

Linearisation of Biomolecular Systems

Peter Gawthrop (peter.gawthrop@unimelb.edu.au)

November 5, 2019

Contents

1	Introduction	2
1.1	Import some python code	2
2	Example system: enzyme-catalysed reaction	3
2.1	Stoichiometry	3
2.2	Steady state.	4
2.3	Parameters	4
2.4	Linearisation	4
2.5	Pole/zero analysis	6
2.6	Simulation	7
2.6.1	Compare linear and nonlinear	9
3	Example system: enzyme-catalysed reaction with product removal	10
3.1	Stoichiometry	11
3.2	Parameters	11
3.3	Steady state.	11
3.4	Linearisation	11
3.5	Pole/zero analysis	13
3.6	Simulation	14

Note: this is the Linearisation.ipynb notebook. The PDF version "Linearisation of Biomolecular Systems" is available [here](#).

1 Introduction

As discussed by (Gawthrop and Crampin, 2016):

"The bond graph approach gives the set of *nonlinear* ordinary differential equations describing the biomolecular system being modelled. Linearisation of non-linear systems is a standard technique in control engineering: as discussed by (Goodwin et al., 2001), "The incentive to try to approximate a nonlinear system by a linear model is that the science and art of linear control is vastly more complete and simpler than they are for the nonlinear case.". Nevertheless, it is important to realise that conclusions drawn from linearisation can only be verified using the full *nonlinear* equations."

This notebook tutorial examines the linearisation of bond graph models in the context of biomolecular systems using two simple examples:

1. An enzyme-catalysed reaction
2. An enzyme-catalysed reaction with product removal

As will be seen, the first example is actually linear given the particular choice of chemostats; the second is non-linear the deviation from linearity is examined.

Linearisation of a dynamic system $\dot{x} = f(x, v)$ is with reference to a steady state defined by constant states $x = x_{ss}$ and constant flows $v = v_{ss}$ such that $f(x_{ss}, v_{ss}) = 0$. In general, determination of steady-states is a difficult problem and can only be determined numerically. Some theoretical results are given by (Gawthrop, 2018). In this tutorial, steady-states can be determined theoretically.

1.1 Import some python code

The bond graph analysis uses a number of Python modules:

```
In [1]: ## Some useful imports
import BondGraphTools as bgt
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
import IPython.display as disp

## Stoichiometric analysis
import stoich as st

## SVG bg representation conversion
import svgBondGraph as sbg

## Control systems package
import control as con

## Set quiet=False for verbose output
```

```

quiet = True

## Set slycot=True if slycot is installed (see control module)
slycot=True

```

2 Example system: enzyme-catalysed reaction

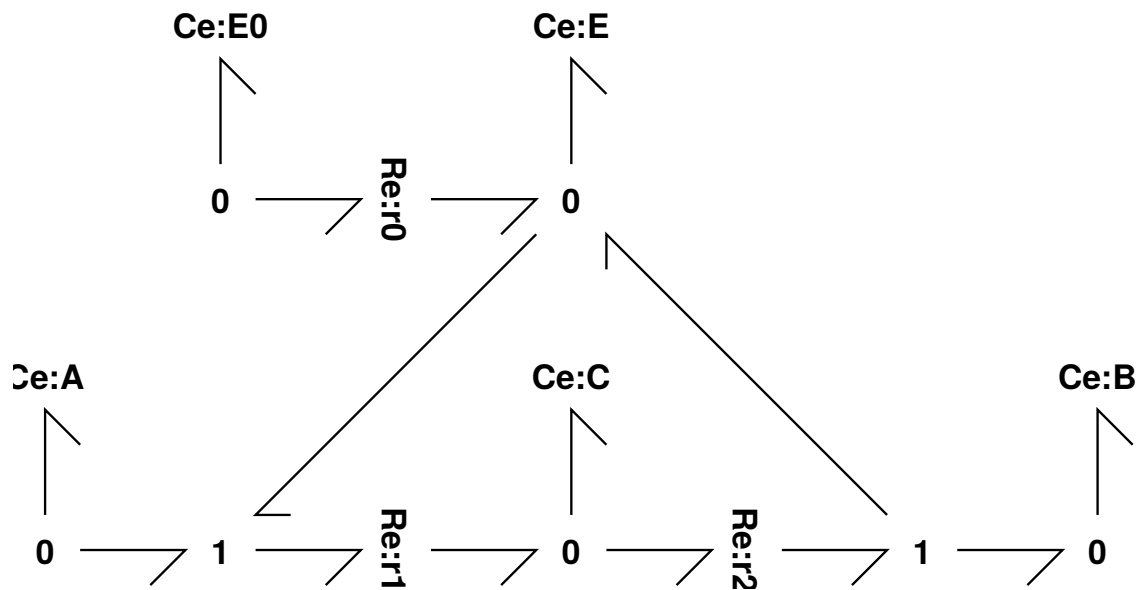
The bond graph representation of the (reversible) enzyme-catalysed reaction is given by (Gawthrop and Crampin, 2014) and is discussed in the tutorial [ECR](#).

```

In [2]: sbg.model('eRE_abg.svg')
import eRE_abg
disp.SVG('eRE_abg.svg')

```

Out[2]:



2.1 Stoichiometry

```

In [3]: s = st.stoich(eRE_abg.model(),quiet=quiet)
print('species:',s['species'])
chemostats=['A','B','E0']
disp.Latex(st.sprintrl(s))
sc = st.statify(s,chemostats=chemostats)

```

species: ['A', 'B', 'C', 'E', 'E0']

