Cubic Equations of State and How to Use Them

Introduction

An equation of state relates thermodynamic quantities of a substance. Although the most preferred relation would be one between temperature, pressure and a free energy (e.g. Gibbs or Helmholtz), the most common equations of state relate temperature and pressure with volume. The most widely used among these are the Two Parameter Cubic Equations of State which have the following form:

$$P = \frac{RT}{v-b} - \frac{a\alpha(T)}{(v+\epsilon b)(v+\sigma b)}$$

Here a and b are given by:

$$a = \frac{\Omega_a R^2 T_c^2}{P_c}$$

and

$$b = \frac{\Omega_b R T_c}{P_c}$$

where T_c and P_c are the critical temperature and pressure respectively. $\alpha(T)$ is itself a function of temperature and given by:

$$\alpha(T) = \left(1 + \kappa\left(1 - \sqrt{\frac{T}{T_c}}\right)\right)^2$$

. Where κ is a function of the fluid's acentric factor ω.

Importing Our First Container Objects

We have already stored values of P_c, T_c and ω for Benzene in the file 'benzene.py': which is a $container-object$. We can access those values simply be importing benzene as follows:

```
In [1]: import  benzene
```

Now the critical pressure (in Pa), temperature (in K) and acentric factor (unitless) for benzene are:

```
In [2]: benzene.Pc, benzene.Tc, benzene.acc

Out[2]:  (4895000.0, 562.05, 0.212)
```

The Equation of State specific constants Ω_a, Ω_b, ϵ and σ as well as formula for calculation κ and α are stored in the container object called 'srk.py' for the Soave-Redlich-Kwong equation of state. We can access them by importing srk.py as follows:

```
In [3]: import srk
```

Hence, Ω_a, Ω_b, ϵ and σ are:

```
In [4]: srk.OmegaA, srk.OmegaB, srk.epsilon, srk.sigma

Out[4]:  (0.42748, 0.08644, 0.0, 1.0)
```

Our first Class and an introduction to Abstraction

The next step is to combine the information in srk.py and benzene.py to be able to plot pressure as a function of molar volume for a given temperature. The resulting curve is called an $isotherm$. To do the combining, we have defined a called EOS and placed it in the container called eosClass.py. We will import the container now.

```
In [5]: import eosClass
```

Inside the container 'eosClass' is the definition of the object-template called 'EOS'. It is a template because several objects can be instanciated using it. Lets instantiate an object that takes srk and applies it to benzene. We will call this object Benzene.

```
In [6]: Benzene = eosClass.EOS(srk, benzene)
```

If you go over to the Spyder editor and take a look at eosClass.EOS, you will find that the arguments passed in the line above are those for the function called `__init__`. This is called the $constructor$ of the class and it is executed entirely when an object of that type is first intanciated.

The object oriented approach has many advantages. We can see this immediately when we realize that the parameters a and b have already been calculated by the EOS class and are accessible from it. Hence,

```
In [7]: Benzene.a, Benzene.b

Out[7]: (1906922.5270022994, 0.08251768478610827)
```

Now, lets recalculate a and b "manually",

```
In [8]: srk.OmegaA*8314.0**2*benzene.Tc**2/benzene.Pc, srk.OmegaB*8314.0*benzene.Tc/benzene.Pc

Out[8]: (1906922.5270022994, 0.08251768478610827)
```

The Scipy Library

They are clearly exactly the same, except we got them in a much cleaner, less messy manner. This is called $abstraction$. We abstracted the calculation of a and b inside the code, hiding it from the user (us) so that we can build on this to do more complex things. Lets now just use this powerful ability to calculate and plot P as a function of v for a given value of T.

```
In [9]: T = 500.0
        import scipy
```

Scientific Python (scipy) is a large collection of libraries meant for scientific calculations. Among the various useful tools in scipy is a tool that quickly returns a grid. Lets make a grid for molar volume. Since pressure has a singularity at $v = b$, we will choose a safe grid between $1.1b$ and $100b$. We do it as follows:

```
In [10]: vv = scipy.linspace(1.1*Benzene.b, 100*Benzene.b, 100000)
```

The arguments for the scipy.linspace function are: initial point of grid, final point of the grid and number of grid points (inclusive on initial and final points). So vv will be 100000 elements long as we can check for ourselves:

```
In [11]: len(vv)

Out[11]: 100000
```

Now, in the class EOS, we have abstracted a $method$ or to return pressure given temperature and volume. We use this method to obtain an array of pressures corresponding to T and vv. Hence:

```
In [12]: pp = Benzene.P(T, vv)
```

```
In [13]: len(pp)

Out[13]: 100000
```

Plotting using pylab

Lets now plot pressure vs volume for benzene as for a given temperature. For this we will need to import some plotting libraries. Lets do so now:

```
In [14]: import pylab
```

Now pass the x-array, the y-array and the type of the plot. Then we give an instruction to show the plot.

