'_{ji} : Moles of species j in the reactant-side of Reaction i.

$$v'_{ji} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

v''_{ji} : Moles of species j in the product-side of Reaction i.

$$v''_{ji} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

$$v_{ji} = (v''_{ji} - v'_{ji}) = \begin{bmatrix} -1 & -1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 2 & 0 \\ -1 & 0 & 1 & 0 & 0 & -1 & 1 \\ 0 & -1 & 0 & 1 & 0 & 1 & -1 \end{bmatrix}$$

1.4 Net Reaction Rate for each Reaction

Here, we define the Reaction Rate (q_i), that is the speed at which a Reaction takes place.

$$\mathbf{Q_i} = (\mathbf{K_{fi}}.\prod_{j=1}^{N}[\mathbf{X_j}]^{\mathbf{v'_{ji}}}) - (\mathbf{K_{ri}}.\prod_{j=1}^{N}[\mathbf{X_j}]^{\mathbf{v''_{ji}}}) \tag{1.1}$$

where,
K_{fi} : Rate of Forward Reaction
K_{ri} : Rate of Reverse Reaction
$[X_j]^{v'_{ji}}$: Concentration per unit Volume for Reactants
$[X_j]^{v''_{ji}}$: Concentration per unit Volume for Products
N : Total Number of reactions in the Reaction Mechanism

If Q_i is positive, the reaction will proceed in the Forward Direction & vice versa. Hence, for each individual case, we have :

$$\mathbf{Q_1} = (\mathbf{K_{f1}}.[\mathbf{CO}]^{\mathbf{1}}.[\mathbf{O2}]^{\mathbf{1}}) - (\mathbf{K_{r1}}.[\mathbf{CO_2}]^{\mathbf{1}}.[\mathbf{O}]^{\mathbf{1}}) \tag{4.1.1}$$

$$\mathbf{Q_2} = (\mathbf{K_{f2}}.[\mathbf{O}]^{\mathbf{1}}.[\mathbf{H_2O}]^{\mathbf{1}}) - (\mathbf{K_{r2}}.[\mathbf{OH}]^{\mathbf{2}}) \tag{4.1.2}$$

$$\mathbf{Q_3} = (\mathbf{K_{f3}}.[\mathbf{CO}]^{\mathbf{1}}.[\mathbf{OH}]^{\mathbf{1}}) - (\mathbf{K_{r3}}.[\mathbf{CO_2}]^{\mathbf{1}}.[\mathbf{H}]^{\mathbf{1}}) \tag{4.1.3}$$

$$\mathbf{Q_4} = (\mathbf{K_{f4}}.[\mathbf{H}]^{\mathbf{1}}.[\mathbf{O_2}]^{\mathbf{1}}) - (\mathbf{K_{r4}}.[\mathbf{OH}]^{\mathbf{1}}.[\mathbf{O}]^{\mathbf{1}}) \tag{4.1.4}$$

1.5 Net Production Rate for each Species

The Net Production Rate for each species is defined as :

$$\mathbf{W_j} = \frac{d[X_j]}{dt} = \sum_{i=1}^{N}\mathbf{V_{ji}}.\mathbf{Q_i} \tag{1.2}$$

Therefore, for each species, we have :

$$\mathbf{W_1} = \frac{d[CO]}{dt} = -(Q_1 + Q_3) \tag{5.1.1}$$

$$\mathbf{W_2} = \frac{d[O_2]}{dt} = -(Q_1 + Q_4) \tag{5.1.2}$$

$$\mathbf{W_3} = \frac{d[CO_2]}{dt} = (Q_1 + Q_3) \tag{5.1.3}$$

$$\mathbf{W_4} = \frac{d[O]}{dt} = (Q_1 + Q_4 - Q_2) \tag{5.1.4}$$

$$\mathbf{W_5} = \frac{d[HO_2]}{dt} = -Q_2 \tag{5.1.5}$$

$$\mathbf{W_6} = \frac{d[OH]}{dt} = (2.Q_2 + Q_4 - Q_3) \tag{5.1.6}$$

$$\mathbf{W_7} = \frac{d[H]}{dt} = (Q_3 - Q_4) \tag{5.1.7}$$

Writting the above set of equations in matrix notations, we get:

$$\begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \\ W_7 \end{bmatrix} = \begin{bmatrix} -1 & -1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 2 & 0 \\ -1 & 0 & 1 & 0 & 0 & -1 & 1 \\ 0 & -1 & 0 & 1 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \end{bmatrix}$$