2.2 Steady state.

In this particular case, the system steady state can be found theoretically assuming constant amounts of A and B (Gawthrop and Crampin, 2014). Function ssECR does this:

```
In [4]: def ssECR(x_A,x_B,e0=1,
               K_A = 1,K_B=1,K_C=1,K_E=1, K_E0 = 1e-3,
               kappa_r1 = 1,kappa_r2=1):
    """Theoretical steady state of Enzyme-catalysed Reactions
    """

    kappa_bar = (kappa_r1*kappa_r2)/(kappa_r1+kappa_r2)
    delta = K_A*x_A - K_B*x_B
    sigma = (kappa_r1*K_A*x_A + kappa_r2*K_B*x_B)/(kappa_r1 + kappa_r2)
    K_m = K_C/K_E

    v = kappa_bar*e0*K_C*delta/(K_m+sigma)
    x_E = e0/(1+(sigma/K_m))
    x_C = e0 - x_E
    x_E0 = (K_E/K_E0)*x_E
    x = np.array([x_A,x_B,x_C,x_E,x_E0])
    #x = [x_A,x_B,x_C,x_E,x_E0]
    return x,v

x_A = 2
x_B = 1
K_E0 = 1e-4
K_C = 1
x_ss,v_ss = ssECR(x_A,x_B,K_C=K_C,K_E0=K_E0)
print('x_ss =',x_ss)
print('v_ss =',v_ss)

x_ss = [2.e+00 1.e+00 6.e-01 4.e-01 4.e+03]
v_ss = 0.2
```

2.3 Parameters

```
In [5]: ## Parameters
        x_E0 = x_ss[4]

        parameter = {}
        parameter['K_E0'] = K_E0
        parameter['K_C'] = K_C
```

2.4 Linearisation

The function lin provides the linear transfer function sys in control toolbox format and puts the corresponding state-space matrices a , b , c and d into the data structure sc where $\dot{x} = ax + bu$

and $y = cx + du$ The system is the *reduced* form (Gawthrop and Crampin, 2016). The output y corresponds to the flows V , the input u to the chemostats X_{chemo} and x to the reduced state. In this case

$$y = V = \begin{pmatrix} V_{r0} \\ V_{r1} \\ V_{r2} \end{pmatrix} \quad (1)$$

$$u = X_{chemo} = \begin{pmatrix} X_A \\ X_B \\ X_{E0} \end{pmatrix} \quad (2)$$

$$x = \begin{pmatrix} X_C \\ X_E \end{pmatrix} \quad (3)$$

It also gives the symbolic matrix dv/dx relating incremental flows to incremental states where the states X are:

$$X = \begin{pmatrix} X_A \\ X_B \\ X_C \\ X_E \\ X_{E0} \end{pmatrix} \quad (4)$$

```
In [6]: ## imp.reload(st)
        sys = st.lin(s,sc,x_ss=x_ss,parameter=parameter,quiet=quiet)
```

```
In [7]: disp.Latex(st.sprintl(sc,'dvdxd'))
```

Out[7]:

$$dvdxd = \begin{pmatrix} 0 & 0 & 0 & -K_E\kappa_{r0} & K_{E0}\kappa_{r0} \\ K_A K_E \kappa_{r1} x_E & 0 & -K_C \kappa_{r1} & K_A K_E \kappa_{r1} x_A & 0 \\ 0 & -K_B K_E \kappa_{r2} x_E & K_C \kappa_{r2} & -K_B K_E \kappa_{r2} x_B & 0 \end{pmatrix} \quad (5)$$

```
In [8]: disp.Latex(st.sprintl(sc,'a'))
```

Out[8]:

$$a = \begin{pmatrix} -2.0 & 3.0 \\ 2.0 & -4.0 \end{pmatrix} \quad (6)$$

```
In [9]: disp.Latex(st.sprintl(sc,'b'))
```

Out[9]:

$$b = \begin{pmatrix} 0.4 & 0.4 & 0.0 \\ -0.4 & -0.4 & 0.0001 \end{pmatrix} \quad (7)$$

```
In [10]: disp.Latex(st.sprintl(sc, 'c'))
```

```
Out[10]:
```

$$c = \begin{pmatrix} 0.0 & -1.0 \\ -1.0 & 2.0 \\ 1.0 & -1.0 \end{pmatrix} \quad (8)$$

```
In [11]: disp.Latex(st.sprintl(sc, 'd'))
```

```
Out[11]:
```

$$d = \begin{pmatrix} 0.0 & 0.0 & 0.0001 \\ 0.4 & 0.0 & 0.0 \\ 0.0 & -0.4 & 0.0 \end{pmatrix} \quad (9)$$

2.5 Pole/zero analysis

One advantage of dealing with linear systems is the possibility of using standard control system methods (Goodwin et al., 2001). One such method is examining the poles and zeros of the system transfer function relating inputs and outputs of interest. In this case, the transfer function relating input X_{E0} to output V_{r2} is examined.

NB con.zero requires slycot to be installed; set slycot=False if slycot is not installed.

Note that the positive zero ($s = 1$) corresponds to the initial negative response of the flow - this is a non-minimum phase system

```
In [12]: i_E0 = chemostats.index('E0')
         reaction = sc['reaction']
         i_r = reaction.index('r2')
         np.set_printoptions(precision=2)
         aa = sc['a']
         bb = np.array([sc['b'][:,i_E0]]).T
         cc = sc['c'][i_r,:]
         dd = sc['d'][i_r,i_E0]
         siso = con.ss(aa,bb,cc,dd)
         np.set_printoptions(precision=2)
         gain = x_E0*con.dcgain(siso)
         print('System gain = {0:.2f}'.format(gain))
         print('System poles = ',con.pole(siso))
         if slycot:
             ## This needs slycot
             print('System zeros = ',np.real(con.zero(siso)))
```

```
System gain = 0.20
```

```
System poles = [-0.35 -5.65]
```

ModuleNotFoundError

Traceback (most recent call last)

ModuleNotFoundError: No module named 'numpy.core._multiarray_umath'

System zeros = [1.]