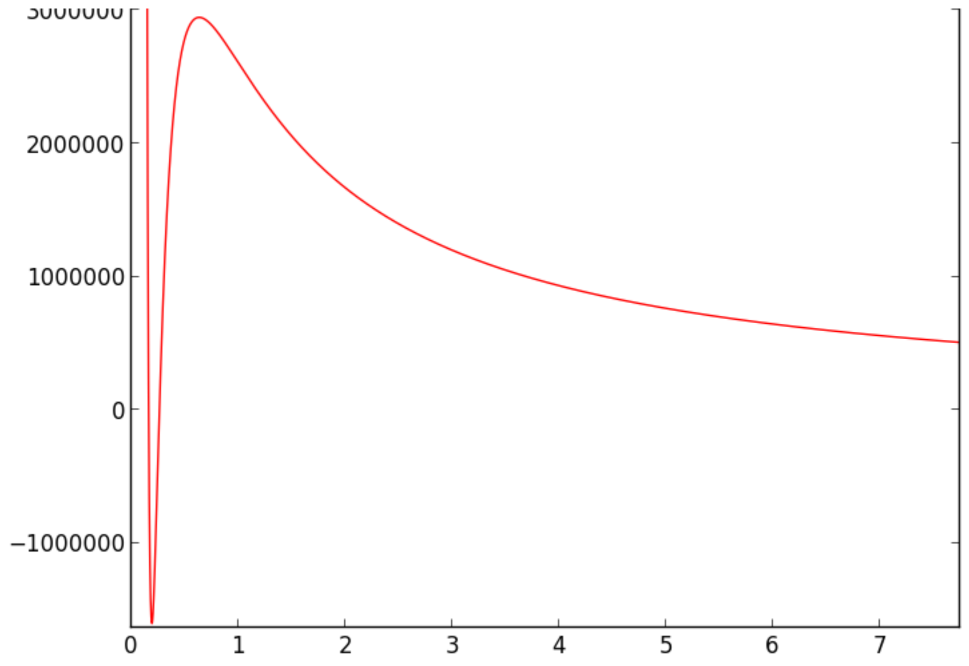
```
In [15]: pylab.plot(vv, pp, 'r'); pylab.show()
```

It opens in another window (as indicated by the * where there should be a number in the code numbering). We can interact with the graph in various ways including zooming in to make it more insightful. Then save it as an image. We can now get that image here as follows:

IPython Display

```
In [16]: from IPython.display import Image
         img = Image(filename='eos_T500_Benzene_SRK.png')
         img

Out[16]:
```



This is all well and good, but in process simulation, we are given T and P and almost never are we given T and v. That presents a problem since the cubic equation of state is a cubic in v. Hence it needs to be solved for the roots. Now, solving the equation as it is written is very difficult and fraught with numerical round-off errors. A further problem is that the vapor and liquid volumes are very different, again raising issues of numerical roundoff errors. Still another problem is the singularity encountered at $v = b$ which makes solving more complex.
All of these problems can be neatly avoided once we recognize that the problem is actually a polynomial problem. But before we transform it, lets get some help.

Sympy!! And all your algebra problems are gone!

In [17]:
```
from sympy import init_session
init_session()
```

```
Welcome to pylab, a matplotlib-based Python environment [backend: Qt4Agg].
For more information, type 'help(pylab)'.
IPython console for SymPy 0.7.2 (Python 2.7.3-32-bit) (ground types: python)

These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)

Documentation can be found at http://www.sympy.org
C:\Python27\lib\site-packages\matplotlib\__init__.py:921: UserWarning:  This call to matplotlib.use() has no effect
because the the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

  if warn: warnings.warn(_use_error_msg)
```

Transforming the Cubic Equation of State to a Monic Polynomial

Great, lets define some symbols:

In [18]:
```
P, v, R, T, Z, a, b, alp, s, e = symbols('P, v, R, T, Z, a, b, alpha, sigma, epsilon')
```

Lets take a look at them: very pretty!

In [19]:
```
P, v, R, T, Z, a, b, alp, s, e
```

Out[19]: $(P, \quad v, \quad R, \quad T, \quad Z, \quad a, \quad b, \quad \alpha, \quad \sigma, \quad \epsilon)$

Alright, the RHS of the cubic equation of state is:

In [20]:
```
RHS = R*T/(v-b) - a*alp/(v+e*b)/(v+s*b); RHS
```

Out[20]:
$$\frac{RT}{-b+v} - \frac{a\alpha}{(b\epsilon+v)(b\sigma+v)}$$

Lets get it together:

In [21]:
```
RHS = RHS.together(); RHS
```

Out[21]:
$$\frac{RT(b\epsilon+v)(b\sigma+v) - a\alpha(-b+v)}{(-b+v)(b\epsilon+v)(b\sigma+v)}$$

Lets now get the numerator and denominator separately using a neat trick.

In [22]:
```
RHS = fraction(RHS); nr = RHS[0]; dr = RHS[1]; nr, dr
```

Out[22]:
$$\left(RT(b\epsilon+v)(b\sigma+v) - a\alpha(-b+v),\quad (-b+v)(b\epsilon+v)(b\sigma+v)\right)$$

That is just wicked cool! Now the denominator multiplies the LHS which is simply P and the numerator get shifted to the LHS with a flipped sign to give two terms.

In [23]:
```
term1 = P*dr; term2 = -nr; term1, term2
```

Out[23]:
$$\left(P(-b+v)(b\epsilon+v)(b\sigma+v),\quad -RT(b\epsilon+v)(b\sigma+v) + a\alpha(-b+v)\right)$$

And now it only remains to expand both of them to get the cubic (sort of):

In [24]:
```
expr = term1.expand() + term2.expand(); expr
```

Out[24]:
$$-Pb^3\epsilon\sigma + Pb^2\epsilon\sigma v - Pb^2\epsilon v - Pb^2\sigma v + Pb\epsilon v^2 + Pb\sigma v^2 - Pbv^2 + Pv^3 - RTb^2\epsilon\sigma - RTb\epsilon v - RTb\sigma v - RTv^2 - a\alpha b + a\alpha v$$

But we still haven't overcome the problem of large differences in v of gas and liquid. Lets do that by defining $Z = \frac{Pv}{RT}$. And now we have another little trick:

In [25]:
```
expr = expr.subs(v, Z*R*T/P); expr
```

Out[25]:
$$-Pb^3\epsilon\sigma + RTZb^2\epsilon\sigma - RTZb^2\epsilon - RTZb^2\sigma - RTb^2\epsilon\sigma - a\alpha b + \frac{R^2T^2Z^2b\epsilon}{P} + \frac{R^2T^2Z^2b\sigma}{P} - \frac{R^2T^2Z^2b}{P} - \frac{R^2T^2Zb\epsilon}{P} - \frac{R^2T^2Zb\sigma}{P}$$

That is just mad! If only we could collect like terms ... Hang on! We can ...