This is the equation that we shall feed into the computer. In this way, we have converted a Chemical Reaction Mechanism into a System of Ordinary Differential Equations that will subsequently be solved using Numerical Integration. In the next section, we shall discuss the solution of a Non-Linear Coupled System of stiff ODEs using the Multivariate Newton Raphson Solver.

Chapter 2

Solving a System of ODEs using Multivariate Newton-Raphson Technique

2.1 Calculations

Let us consider the following set of Equations :

$$\frac{dy_1}{dt} = -0.04y_1 + 10^4 y_2 y_3 \tag{2.1}$$

$$\frac{dy_2}{dt} = 0.04y_1 - 10^4 y_2 y_3 - 3.10^7 y_2^2 \tag{2.2}$$

$$\frac{dy_3}{dt} = 3.10^7 y_2^2 \tag{2.3}$$

Let the above System of ODEs represent an imaginary set of chemical reactions, in which y_1, y_2 & y_3 represent the mole fractions of Species A, B & C. We will also specify a time-scale of 600 seconds. Our primary interest is to calculate the final converged values of y_1, y_2 & y_3, i.e, the mole fractions of A, B & C during the final time-step. We use Euler's Implicit Method (Backward Differencing) on the above equations to arrive at the following :

$$\frac{y_1^n - y_1^{n-1}}{dt} = -0.04y_1 + 10^4 y_2 y_3 \tag{2.4}$$

$$\frac{y_2^n - y_2^{n-1}}{dt} = 0.04y_1 - 10^4 y_2 y_3 - 3.10^7 y_2^2 \tag{2.5}$$

$$\frac{y_3^n - y_3^{n-1}}{dt} = 3.10^7 y_2^2 \tag{2.6}$$

We shall now transform this to a Minimization Problem, to be solved using Newton-Raphson Technique. Rearranging the above equations, we get :

$$f_1 = y_1^{n-1} - y_1^n + dt * (-0.04y_1 + 10^4 y_2 y_3) = 0 \tag{2.7}$$

$$f_2 = y_2^{n-1} - y_2^n + dt * (0.04y_1 - 10^4 y_2 y_3 - 3.10^7 y_2^2) = 0 \tag{2.8}$$

$$f_3 = y_3^{n-1} - y_3^n + dt * (3.10^7 y_2^2) = 0 \tag{2.9}$$

For a particular value of y, the update for the Multivariate Newton-Raphson Solver takes place as follows :

$$y^{new} = y^{old} - \alpha . J^{-1} . F \tag{2.10}$$

where,
α : Relaxation Factor
J^{-1} : Inverse of the Jacobian Matrix
F : Coloum Matrix containing f_1, f_2 & f_3
The above Equation in Matrix Form is as follows :

$$\begin{bmatrix} y_1^{new} \\ y_2^{new} \\ y_3^{new} \end{bmatrix} = \begin{bmatrix} y_1^{old} \\ y_2^{old} \\ y_3^{old} \end{bmatrix} - \alpha . \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \frac{\partial f_1}{\partial y_3} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \frac{\partial f_2}{\partial y_3} \\ \frac{\partial f_3}{\partial y_1} & \frac{\partial f_3}{\partial y_2} & \frac{\partial f_3}{\partial y_3} \end{bmatrix} . \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