2.6 Simulation

1. As simulation starts at a steady-state value, states and flows remain constant until the chemostat E0 changes at time $t_0 = 5$.
2. For each fixed chemostat value, the non-linear system is actually linear (but with parameters dependent on the steady-state).
3. The flow in reaction r2 is the same for both linear and non-linear simulation.
4. The flow in reaction r2 increases by the gain=0.2.

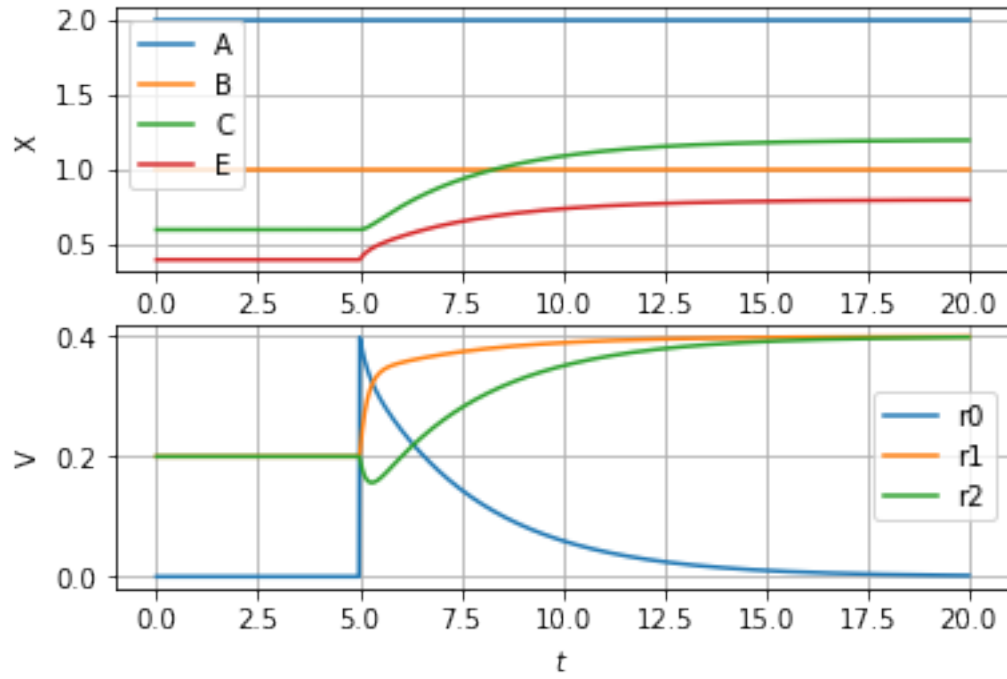
```
In [13]: ## imp.reload(st)
         ## Time
         t_max = int(20)
         t = np.linspace(0,t_max,1000)
         t_0 = 5

         ## Chemostat
         x_chemo = '{0}*(1+np.heaviside(t-{1},1))'.format(str(x_E0),str(t_0))
         x_chemo_lin = '{0}*(np.heaviside(t-{1},1))'.format(str(x_E0),str(t_0))

         ## Simulate
         print('NON-LINEAR SIMULATION')
         #X0 = x_ss
         X_chemo = {'EO':x_chemo}
         ndat = st.sim(s,sc=sc,t=t,parameter=parameter,X0=x_ss,X_chemo=X_chemo,quiet=False)
         st.plot(s,ndat,species=['A','B','C','E'])

         print('LINEAR SIMULATION')
         #X0 = x_ss
         X_chemo = {'EO':x_chemo_lin}
         V0 = [0,v_ss,v_ss] # Steady-state flows
         ldat = st.sim(s,sc=sc,t=t,linear=True,V0=V0,parameter=parameter,X0=x_ss,X_chemo=X_chemo)
         st.plot(s,lдат,species=['A','B','C','E'])
         #st.plot(s,lдат,species=[])

NON-LINEAR SIMULATION
Setting K_C to 1
Setting K_E0 to 0.0001
```



LINEAR SIMULATION

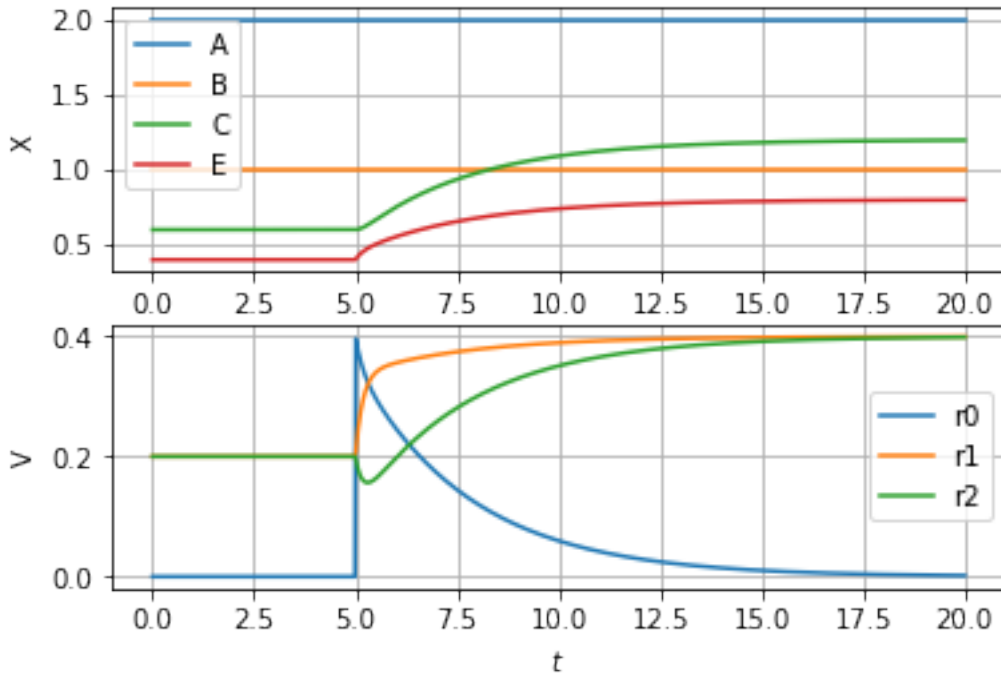
Setting K_C to 1

Setting K_{E0} to 0.0001

Setting K_C to 1

Setting K_{E0} to 0.0001

/home/peterg/WORK/Research/SystemsBiology/lib/python/stoich.py:109: RuntimeWarning: divide by zero
 lx_i = np.log(xx_i)