In [26]:
```
collect(expr, Z)
```

Out[26]:
$$-Pb^3\epsilon\sigma - RTb^2\epsilon\sigma + Z^2\left(\frac{R^2T^2b\epsilon}{P} + \frac{R^2T^2b\sigma}{P} - \frac{R^2T^2b}{P} - \frac{R^3T^3}{P^2}\right) + Z\left(RTb^2\epsilon\sigma - RTb^2\epsilon - RTb^2\sigma - \frac{R^2T^2b\epsilon}{P} - \frac{R^2T^2b\sigma}{P} + \frac{R}{}\right.$$

Wow! But there is something multiplying Z^3. Thats no good. Lets turn this into a monic polynomial ...

In [27]:
```
expr = (expr*P**2/R**3/T**3).expand(); expr
```

Out[27]:
$$-\frac{P^3b^3\epsilon\sigma}{R^3T^3} + \frac{P^2Zb^2\epsilon\sigma}{R^2T^2} - \frac{P^2Zb^2\epsilon}{R^2T^2} - \frac{P^2Zb^2\sigma}{R^2T^2} - \frac{P^2b^2\epsilon\sigma}{R^2T^2} - \frac{P^2a\alpha b}{R^3T^3} + \frac{PZ^2b\epsilon}{RT} + \frac{PZ^2b\sigma}{RT} - \frac{PZ^2b}{RT} - \frac{PZb\epsilon}{RT} - \frac{PZb\sigma}{RT} + \frac{PZa\alpha}{R^2T^2} +$$

It appears that we can identify certain groups. One is $p_p = \frac{Pb}{RT}$ and the other is $a_p = \frac{Pa\alpha}{R^2T^2}$. Lets define them and substitute.

In [28]:
```
ap, pp = symbols('a_p, p_p'); exprpp = expr.subs(P*b/R/T, pp); exprap = exprpp.subs(P*a*alp/R**2/T**2, ap); exprap
```

Out[28]:
$$Z^3 + Z^2\epsilon p_p + Z^2p_p\sigma - Z^2p_p - Z^2 + Za_p + Z\epsilon p_p^2\sigma - Z\epsilon p_p^2 - Z\epsilon p_p - Zp_p^2\sigma - Zp_p\sigma - a_pp_p - \epsilon p_p^3\sigma - \epsilon p_p^2\sigma$$

Lets collect now:

In [29]:
```
expr = collect(exprap, Z); expr
```

Out[29]:
$$Z^3 + Z^2\left(\epsilon p_p + p_p\sigma - p_p - 1\right) + Z\left(a_p + \epsilon p_p^2\sigma - \epsilon p_p^2 - \epsilon p_p - p_p^2\sigma - p_p\sigma\right) - a_pp_p - \epsilon p_p^3\sigma - \epsilon p_p^2\sigma$$

But we can also get the coefficients separately:

In [30]:
```
a0 = expr.coeff(Z, 0); a1 = expr.coeff(Z, 1); a2 = expr.coeff(Z, 2); a3 = expr.coeff(Z, 3); a0, a1, a2, a3
```

Out[30]: $\left(-a_p p_p - \epsilon p_p^3 \sigma - \epsilon p_p^2 \sigma, \quad a_p + \epsilon p_p^2 \sigma - \epsilon p_p^2 - \epsilon p_p - p_p^2 \sigma - p_p \sigma, \quad \epsilon p_p + p_p \sigma - p_p - 1, \quad 1\right)$

Lets get the coefficients in a form we can use in our code:

In [31]:
```
print a0
```

```
-a_p*p_p - epsilon*p_p**3*sigma - epsilon*p_p**2*sigma
```

In [32]:
```
print a1
```

```
a_p + epsilon*p_p**2*sigma - epsilon*p_p**2 - epsilon*p_p - p_p**2*sigma - p_p*sigma
```

In [33]:
```
print a2
```

```
epsilon*p_p + p_p*sigma - p_p - 1
```

In [34]:
```
print a3
```

```
1
```

Putting Numbers:

We can straightaway use them in our code and we have. Just to check though, lets do some "manual" work. We pick the T=500K isotherm (plotted above) and pick $P = 10^6 Pa$ pressure which should give us three real roots. Note that everywhere we have used $R = 8314.0 \frac{J}{kmol-K}$.