The Jacobian Matrix above is calculated using Numerical Differentiation as follows :

$$\frac{\partial f_1}{\partial y_1} = \frac{f(y_1 + h) - f(y_1)}{h} \tag{2.11}$$

where, h is the step size. We implement this with the aid of the following code:

2.2 Code & Results

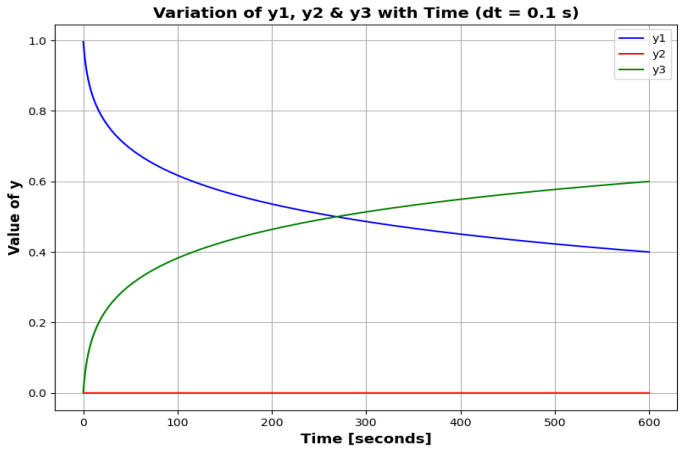
```
# Program to solve a Coupled System of ODEs
using Multivariate Newton-Raphson Solver :

""""
We consider the following Equations :
    d(y1)/dt = (-0.04*y1) + (1e4*y2*y3)
    d(y2)/dt = (0.04*y1) - (1e4*y2*y3) - (3*1e7*(y2)^2)
    d(y3)/dt = (3*1e7*(y2)^2)
After applying Backward Differencing, we arrive at the following :
    f1 = y1_old - y1 + dt*((-0.04*y1) + (1e4*y2*y3)) = 0
    f2 = y2_old-y2+dt*((0.04*y1)-(1e4*y2*y3)-(3*1e7*pow(y2,2))=0
    f3 = y3_old - y3 + dt*(3e7 * pow(y2, 2)) = 0
We shall try to solve for y1, y2, y3 using the Multivariate
Newton-Raphson Solver, where the update occurs as follows :
    X_new=X_old-(Relaxation Factor)*(Inverse of the Jacobian Matrix)
          *(Matrix containing old values)
 => x_(n+1) = x_n - (apha) * [J]^(-1) * f(x_n)
 """"


import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt

#############################################################################

# Defining the 3 Non-Linear Equations & their Jacobian Matrix

def f1(y1, y2, y3, y1_old, dt) :
    return (y1_old - y1 + dt * ((-0.04 * y1) + (1e4 * y2 * y3)))

def f2(y1, y2, y3, y2_old, dt) :
    return (y2_old-y2+dt*((0.04*y1)-(1e4*y2*y3)-(3*1e7*pow(y2,2))))

def f3(y1, y2, y3, y3_old, dt) :
    return (y3_old - y3 + dt * (3e7 * pow(y2, 2)))

# Jacobain Matrix calculated using Numerical Differentiation
```