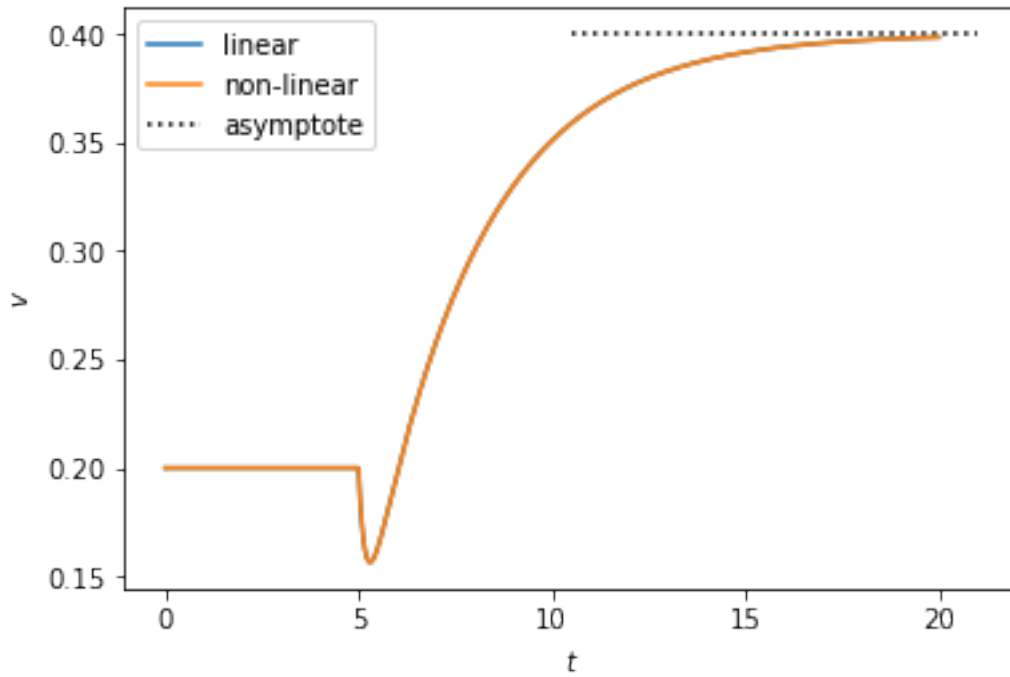
2.6.1 Compare linear and nonlinear

```
In [14]: def compare(ndat,ldat,v_ss,gain,i):

    v_1_l = ldat['V'][:,i]
    v_1_n = ndat['V'][:,i]

    ax = plt.gca() # gca stands for 'get current axis'
    plt.plot(t,v_1_l,label='linear')
    plt.plot(t,v_1_n,label='non-linear')
    xlim = ax.get_xlim()
    plt.hlines(v_ss+gain, xlim[1]/2, xlim[1],linestyles='dotted',label='asymptote')
    plt.xlabel('$t$')
    plt.ylabel('$v$')
    plt.legend()
    plt.grid
    plt.show()

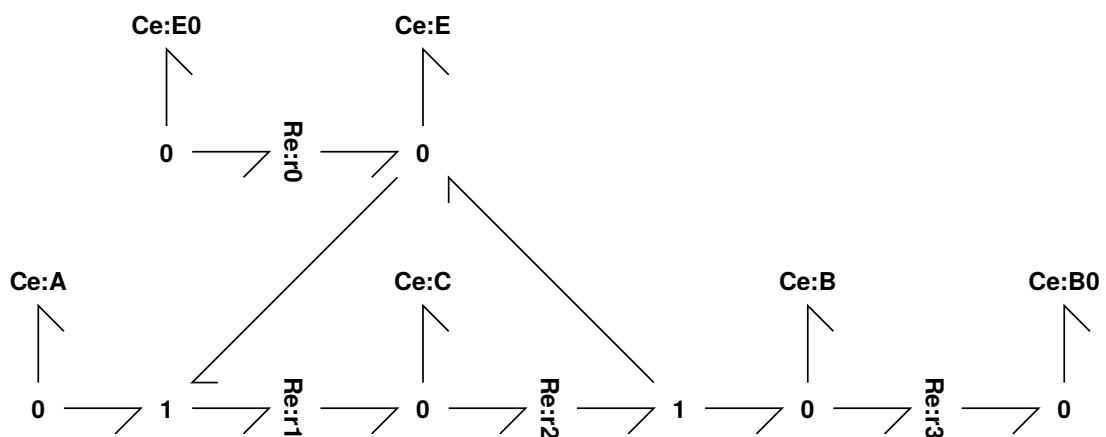
    compare(ndat,ldat,v_ss,gain,i_r)
```



3 Example system: enzyme-catalysed reaction with product removal

```
In [15]: ## imp.reload(st)
          sbg.model('eREr_abg.svg')
          import eREr_abg
          disp.SVG('eREr_abg.svg')
```

Out[15]:



3.1 Stoichiometry

```
In [16]: s = st.stoich(eREr_abg.model(),quiet=quiet)
         print(s['species'])
         chemostats=['A','B0','E0']
         disp.Latex(st.sprintrl(s))
         sc = st.statify(s,chemostats=chemostats)

['A', 'B', 'B0', 'C', 'E', 'E0']
```

3.2 Parameters

```
In [17]: ## Parameters
         # x_E0 = x_ss[4]
         # x_A = 2
         # x_B = 1

         parameter = {}
         parameter['K_E0'] = K_E0
         parameter['K_C'] = K_C
```

3.3 Steady state.

As a special case, the amount of B0 is taken to be very small and the parameter $\kappa_3 = v_{ss}$. This gives the same steady states as the previous section.

```
In [18]: ## Steady states for ECR
         x_ss,v_ss = ssECR(x_A,x_B,K_C=K_C,K_E0=K_E0)

         ## Steady states for this examples
         X_ss = np.array([x_ss[0],x_ss[1],1e-10,x_ss[2],x_ss[3],x_ss[4]])
         #print(X_ss)
         V_ss = np.array([0,v_ss,v_ss,v_ss])
         #print(V_ss)

         ## Set appropriate kappa3
         kappa_r3 = v_ss
         parameter['kappa_r3'] = kappa_r3
```

3.4 Linearisation

The function `lin` provides the linear transfer function `sys` in control toolbox format and puts the corresponding state-space matrices a , b , c and d into the data structure `sc` where $\dot{x} = ax + bu$ and $y = cx + du$. The system is the *reduced* form (Gawthrop and Crampin, 2016). The output y corresponds to the flows V , the input u to the chemostats X_{chemo} and x to the reduced state. In this case

$$y = V = \begin{pmatrix} V_{r0} \\ V_{r1} \\ V_{r2} \\ V_{r3} \end{pmatrix} \quad (10)$$

$$U = X_{chemo} = \begin{pmatrix} X_A \\ X_{B0} \\ X_{E0} \end{pmatrix} \quad (11)$$

$$x = \begin{pmatrix} X_B \\ X_C \\ X_E \end{pmatrix} \quad (12)$$

It also gives the symbolic matrix dv/dx relating incremental flows to incremental states where the states X are:

$$X = \begin{pmatrix} X_A \\ X_B \\ X_{B0} \\ X_C \\ X_E \\ X_{E0} \end{pmatrix} \quad (13)$$

```
In [19]: ## imp.reload(st)
          sys = st.lin(s,sc,x_ss=X_ss,parameter=parameter)
          disp.Latex(st.sprintl(sc,'dvdv'))
```

```
Setting K_C to 1
Setting K_E0 to 0.0001
Setting kappa_r3 to 0.2
```

Out[19]:

$$dvdv = \begin{pmatrix} 0 & 0 & 0 & 0 & -K_E\kappa_{r0} & K_{E0}\kappa_{r0} \\ K_A K_E \kappa_{r1} x_E & 0 & 0 & -K_C \kappa_{r1} & K_A K_E \kappa_{r1} x_A & 0 \\ 0 & -K_B K_E \kappa_{r2} x_E & 0 & K_C \kappa_{r2} & -K_B K_E \kappa_{r2} x_B & 0 \\ 0 & K_B \kappa_{r3} & -K_{B0} \kappa_{r3} & 0 & 0 & 0 \end{pmatrix} \quad (14)$$

```
In [20]: disp.Latex(st.sprintl(sc,'a'))
```

Out[20]:

$$a = \begin{pmatrix} -0.6 & 1.0 & -1.0 \\ 0.4 & -2.0 & 3.0 \\ -0.4 & 2.0 & -4.0 \end{pmatrix} \quad (15)$$

```
In [21]: disp.Latex(st.sprintl(sc,'b'))
```

Out [21]:

$$b = \begin{pmatrix} 0.0 & 0.2 & 0.0 \\ 0.4 & 0.0 & 0.0 \\ -0.4 & 0.0 & 0.0001 \end{pmatrix} \quad (16)$$

```
In [22]: disp.Latex(st.sprintl(sc, 'c'))
```

Out [22]:

$$c = \begin{pmatrix} 0.0 & 0.0 & -1.0 \\ 0.0 & -1.0 & 2.0 \\ -0.4 & 1.0 & -1.0 \\ 0.2 & 0.0 & 0.0 \end{pmatrix} \quad (17)$$

```
In [23]: disp.Latex(st.sprintl(sc, 'd'))
```

Out [23]:

$$d = \begin{pmatrix} 0.0 & 0.0 & 0.0001 \\ 0.4 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & -0.2 & 0.0 \end{pmatrix} \quad (18)$$

3.5 Pole/zero analysis

One advantage of dealing with linear systems is the possibility of using standard control system methods (Goodwin et al., 2001). One such method is examining the poles and zeros of the system transfer function relating inputs and outputs of interest. In this case, the transfer function relating input X_{E0} to output V_{r2} is examined.

NB con.zero requires slycot to be installed; set slycot=False if slycot is not installed.

Note that the positive zero ($s = 1$) corresponds to the initial negative response of the flow - this is a non-minimum phase system.

```
In [24]: i_E0 = chemostats.index('E0')
         reaction = sc['reaction']
         i_r = reaction.index('r3')
         np.set_printoptions(precision=2)
         aa = sc['a']
         bb = np.array([sc['b'][:, i_E0]]).T
         cc = sc['c'][i_r, :]
         dd = sc['d'][i_r, i_E0]
         siso = con.ss(aa, bb, cc, dd)
         np.set_printoptions(precision=2)
         gain = x_E0 * con.dcgain(siso)
         print('System gain = {0:.2f}'.format(gain))
         print('System poles = ', con.pole(siso))
         if slycot:
             ## This needs slycot
             print('System zeros = ', np.real(con.zero(siso)))
```

```
System gain = 0.10
System poles = [-5.8 -0.56 -0.25]
```

```
-----

ModuleNotFoundError                                Traceback (most recent call last)

ModuleNotFoundError: No module named 'numpy.core._multiarray_umath'
```

```
System zeros = [1.]
```