In [35]:
```
T = 500.0; P = 1e6; alpha = srk.alpha(T/benzene.Tc, benzene.acc); a = srk.OmegaA*8314.0**2*benzene.Tc**2/benzene.Pc; b = srk.O
a_p = P*a*alpha/8314**2/T**2; p_p = P*b/8314.0/T; epsilon = srk.epsilon; sigma = srk.sigma
```

Hence, the coefficients are (just copy and paste from the lines above):

In [36]:
```
a_0 = -a_p*p_p - epsilon*p_p**3*sigma - epsilon*p_p**2*sigma
a_1 = a_p + epsilon*p_p**2*sigma - epsilon*p_p**2 - epsilon*p_p - p_p**2*sigma - p_p*sigma
a_2 = epsilon*p_p + p_p*sigma - p_p - 1
a_3 = 1.0
```

Getting Roots of A Polynomial: scipy.roots

Now we have everything to get the roots of the cubic. We now will use a special function from the scipy library that takes in the coefficients as a list (highest order first, constant last) and returns all the roots. A cubic will have 3 roots of which all may be real or just one may be real and the others complex conjugates of each other. Hence the roots are:

In [37]:
```
roots = scipy.roots([a_3, a_2, a_1, a_0]); roots
```

Out[37]:
```
[ 0.89019298  0.0728829   0.03692412]
```

Clearly we have three real roots. Now only the maximum and minimum of these have physical meaning. The middle roots occurs in the unphysical branch of the P vs v isotherm where $\frac{\partial P}{\partial v} > 0$ and is ignored. Hence,

In [38]:
```
ZL = min(roots); ZG = max(roots)
```

Lets see what they look like!

In [39]:
```
ZL, ZG
```

Out[39]: $\left(0.0369241213247, \quad 0.890192980998\right)$

But we have $abstracted$ this same code in the method attribute $Z(T, P)$ of the $ClassEOS$ in $eosClass.py$. We have already instanciated an object of that class as $Benzene$. So lets use it!

In [40]:
```
[ZL, ZG] = Benzene.Z(T, P); ZL, ZG
```

Out[40]: $\left(0.0369241213247, \quad 0.890192980998\right)$

Exactly the same, and with just a fraction of the effort! And without making any mistakes.

Modularity

But what if I want it for methane and using the Peng Robinson equation of state? No sweat!

In [41]:
```
import methane; import pr; Methane = eosClass.EOS(pr, methane)
Methane.Z(T, P)
```

Out[41]:
```
[ 0.99832131]
```

Of course, the temperature is much higher than the critical temperature of methane which is:

In [42]:
```
methane.Tc
```

Out[42]: 190.5

so we would only expect to get one real root. Which we did.

:) This is modularity. If you really want to appreciate this, try redoing everything with Excel. Heck, try Matlab! Go on, I dare ya' ... And y' ain't seen nothin' yet!

Object Oriented Plotting: matplotlib.pyplot

We have already seen how to use $pylab$. But pylab is just a lame-ass compromise solution meant to wean people off Matlab and their ilk. For really elegant code, we need OO (Object Oriented) 'ness. So we have the following:

In [43]:
```
import matplotlib.pyplot as plt
```

You did not know that we could import thing under different names, did you? Well you can. And it is a great convenience. Imagine having to go about writing $matplotlib.pyplot$ everywhere! Lets do something with this. First define a figure.

In [44]:
```
fig = plt.figure()
```

A blank Matplotlib window should open up. There is nothing in it. If you had used pylab, you would have got the axes automatically. But you aren't using pylab. So you have to specify what you want more explicitly. So lets do it.

In [45]:
```
ax1 = fig.add_subplot(2,1,1) #To be read as: Of a grid of 2 rows and 1 column, pick the 1 cell.
ax2 = fig.add_subplot(2,1,2) #To be read as: Of a grid of 2 rows and 1 column, pick the 2 cell.
```

See what has happened in the other window. Nothing? Nothing. That is because you have to explicitly re-draw the canvas.

In [46]:
```
fig.canvas.draw()
```

Now take a look. There, two axes! Lets plot something in the first one. That will be $ax1$. Lets plot an isotherm for benzene.

In [47]:
```
vv = scipy.linspace(Benzene.b*1.74, Benzene.b*100, 10000)
pp = Benzene.P(T, vv)
ax1.plot(vv, pp, color = '#ff00ff', linewidth = 1, linestyle = '-', label = 'T=%f'%(T))
```

Out[47]:
```
[Line2D(T=500.000000)]
```

Lets unpack this. In pylab you would go $pylab.plot$ but here you have to specify $where$ you want the plot to appear. You can also specify a whole range of attributes of the plot. The x and y data we don't need to talk about. The color we do. The color attribute is given by a bizzare string of alphanumerics preceded by a '#'. It is actually an elegant way of expressing RGB values of a colour. R=Red, G=Green, B=Blue are the $PrimaryColors$. In RGB nomenclature each colour can take a value from 0 to 255. These numbers in hexadecimal are 00 and ff respectively. Using hexadecimal notation is elegant because you can go from 0 to 255 (thats 256 levels) using only two digits. The hexadecimal system of integers is 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f. Hence '#ff00ff' means R=255, G=0, B=255 i.e. purple. You can go online and look up the RGB values of practically any colour you like. Just for fun, look up the RGB value of "Cosmic Latte". Ok, $linewidth$ is just the width of the line in pixels. It is 1 by default but we have made it 0.5 so that the plot looks sophisticated.
$linestyle$ allows us to choose between full ('-') or broken ('--') or dotted-dashed ('-.') lines. $label$ just allows us to tag the curve in case we want to add a legend later on. Now text in python allows formatted input. Hence the 'T=%f'%(T) merely means that in the string, the program leaves room for a 'float' value for which it searches in the tuple following the string. Hence:

In [48]:
```
a = 30.0; s = 'Batman'; i = 3
string = '%s is one of only %d superheroes who has no superpowers. \n His chances of surviving a train crash are %f percent.'%
string
```