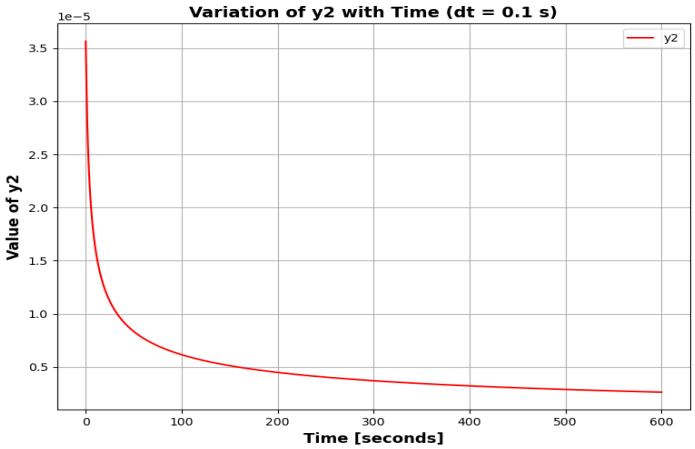
```
def Jacob(y1, y2, y3, y1_old, y2_old, y3_old, dt) :
h = 1e-8        # Step-Size for Numerical Differentiation
J = np.ones((3, 3))
# Row 1 :
J[0,0] = (f1(y1+h,y2,y3,y1_old,dt) - f1(y1,y2,y3,y1_old,dt))/h
J[0,1] = (f1(y1,y2+h,y3,y1_old,dt) - f1(y1,y2,y3,y1_old,dt))/h
J[0,2] = (f1(y1,y2,y3+h,y1_old,dt) - f1(y1,y2,y3,y1_old,dt))/h
# Row 2 :
J[1,0] = (f2(y1+h,y2,y3,y2_old,dt) - f2(y1,y2,y3,y2_old,dt))/h
J[1,1] = (f2(y1,y2+h,y3,y2_old,dt) - f2(y1,y2,y3,y2_old,dt))/h
J[1,2] = (f2(y1,y2,y3+h,y2_old,dt) - f2(y1,y2,y3,y2_old,dt))/h
# Row 3 :
J[2,0] = (f3(y1+h,y2,y3,y3_old,dt) - f3(y1,y2,y3,y3_old,dt))/h
J[2,1] = (f3(y1,y2+h,y3,y3_old,dt) - f3(y1,y2,y3,y3_old,dt))/h
J[2,2] = (f3(y1,y2,y3+h,y3_old,dt) - f3(y1,y2,y3,y3_old,dt))/h

return J

#########################################################################

# Following are the Initial Guess Values for y1, y2 & y3
y1_old = 1
y2_old = 0
y3_old = 0

# Creating a Matrix for the above values
Y_old = np.ones((3, 1))
Y_old[0] = y1_old
Y_old[1] = y2_old
Y_old[2] = y3_old

F = np.copy(Y_old)
dt = 0.1        # Time-Step Size
t = np.arange(0, 600, dt)   # creating the Time Array
alpha = 0.8     # Relaxation Factor
iter = 1        # Initial Iteration Value

# Arrays for storing y1,y2,y3 & error values for each time-step
y1_new = []
```