3.6 Simulation

1. As simulation starts at a steady-state value, states and flows remain constant until the chemostat E0 changes at time $t_0 = 5$.
2. For each fixed chemostat value, the non-linear system leads to an approximate linear system (but with parameters dependent on the steady-state).
3. The flow in reaction r2 is the different for linear and non-linear simulation, but the difference is smaller for smaller step changes in x_{E0} .
4. In the linear case, the flow in reaction r2 increases by gain=0.1 multiplied by the step.

```
In [25]: ## imp.reload(st)
```

```
##Time
t_max = int(40)
t = np.linspace(0,t_max,1000)
t_0 = 5

for step in [1,0.1]:

    ## Chemostat
    #step = 1
    x_chemo = '{0}*(1+{2}*np.heaviside(t-{1},1))'.format(str(x_E0),str(t_0),str(step))
    x_chemo_lin = '{0}*({2}*np.heaviside(t-{1},1))'.format(str(x_E0),str(t_0),str(step))

    ## Simulate
    print('NON-LINEAR SIMULATION: Step = '+str(step))
    #X0 = x_ss
    X_chemo = {'E0':x_chemo}
    ndat = st.sim(s,sc=sc,t=t,parameter=parameter,X0=X_ss,X_chemo=X_chemo,quiet=quiet)
    st.plot(s,ndat,species=['A','B','C','E'])

    print('LINEAR SIMULATION: Step = '+str(step))
    #X0 = x_ss
    X_chemo = {'E0':x_chemo_lin}
```

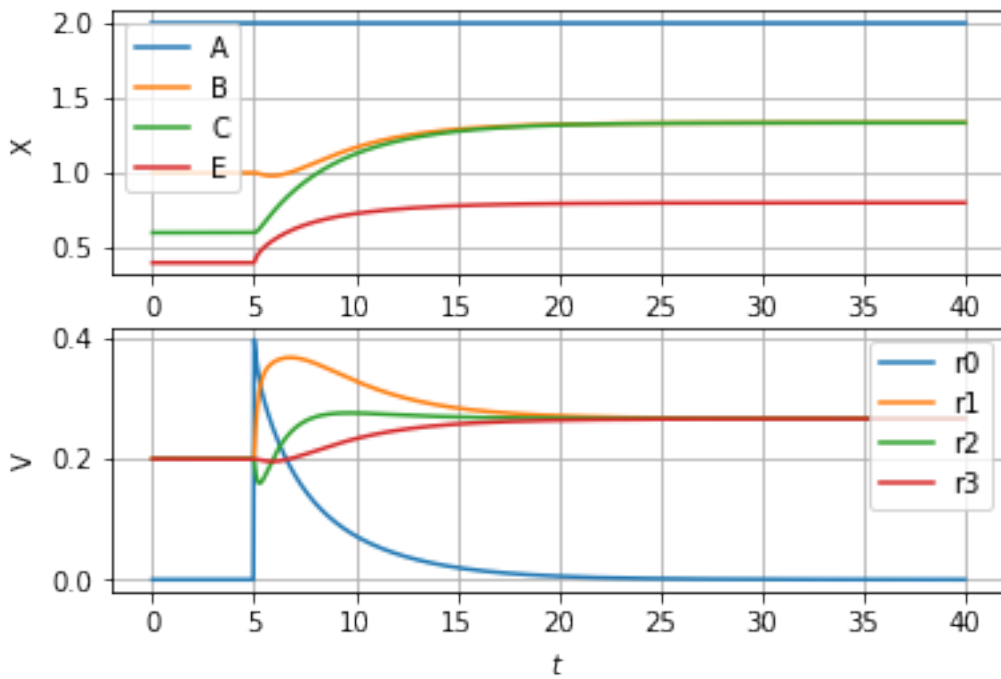
```

V0 = [0,v_ss,v_ss] # Steady-state flows
ldat = st.sim(s,sc=sc,t=t,linear=True,V0=V_ss,parameter=parameter,X0=X_ss,X_chemo=X
st.plot(s,ldat,species=['A','B','C','E'])

print('COMPARE LINEAR & NONLINEAR: Step = '+str(step))
compare(ndat,ldat,v_ss,step*gain,i_r)