Out[48]:
```
Batman is one of only 3 superheroes who has no superpowers.
 His chances of surviving a train crash are 30.000000 percent.
```

Ok. I bet we are dying to look at the curve. Hence,

```
In [49]: fig.canvas.draw()
```

You can also clear the figure. Hence:

```
In [50]: ax1.cla(); fig.canvas.draw()
```

And the curves are gone. Lets replot them to bring them back.

```
In [51]: ax1.plot(vv, pp, color = '#ff00ff', linewidth = 1, linestyle = '-', label = 'T=%f'%(T))
         fig.canvas.draw()
```

Lets give the axis some names as well: What about v for the abscissa and P for the ordinate?

```
In [52]: ax1.xaxis.label.set_text('$v$'); ax1.yaxis.label.set_text('$P$')
         fig.canvas.draw()
```

What? You don't see the P? Stretch out the window and there it is ...

Alright, what we now want to do is calculate another quantity and that is the residual Gibbs free energy given by the symbol g^R. The ideal gas contribution to the free energy of both the gas and the liquid is identical. What is different is the "real" part of the contribution i.e. the residual. As is often the case in thermodynamics, and for really good reasons that we will see later, a dimensionless parameter ϕ is defined which is related to the residual Gibbs free energy by:

$$g^R = RT\log\phi$$

Here $\log x$ is to be understood as the natural logarithm. Since the equation of state somehow describes the "real" behaviour of simple fluids, there has to be a relationship between it and g_R. And there is, of course. Lets look at a basic thermodynamic relationship

$$dg = -sdT + vdP$$

We don't really want to worry about temperature just now. So for an isothermal process,

$$dg_T = vdP$$

We can write a similar equation for the ideal gas

$$dg^{id}_T = v^{id}dP = \frac{RT}{P}dP$$

. Subtract the two and we have:

$$dg^R_T = \left(v - \frac{RT}{P}\right)dP$$

Now integrate from $P = 0$ to P. Hence,

$$g^R_{T,P} - g^R_{T,P=0} = \int_{P=0}^{P}\left(v - \frac{RT}{P}\right)dP$$

Now since, by definition, an ideal gas is one what does not interact with any other gas molecule, the real gas at $P = 0$ actually shows ideal behaviour since it is so sparse that any molecule is infinitely away from any other. Hence $g^R_{T,P=0} = 0$. Hence:

$$g^R_{T,P} = \int_{P=0}^{P}\left(v - \frac{RT}{P}\right)dP$$

Now we could, in theory, integrate the above equation straightaway since the equation of state gives us a relationship between P and v. But this relationship is explicit in P and the form of the equation above is quite awkward. But we can use the chain rule of differentiation to realise that:

$$d(Pv) = vdP + Pdv$$

Hence, dividing by Pv we have:

$$\frac{d(Pv)}{Pv} = \frac{dP}{P} + \frac{dv}{v}$$

Substituting both vdP and $\frac{dP}{P}$ in the integral and realizing that $v \to \infty$ and $Pv \to RT$ as $P \to 0$ we have:

$$g^R_{T,P} = \int_{Pv=RT}^{P}\left(1 - \frac{RT}{Pv}\right)d(Pv) - \int_{v\to\infty}^{v}\left(P - \frac{RT}{v}\right)dv$$

Substituting P from the cubic equation of state and introducing $Z = \frac{Pv}{RT}$ we get:

$$\frac{g^R_{T,P,Z}}{RT} = \log\phi(T,P,Z) = Z - 1 - \log\left(Z - \frac{Pb}{RT}\right) + \frac{a\alpha(T)}{bRT(\sigma-\epsilon)}\log\left(\frac{Z + \epsilon\frac{Pb}{RT}}{Z + \sigma\frac{Pb}{RT}}\right)$$

The above expression is abstracted in the $eosClass.EOS$ object as $lnPhi(T,P,Z)$. Now at any temperature and pressure, if there exist two physically tenable values of Z then there are two physically tenable values of ϕ as well. At a given temperature and at a pressure where the cubic equation of state has three real roots, the phase the fluid exists in is one with the lower g^R i.e. the lower value of ϕ.

Lets see what the value of $\log\phi_L$ and $\log\phi_G$ are for $T = 500K$ and $P = 2e6Pa$.

```
In [53]: T = 500.0 #K
         P = 2e6 #Pa
         [ZL, ZG] = Benzene.Z(T, P) #Our abstracted values
         pb = P*Benzene.b/(8314.0*T); A = Benzene.a*Benzene.alpha(T)/Benzene.b/8314.0/T/(srk.sigma - srk.epsilon)
         lnphiL = ZL - 1 - scipy.log(ZL - pb) + A*scipy.log((ZL+srk.epsilon*pb)/(ZL+srk.sigma*pb))
         lnphiG = ZG - 1 - scipy.log(ZG - pb) + A*scipy.log((ZG+srk.epsilon*pb)/(ZG+srk.sigma*pb))
         lnphiL, lnphiG
```

Out[53]: $(-0.165257908604, \quad -0.221529113139)$

And redoing it with the abstracted code:

In [54]:
```
lnphiL = Benzene.lnPhi(T, P, ZL); lnphiG = Benzene.lnPhi(T, P, ZG)
lnphiL, lnphiG
```

Out[54]: $(-0.165257908604, \quad -0.221529113139)$

Yay!