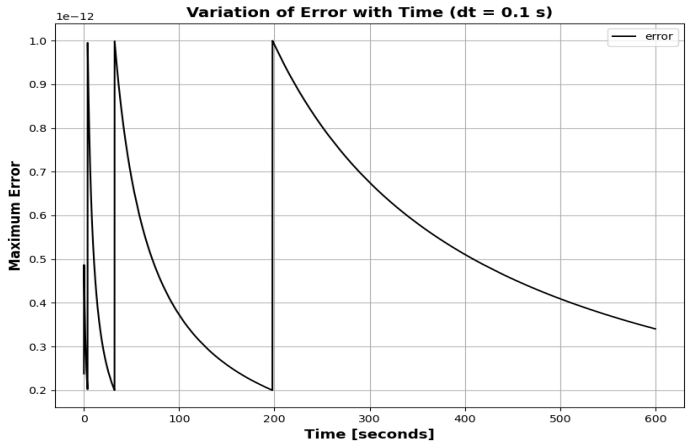
```
y2_new = []
y3_new = []
error = []

##########################################################################

# Outer Time Loop
for i in range(0, len(t)) :

    err = 1e-09
    tol = 1e-12

    # Inner Newton-Raphson Loop
    while err > tol :

        # Constructing F by sending old values to f1,f2,f3
        F[0] = f1(Y_old[0], Y_old[1], Y_old[2], y1_old, dt)
        F[1] = f2(Y_old[0], Y_old[1], Y_old[2], y2_old, dt)
        F[2] = f3(Y_old[0], Y_old[1], Y_old[2], y3_old, dt)

        # Constructing J by sending old values to Jacobian
        J=Jacob(Y_old[0],Y_old[1],Y_old[2],y1_old,y2_old,y3_old,dt)

        # Matrix Y_new is calculated using Matrices Y_old,J & F
        Y_new = Y_old - alpha * np.matmul(inv(J), F)

        # Defining Error as the Maximum(Absolute(Y_new - Y_old))
        err = np.max(np.abs(Y_new - Y_old))

        # After each Iteration of the Inner Loop, the resulting
        # Y_new Matrix is set to Y_old

        Y_old = Y_new
        iter = iter + 1
    # Inner Loop End

    log_message='Time={0} ,y1={1} ,y2={2} ,y3={3} ,error={4}'
    .format(t[i], Y_new[0], Y_new[1], Y_new[2], err)
    print(log_message)
```

```
    # Value Update fotr the Outer Loop
    Y_old = Y_new

    # Storing newly-calculated values in Arrays
    y1_new.append(Y_new[0])
    y2_new.append(Y_new[1])
    y3_new.append(Y_new[2])
    error.append(err)

    # Updating the values which make up the Y_old Matrix
    y1_old = Y_new[0]
    y2_old = Y_new[1]
    y3_old = Y_new[2]
# Outer Loop End

##########################################################################

# Plotting the values of y1, y2 & y3 with Time
plt.plot(t, y1_new, '-', color = 'blue', label = 'y1')
plt.plot(t, y2_new, '-', color = 'red', label = 'y2')
plt.plot(t, y3_new, '-', color = 'green', label = 'y3')
plt.legend()
plt.grid('on')
plt.xlabel('Time[seconds]',fontsize='12',fontweight='bold')
plt.ylabel('Value of y',fontsize ='12',fontweight='bold')
plt.title('y1,y2 & y3 VS Time',fontsize='13',fontweight='bold')
plt.show()

# Plotting Error with Time
plt.plot(t, error, '-', color = 'black', label = 'error')
plt.legend()
plt.grid('on')
plt.xlabel('Time[seconds]',fontsize='12',fontweight='bold')
plt.ylabel('Maximum Error',fontsize='12',fontweight='bold')
plt.title('Error VS Time',fontsize='13',fontweight='bold')
plt.show()
```

The output for dt = 0.1 s are the following :


Variation of y1, y2 & y3 with Time (dt = 0.1 s)
y1
y2
y3
Value of y
Time [seconds]


Variation of y2 with Time (dt = 0.1 s)
1e−5
y2
Value of y2
Time [seconds]


Variation of Error with Time (dt = 0.1 s)
1e−12
error
Maximum Error
Time [seconds]

Following are the final time-step values of y1, y2, y3 & error for different values of dt :

dt	**0.1 s**	**1 s**	**100 s**
y1	0.399995498	0.4002479	0.41467009
y2	2.6315555e-06	2.63471135e-06	2.79376308e-06
y3	0.60004239	0.59974946	0.58532712
error	3.40394379e-13	6.825651155e-13	6.6713301549e-13

2.3 Observations

The above data elucidates that Implicit Schemes are unbothered by step-size values. Even when we choose a step-size of 100, there is no significant change in the results, which is unlike any Explicit Scheme, where the solution is heavily influenced by the chosen step-size. It can be observed that y2 barely changes (as compared to y1 & y3), which makes this a System of Stiff ODEs. The solution tends to stabilize somewhere around t = 280 s, after which, there have been only small changes in values of all 3 variables. Furthermore, the error fluctuates quite a bit throughout the computation, however, it is within the acceptable range.

Chapter 3

Conclusion

This project sheds light into the basics of Computational Chemical Kinetics. First, we convert the Reaction Mechanism into a System of ODEs, followed by their solution using iterative means. Implicit Schemes are always recommended, despite the increased computational cost. I used the following platforn for this project :
Python 3.8.8 -> Anaconda -> Windows 10