```

NON-LINEAR SIMULATION: Step = 1

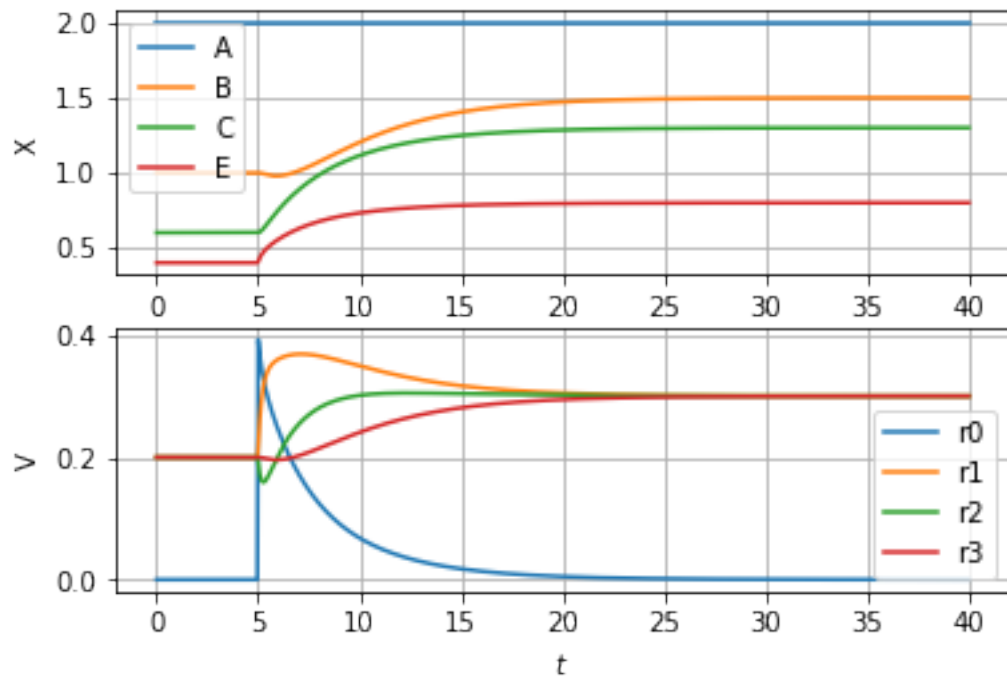


LINEAR SIMULATION: Step = 1

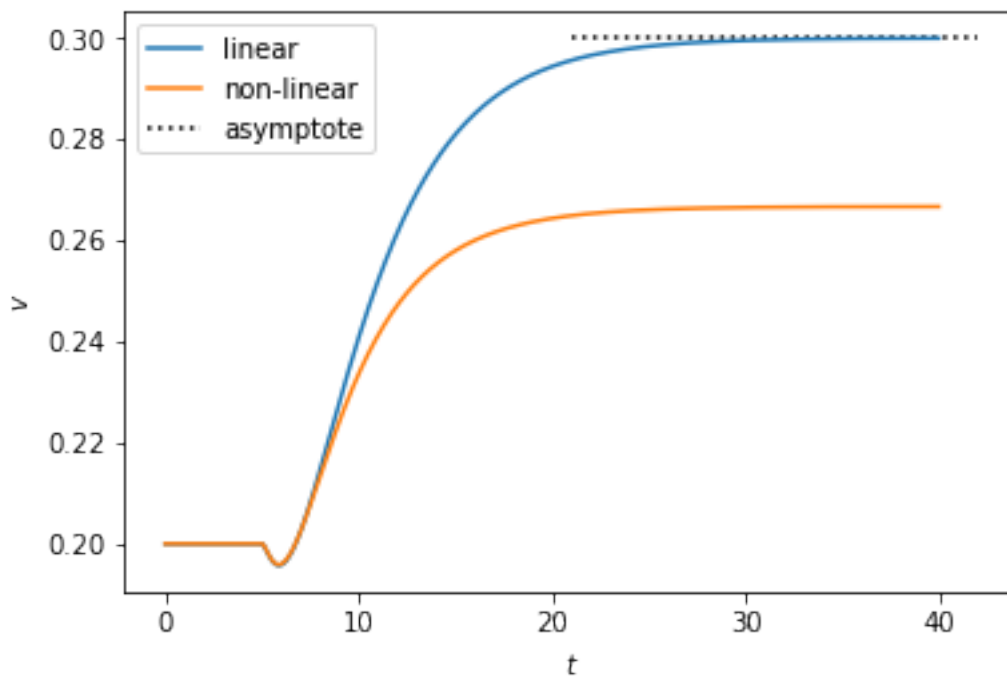
```

/home/peterg/WORK/Research/SystemsBiology/lib/python/stoich.py:109: RuntimeWarning: divide by ze
    lx_i = np.log(xx_i)

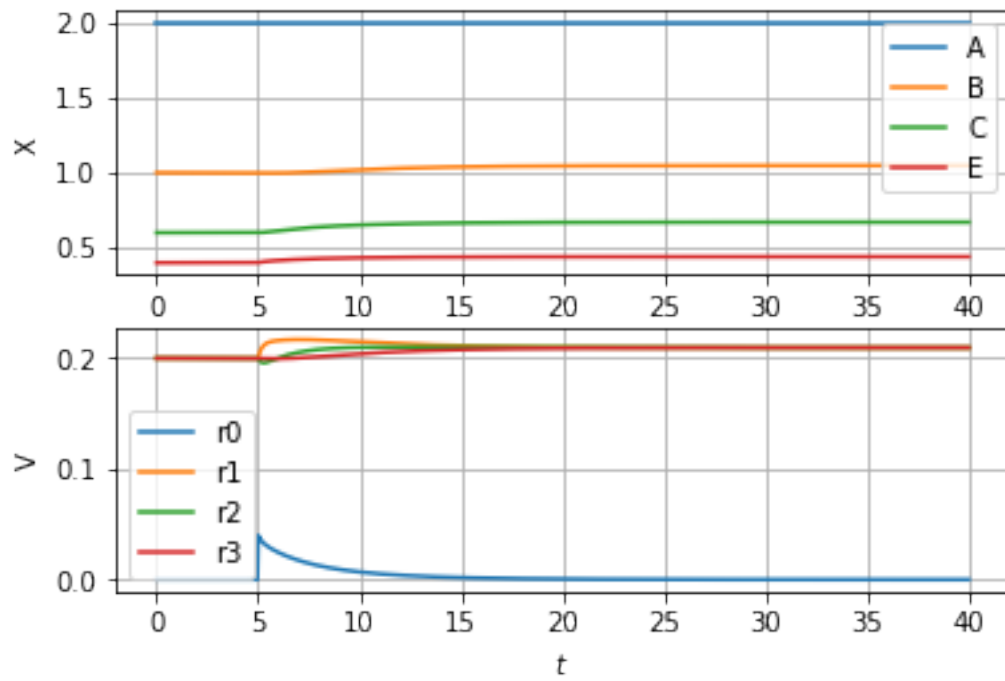
```