Lets now replot the isotherm so that v is the ordinate. We already have the arrays, so we do:

In [55]:
```
ax1.cla() #clear current axis
ax1.plot(pp, vv, color = '#ff0000', linewidth = 1.0, linestyle = '-', label = 'T=%f'%(T)) #Note we have flipped postions of pp
ax1.xaxis.label.set_text('$P$')
ax1.yaxis.label.set_text('$v$')
fig.canvas.draw()
```

If you have been wondering why we have the second set of axes, wonder know further because I am going to tell you. We are going to plot $\log \phi_L$ and $\log \phi_G$ on it. But first lets just plot a raw $\log \phi$ on it.

In [56]:
```
zz = pp*vv/(8314.0*T); lnphi = Benzene.lnPhi(T, pp, zz)
ax2.cla()
ax2.plot(pp, lnphi, color = '#00ff00', linewidth = 0.25, linestyle='-', label='raw $\log{\phi}$')
fig.canvas.draw()
```

```
C:\Python27\lib\site-packages\numpy\core\numeric.py:235: ComplexWarning: Casting complex values to real discards the
imaginary part
  return array(a, dtype, copy=False, order=order)
```

That is not very helpful. Lets plot $\log \phi_L$ and $\log \phi_G$ separately. When we encounter a region with has a single real root, we attribute it to both Vapour and Liquid! We will need a loop for this:

In [57]:
```
lnphiL, lnphiG = [], []  #We define empty lists.  We will fill these up soon.
for p in pp:  #This is how a loop works.  It iterates through all elements of a list.
    listZ = Benzene.Z(T, p)  #Remember to indent.
    if len(listZ) == 2:  #i.e. if there are three real roots.
        [ZL, ZG] = listZ
        lnpL = Benzene.lnPhi(T, p, ZL); lnpG = Benzene.lnPhi(T, p, ZG)
        lnphiL.append(lnpL); lnphiG.append(lnpG) #This is how you add to the end of a list.  The append method.
    else:
        ZL = listZ[0]; ZG = listZ[0] #We give the same value to ZL and ZG
        lnp = Benzene.lnPhi(T, p, ZL)
        lnphiL.append(lnp); lnphiG.append(lnp)
ax2.plot(pp, lnphiL, color='#990000', linewidth = 0.25, linestyle = '-', label = '$\log{\phi_L}$')
ax2.plot(pp, lnphiG, color='#000099', linewidth = 0.25, linestyle = '-', label = '$\log{\phi_G}$')

fig.canvas.draw() #Don't forget to draw!
```

And now we want to do something even more sophisticated. On both graphs we want to highlight the curve which is the actual physical curve i.e. liquid where $\phi_L < \phi_G$ and vapour elsewhere.

In [58]:
```
lnphi = [min([lnphiL[i],lnphiG[i]]) for i in range(len(pp))] #We have used list comprehensions.  We can read it as "Make a lis
ax2.plot(pp, lnphi, color = 'cyan', linewidth = 3.0, linestyle = '-', label = '$\log{\phi}$', alpha = 0.4) #alpha sets the tra
fig.canvas.draw()
```

And it is not so easy for the pressure. We have to set up a whole new set of lists.

In [59]:
```
Pp = scipy.linspace(6e5, max(pp), 10000); vp = []  #We need to start from a non-zero value of P else otherwise the vapour volu
for p in Pp:
    listZ = Benzene.Z(T, p)
    if len(listZ) == 2:
        [ZL, ZG] = listZ
        lnphiL, lnphiG = Benzene.lnPhi(T, p, ZL), Benzene.lnPhi(T, p, ZG)
        Z = ZL if lnphiL < lnphiG else ZG
    else:
        Z = listZ[0]
    v = Z*8314*T/p; vp.append(v)
ax1.cla() #Clear the axes
ax1.plot(pp, vv, color = '#ff0000', linewidth = 1.0, linestyle = '-', label = 'T=%f'%(T)) #Replot the equation of state
ax1.plot(Pp, vp, color = 'cyan', linewidth = 3.0, linestyle = '-', label = '$Real$', alpha = 0.4) #Plot the "real" curve
fig.canvas.draw()
```

If you close that window, you will need to re-run the code from where the figure was first called: somewhere in lines 40-45.

Saturation Pressure or Vapour Pressure.

It now remains to find the pressure at which $\phi_L = \phi_G$. As you can see from the plots we have already made, there is one pressure for each temperature where this happens. This special pressure is called the *Saturation Pressure* or *Vapour Pressure* and it is a function of the temperature. It now remains to find it. But before we do that, lets plot a more useful quantity in the *ax2*. This is the difference $\log\phi_L - \log\phi_G$ vs P. So here we go ...

But hang on! We have already written the code we need for this above (somewhere in lines 55-60). We could copy and paste it, of course but that is **not** good practice. We *have* to find a way of reusing code. So we take the same lines of code and put them in a function:

Reusing Code

In [60]:
```
def getPhiLG(T, P):  #See how easy this is.  Just the same code slightly modified.
    listZ = Benzene.Z(T, P)
    if len(listZ) == 2:
        [ZL, ZG] = listZ
        lnphiL = Benzene.lnPhi(T, P, ZL); lnphiG = Benzene.lnPhi(T, P, ZG)
    else:
        Z = listZ[0] #We give the same value to ZL and ZG
        lnphiL = Benzene.lnPhi(T, P, Z); lnphiG = lnphiL + 0.0
    return lnphiL, lnphiG
```

Now lets use our brand new function!