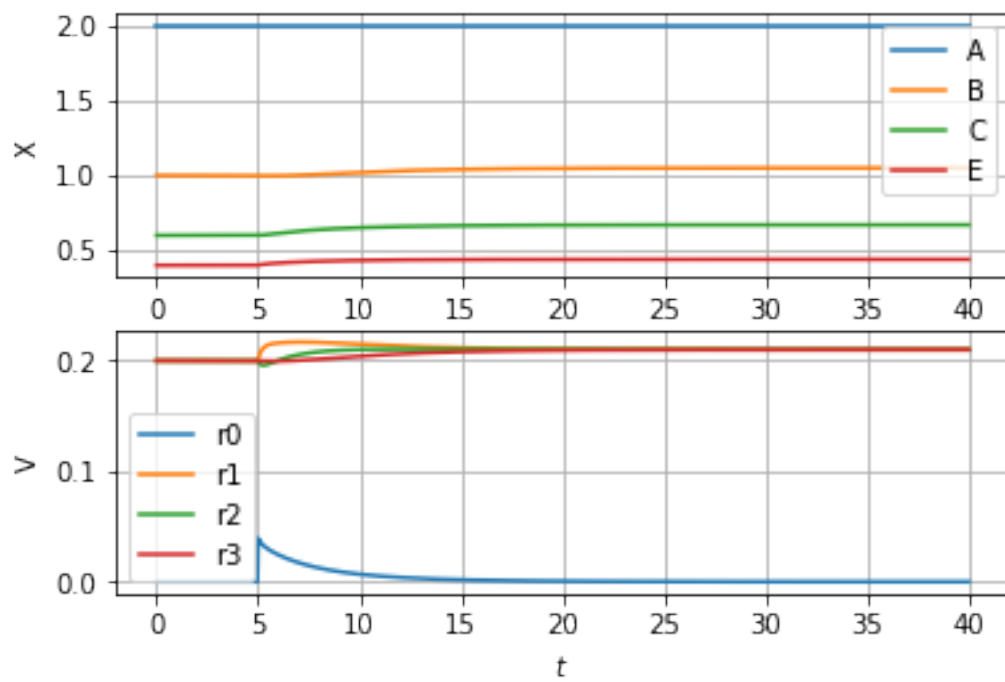
COMPARE LINEAR & NONLINEAR: Step = 1



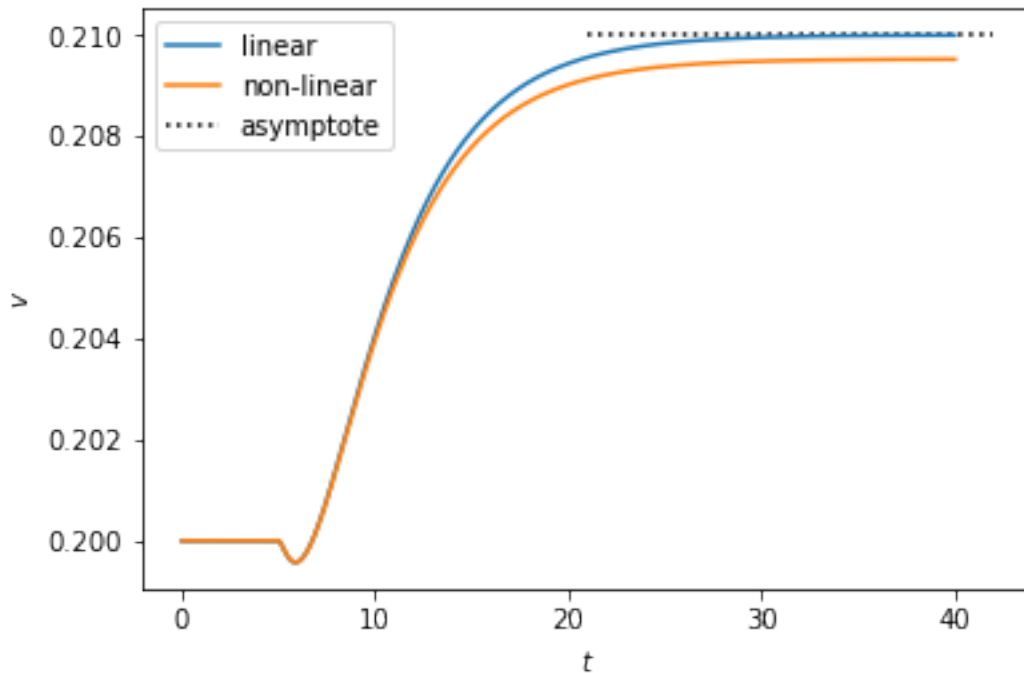
NON-LINEAR SIMULATION: Step = 0.1



LINEAR SIMULATION: Step = 0.1



COMPARE LINEAR & NONLINEAR: Step = 0.1



References

- P. Gawthrop. Computing biomolecular system steady-states. *IEEE Transactions on NanoBioscience*, 17(1):36–43, March 2018. ISSN 1536-1241. doi:[10.1109/TNB.2017.2787486](https://doi.org/10.1109/TNB.2017.2787486). Published online 25th December 2017.
- P. J. Gawthrop and E. J. Crampin. Modular bond-graph modelling and analysis of biomolecular systems. *IET Systems Biology*, 10(5):187–201, October 2016. ISSN 1751-8849. doi:[10.1049/iet-syb.2015.0083](https://doi.org/10.1049/iet-syb.2015.0083). Available at arXiv:1511.06482.
- Peter J. Gawthrop and Edmund J. Crampin. Energy-based analysis of biochemical cycles using bond graphs. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science*, 470(2171):1–25, 2014. doi:[10.1098/rspa.2014.0459](https://doi.org/10.1098/rspa.2014.0459). Available at arXiv:1406.2447.
- G.C. Goodwin, S.F. Graebe, and M.E. Salgado. *Control System Design*. Prentice Hall, Englewood Cliffs, New Jersey, 2001.