In [61]:
```
ppnew = scipy.linspace(1.0, 4e6, 10000)
lnphiL = scipy.array([getPhiLG(T, p)[0] for p in ppnew]) #We are packing in a lot of instructions in one line! This is power!
lnphiG = scipy.array([getPhiLG(T, p)[1] for p in ppnew])
```

If you look at the code in line 57 or thereabouts, you will notice this is a much more elegant way of doing things. We have packed a lot of instructions in a single line. Let us very quickly copy the code in line 74 to our Spyder editor as an attribute *getPhiLG* in $eosClass.EOS$ template. But notice we have to make some changes. We have to add $self$ to the beginning of the argument list and everywhere that we have used $Benzene$ in (75) we have to use $self$. Also, you have to restart the kernel in order to import the latest iteration of the object. Lets do that now ...

Now that we are back, lets see if the object method attribute works:

In [62]:
```
lnphiL_method = scipy.array([Benzene.getPhiLG(T, p)[0] for p in ppnew])
lnphiG_method = scipy.array([Benzene.getPhiLG(T, p)[1] for p in ppnew])
```

In [63]:
```
##And now lets plot!
ax2.cla()  #Lets clear the axes first
ax2.plot(ppnew, lnphiL-lnphiG, color='#ff0000', linewidth = 0.5,label='diff in fugacity coefficient using Notebook function')
ax2.plot(ppnew, lnphiL_method - lnphiG_method, color='cyan', linewidth = 2, alpha = 0.5, label='diff in fugacity coefficiet us
ax2.xaxis.label.set_text('$P$')
ax2.yaxis.label.set_text('$\log{(\phi_L/\phi_G)}$')
fig.canvas.draw()
```

Using reusable code for plotting ...

It does work! The two plots line up perfectly. But the plotting is taking up too much code. Could we write a function to automate this as well? Sure!

In [64]:
```
def plotDifflnPhi(T, P, molecule, ax):
    lnphiL = scipy.array([molecule.getPhiLG(T, p)[0] for p in P])
    lnphiG = scipy.array([molecule.getPhiLG(T, p)[1] for p in P])
    ax.cla()
    ax.plot(P/1e5, lnphiL-lnphiG, color='#ff0000',linewidth=0.5)
    ax.xaxis.label.set_text('$P$ (bar)')
    ax.yaxis.label.set_text('$\log{(\phi_L/\phi_G)}$')
```

Lets similarly define a function for plotting the cubic eos.

In [65]:
```
def plotEOS(T,molecule, ax, mult=1.1):
    v = scipy.linspace(Benzene.b*mult, Benzene.b*100, 10000)
    P = Benzene.P(T, v)
    ax.cla()
    ax.plot(P/1e5, v, color = 'red', linewidth = 1.0)
    ax.xaxis.label.set_text('$P$ (bar)')
    ax.yaxis.label.set_text('$v$ (m3/kmol)')
    return scipy.linspace(1.0, max(P), 10000)
```

Lets use this!

In [99]:
```
ppnew = plotEOS(530.0, Benzene, ax1, mult=1.9)
plotDifflnPhi(530.0, ppnew, Benzene, ax2)
fig.canvas.draw()
```

Can you explain why this graph is significantly different than the graph for T=500K? Oh, and did you notice how quick it is to manipulate the plots now? Replot for 540K. See? No sweat!

Root Finding

Now we can clearly see that there is a single pressure for which $\log\left(\frac{\phi_L}{\phi_G}\right) = 0$. But how to find it? First we recast it as a root-finding problem formally. Hence we want a value of P for which $\log\left(\frac{\phi_L}{\phi_G}\right) = 0$. Or in terms of the program, we are looking for a value of P for a given value of T so that a function called $d\log Phi(P, T, molecule)$ returns a value of 0. We haven't defined this function yet. So lets do so:

In [100]:
```
def dlogPhi(P, T, molecule):
    lnPhiL, lnPhiG = molecule.getPhiLG(T, P)
    return lnPhiL - lnPhiG
```

In the above function, P is the *unknown* and is a *necessary argument* while T and $molecule$ are the additional (optional) arguments depending on the problem. In our case, of course, they are all *necessary*. But you can imagine root finding problems where they are not. Anyway, we have to find P which will drive dlogPhi to zero.

Now, you know a little bit about root-finding algorithms. So can you see immediately why Newton-Raphson will not work here? **Hint**: *What will you use for an initial guess?*

We know that the P_{sat} *has* to be between the pressure maxima and minima in the Pv diagram. So how do we find those? Well, we know that $\frac{\partial P}{\partial v} = 0$ at the optima. So lets use that. From the cubic equation of state: $P =$

In [101]:
```
P, v, R, T, Z, a, b, alp, s, e = symbols('P, v, R, T, Z, a, b, alpha, sigma, epsilon')
RHS = R*T/(v-b) - a*alp/(v+e*b)/(v+s*b); RHS
```

Out[101]:

$$\frac{RT}{-b+v} - \frac{a\alpha}{(b\epsilon+v)(b\sigma+v)}$$

Lets differentiate wrt v

In [102]:
```
diffRHS = diff(RHS, v); diffRHS
```

Out[102]:

$$-\frac{RT}{(-b+v)^2} + \frac{a\alpha}{(b\epsilon+v)(b\sigma+v)^2} + \frac{a\alpha}{(b\epsilon+v)^2(b\sigma+v)}$$

All together now!

In [103]:
```
diffRHS = together(diffRHS); diffRHS
```

Out[103]:

$$\frac{-RT(b\epsilon+v)^2(b\sigma+v)^2 + a\alpha(-b+v)^2(b\epsilon+v) + a\alpha(-b+v)^2(b\sigma+v)}{(-b+v)^2(b\epsilon+v)^2(b\sigma+v)^2}$$

We are only interested in the numerator:

In [104]:
```
nr = fraction(diffRHS)[0].expand(); nr = (nr/(R*T)).expand(); nr = collect(nr,v); a0 = nr.coeff(v,0); a1 = nr.coeff(v,1); a2 =
```

Out[104]:

$$-b^4\epsilon^2\sigma^2 - v^4 + v^3\left(-2b\epsilon - 2b\sigma + 2\frac{a\alpha}{RT}\right) + v^2\left(-b^2\epsilon^2 - 4b^2\epsilon\sigma - b^2\sigma^2 + \frac{a\alpha b\epsilon}{RT} + \frac{a\alpha b\sigma}{RT} - 4\frac{a\alpha b}{RT}\right) + v\left(-2b^3\epsilon^2\sigma - 2b^3\epsilon\sigma^2 - 2\frac{a\alpha b}{RT}\right.$$

In [105]:
```
print a0
```

```
-b**4*epsilon**2*sigma**2 + a*alpha*b**3*epsilon/(R*T) + a*alpha*b**3*sigma/(R*T)
```

In [106]:
```
print a1
```

```
-2*b**3*epsilon**2*sigma - 2*b**3*epsilon*sigma**2 - 2*a*alpha*b**2*epsilon/(R*T) - 2*a*alpha*b**2*sigma/(R*T) +
2*a*alpha*b**2/(R*T)
```

In [107]:
```
print a2
```
```
-b**2*epsilon**2 - 4*b**2*epsilon*sigma - b**2*sigma**2 + a*alpha*b*epsilon/(R*T) + a*alpha*b*sigma/(R*T) - 4*a*alpha*b/(R*T)
```

In [108]:
```
print a3
```
```
-2*b*epsilon - 2*b*sigma + 2*a*alpha/(R*T)
```

In [109]:
```
print a4
```
```
-1
```

Substitute numbers:

In [110]:
```
T = 530.0; R = 8314.0; epsilon = srk.epsilon; sigma = srk.sigma; alpha = Benzene.alpha(T); a = Benzene.a; b = Benzene.b
```

In [111]:
```
a0 = b**4*epsilon**2*sigma**2 - a*alpha*b**3*epsilon/(R*T) - a*alpha*b**3*sigma/(R*T)
a1 = 2*b**3*epsilon**2*sigma + 2*b**3*epsilon*sigma**2 + 2*a*alpha*b**2*epsilon/(R*T) + 2*a*alpha*b**2*sigma/(R*T) - 2*a*alpha
a2 = b**2*epsilon**2 + 4*b**2*epsilon*sigma + b**2*sigma**2 - a*alpha*b*epsilon/(R*T) - a*alpha*b*sigma/(R*T) + 4*a*alpha*b/(R
a3 = 2*b*epsilon + 2*b*sigma - 2*a*alpha/(R*T)
a4 = 1.0
roots = scipy.roots([a4, a3, a2, a1, a0])
roots
```

Out[111]:
```
[ 0.50983276  0.21626743  0.05625713 -0.04104946]
```

Alright! There is a problem though. There are *four* roots. We wanted only two. Which ones to pick? Lets take a look at a number first:

In [112]:
```
Benzene.b
```

Out[112]: 0.0825176847861

Ah! It appears that only two of the roots are in the physical region of $v > b$. So a little list comprehension:

In [113]:
```
roots = scipy.array([root for root in roots if root > Benzene.b]); roots
```

Out[113]:
```
[ 0.50983276  0.21626743]
```

In [114]:
```
Proots = Benzene.P(T, roots)
```

And plot!

In [115]:
```
ax1.scatter(Proots/1e5, roots, s=10, color = 'blue') #Remember to convert to bars!!
fig.canvas.draw()
```

Yay!! Lets see if our abstracted code gives the same result:

In [116]:
```
boolean, P1, P2 = Benzene.getPborder(T); Pmin = min([P1, P2]); Pmax = max([P1, P2])
Pmin, Pmax, Proots
```

Out[116]:
```
(2042410.86897, 3699734.51108, [ 3699734.51108435  2042410.86896614])
```

Exactly the same!! How wonderful this feels ... But we are not done. We have still to find P_{sat}. But we have it surrounded! Loose the hounds!

In [117]:
```
import scipy.optimize
Pguess = 0.5*(Pmin+Pmax) if Pmin > 0.0 else 0.5*Pmax  #Can you figure out what we have done here?
solution = scipy.optimize.newton(dlogPhi, Pguess, args = (T, Benzene))
solution
```

Out[117]: 3288119.64499

Well that looks right. What does our abstracted code say?

In [118]:
```
boolean, Psat = Benzene.Psat(T); boolean, Psat
```

Out[118]: $(True, \quad 3288119.64499)$

Yahoo!! But is it right? Lets find out:

In [120]:
```
vv = scipy.linspace(1.1*Benzene.b, 100.0*Benzene.b, 10)
ax1.plot(len(vv)*[Psat/1e5], vv, color='blue',linestyle = '-.',linewidth = 1)
ax2.scatter([Psat/1e5],[dlogPhi(Psat, T, Benzene)], color='green', s=10)
fig.canvas.draw()
```

Ja! Alles goed! Heel mooi! Now over to you. How will you find latent heat of vapourization from the cubic equation of state?

In [119]:

In [119]:

In [119]:

In [ ]: