IPyTherm Documentation

Versión 0

Andrés

25 de septiembre de 2018

Índice general

PyTher (Python to Thermodynamics) es una biblioteca open source orientada a cálculos del comportamiento termodinámicos de fases.

Contacto: andres.pyther@gmail.com

Contents:

CAPÍTULO 1

1. Instalación de PyTher

1.1 1.2 Requisitos

Para realizar la instalación de *PyTher* se requiere tener pre-instalado *Jupyter Notebook* y *Python*.

1.2 1.3 Instalación de Jupyter utilizando Anaconda

Se recomienda instalar *Anaconda* porque de forma simple no solo instala *Python* y *Jupyter Notebook* sino que también un gran número de librerías para computación científica.

Pasos de instalación:

1. Descargar Anaconda. Se recomienda la descarga de Anaconda superior a Python 3.X.
2. Instalar la versión de Anaconda que descargo, siguiendo las instrucciones según sea el caso
3. Muy bien, ya se instalo Jupyter Notebook. Ahora vamos a probarlo en una línea de comandos y se ejecuta:

jupyter notebook

Luego de tener abierto *Jupyter Notebook* se puede realizar la instalación de *PyTher* desde una celda del mismo *Jupyter Notebook* utilizando *PyPi* con la instrucción:

```
!pip install pyther
```

```
Requirement already satisfied: pyther in ./anaconda3/lib/python3.5/site-packages
```

NO olvidar el símbolo *!** inicial*

Luego de instalar PyTher, se puede probar con una importación simple de la librería con el sigiente ejemplo:

```
import pyther as pt
```

```
print("PyTher version: ", pt.__version__)
```

```
PyTher version:  0.6
```

En este caso se accedió al atributo ***version*** de PyTher para verificar su correcta instalación.

De esta forma, ya se encuentra disponible la librería PyTher para ser utilizada con los ejemplos que vienen más adelante.

CAPÍTULO 2

2. PyTher: an Open Source Python library for Thermodynamics

2.1 2.1 Abstract

PyTher se desarrolla utilizando el framework de entorno interactivo IPython notebook para el analisis de problemas en termodinámica del equilibrio de fases permitiendo el acceso a su código para el diseño de algoritmos, su implementación, modificación y ampliación de funcionalidades por parte de usuarios avanzados siguiendo la filosofía open source and open science.

PyTher se enmarca en la tesis con título:

Desarrollo de una librería open source para el cálculo de propiedades termodinámicas,equilibrios y diagramas de fases en base a ecuaciones de estado, para sustancias puras, sistemas binarios y mezclas multicomponente

2.1.1 Objetivo General

Desarrollar una librería open source basada en la plataforma Jupyter y lenguaje Python, para el cálculo de propiedades termodinámicas, equilibrios y diagramas de fases, para sustancias puras, sistemas binarios y mezclas multicomponente.

Keywords Thermodynamics; equation of state; open source; IPython notebook

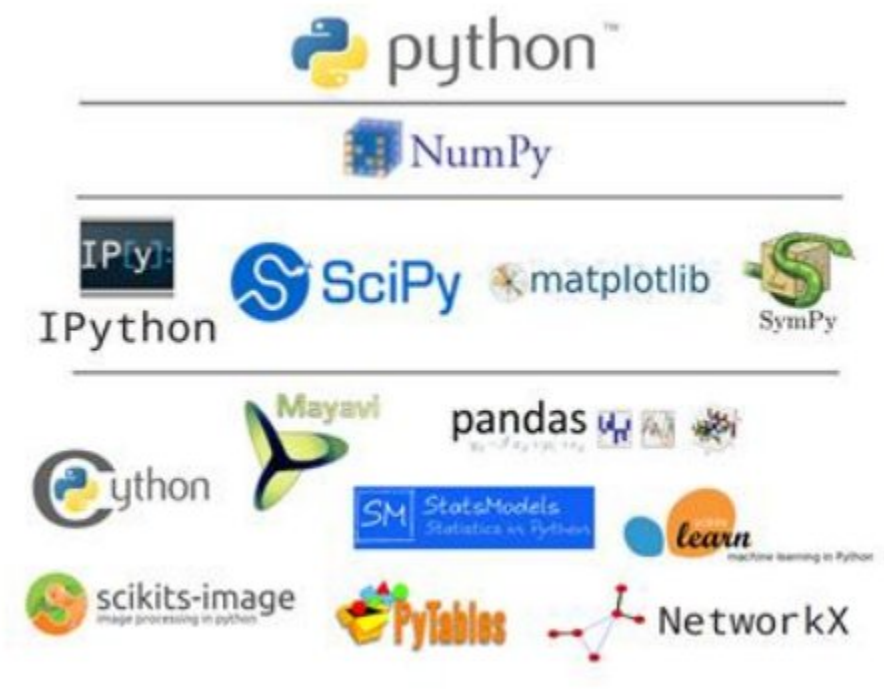
2.2 2.2 Introduction.

En la actualidad en la sociedad de la información se cuenta con acceso a una gran cantidad de datos que requieren ser procesados para obtener conocimiento de los mismos. En la investigación científica con el aumento de la capacidad y acceso a procesadores de computo más capaces se ha integrado a las metodologías tradicionales de investigación del trabajo experimental y teórico, la rama de la computación científica como una tercera componente que toma características de éstas dos, debido principalmente a la capacidad de realizar trabajo de experimentación numérica de acuerdo a diferentes modelos teóricos. De esta forma, en la ciencia actual se requiere del uso del cálculo científico en diferentes áreas de la ciencia y tecnología, principalmente para llevar a cabo experimentación numérica de teorías, además del moldeamiento y simulación diferentes fenómenos y sistemas para su posterior optimización.

Para realizar cálculos científicos no soló se requiere el entendimiento de las teorías bajo las cuales se desarrollan los modelos y ecuaciones que analizan fenómenos de diferentes áreas de las ciencias, también es necesaria la capacidad de implementar éstos modelos en un lenguaje de programación para acceder eficientemente al computo y solución de

miles de ecuaciones. Hace varias décadas se han desarrollado lenguajes de programación especializados en el cálculo científico, destacandose C, C++ y FORTRAN, que siguen siendo empleados por científicos pero al mismo tiempo van reemplazandoce por lenguajes de programación modernos con mayores características, resaltando el lenguaje de programación Python, como un lenguaje de programación libre, multiplataforma, interpretado y multiparadigma que es poderoso y simple de aprender, además permite la implementación de estructuras de datos de forma simple, eficiente, moderna y elegante con su especial enfoque de programación orientado a objetos y una sintaxis simple con un tipado dinámico que facilita el mantenimiento del código, que cuenta con múltiples desarrollos especializados en el cálculo científico (Pérez & Granger, 2007)[1] (Perez, 2015)[2].

Figura 1. Ecosistema científico de Python



De esta forma, tradicionalmente dentro de la metodología de una investigación científica se ha tenido especial atención a los equipos (Hardware) y métodos que se han empleado durante la labor científica, dejando en un segundo plano el software-librería que ya tiene un lugar importante en el mundo científico por medio de herramientas como Excel, Chemdraw, Matlab y lenguajes de programación como Python, R y SQL, por nombrar algunos. Por tal motivo, importantes referencias de la comunidad científica han comenzado a dar espacios especializados para la divulgación del software-librería científico como lo es la sección Nature-Toolbox de la revista Nature que comenzó a ser publicada en septiembre de 2014 (Nature Editorial, 2014)[3]. En Nature-Toolbox se ha presentado casos como el de Caladis.org, el cual es un proyecto open source desarrollado por los matemáticos Lain Jhonston y Nick Jones del Imperial College London que por medio de su librería Caladis online facilitan diferentes cálculos científicos de modelos estadísticos en biología (Van Noorden, 2015)[4].

Dentro de los articulos de Nature-Toolbox, la plataforma IPython recibe atención especial pues es tiene las ventajas del lenguaje Python además de permitir el tipado y ejecución del código online por uno o varios usuarios con la posibilidad de combinar codigo Python con procesadores de textos como Latex, imágenes, vídeos y soportar HTML (Helen Shen,

[1] Pérez, F., Granger, B.E.: IPython: a System for Interactive Scientific Computing. Computing in Science and Engineering 9(3) (May 2007) 21–29

[2] Fernando Perez. (1 de Abril de 2015) A gallary of interesting Ipython Notebooks. GitHub IPython. Web: https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks#reproducible-academic-publications

[3] Helen Shen. Interactive notebooks: Sharing the code. Nature. Toolbox. 5 Noviembre 2014. ISSN: 0028-0836

[4] Richard Van Noorden. My digital toolbox: Ecologist Ethan White on interactive notebook. Nature. Toolbox. 30 de Septiembre 2014.

2014)[5]. IPyhon ya está presente en artículos científicos con sus notebook (Van Noorden. 2014; Tippmann 2014)[6], los cuales son empelados como una extensión dinámica e interactiva del artículo científico publicado permitiendo manipular el código de programación Python y reproducibilidad en tiempo real sin costo de tiempo de programación por parte de los lectores, haciendo más eficiente la comunicación de resultados de investigación. Para ver una lista más extensa de publicaciones científicas con IPython notebook puede consultar el repositorio en GitHub (Fernando Perez, 2015)[7].

Por tanto, se plantea desarrollar una librería open sourse en lenguaje Python usando la plataforma IPython para calcular el comportamiento termodinámico de fases de hidrocarburos usando ecuaciones de estado Soave Redlich Kwong (SKR), Peng Robinson (PR) y (RKPR), basándose en los algoritmos y software (GPEC, FLUIDS) desarrollado por el grupo IDTQ (Cismondi and Mollerup, 2005)[8]. Dando como resultados dos aportes principales, por un lado el desarrollo de software científico especializado en termodinámica de hidrocarburos con características que no se han encontrado en la revisión de la literatura especializada que se ha hecho y la incursión en el campo de la publicación científica con la moderna tecnología de IPython notebook.

2.3 2.3 Actividades y Metodología

La primera fase del trabajo consistirá en el desarrollo de los denominados cálculos directos mostrados en la figura 2. Los cálculos directos se basan en la implementación de la metodología modular (Cismondí et al. 2007)[9], para el cálculo de la función de la energía de Helmholtz, propiedades termodinámicas (presión, entalpía, entropía) y la fugacidad para sustancias puras y mezclas multicomponente usando las ecuaciones cubicas de estado SRK, PR y RKPR.

La segunda fase del trabajo corresponde a los denominados cálculos iterativos, que tienen como base la implementación de la primera fase, puesto que se requiere del cálculo de la fugacidad y propiedades termodinámicas como parte de los algoritmos para resolver los sistemas de ecuaciones planteados de acuerdo a las especificaciones de cada sistema siguiendo la regla de las fases de gibbs. En esta fase, se implementarán métodos de cálculo para puntos críticos, propiedades volumétricas, puntos azeotrópicos, regiones bifásicas, cálculo de distribución de componentes en condiciones especificadas como los flash(T,P) y flash(T,v), entre otros que se muestran en la figura 2.

Para la tercera fase se utilizará un método de continuación presentado por (Cismondí et al., 2008)[10], para trazar líneas PV, líneas de región bifásica, azeotrópica, critica, entre otras, resaltando la importancia de estas, para el entendimiento de los diagramas globales de equilibrio de fases. También se realiza una revisión de la literatura especializada en el diseño de software, puesto que este trabajo es interdisciplinar basado en el concepto de desarrollo de software evolutivo adaptado (Sommerville and Ian, 2004)[11], en el cual luego de realizar un desarrollo del software inicial, se expone a un panel de usuarios, para ir refinando el software en versiones preliminares (versiones beta) hasta lograr una versión del software que cumpla con los requerimientos planteados.

Figura 2. Plan simplificado de la tesis doctoral.

[5] Richard Van noorden. My digial toolbox Back of the envelope biology. 20 Marzo 2015. Nature-Toolbox. doi:10.1038/nature.2015.17140

[6] Sylvia tippmann. My digital toolbox: Nuclear engineer Katy huff on version-control systems.Nature. Toolbox. 29 de Septiembre 2014. doi:10.1038/nature.2014.16014

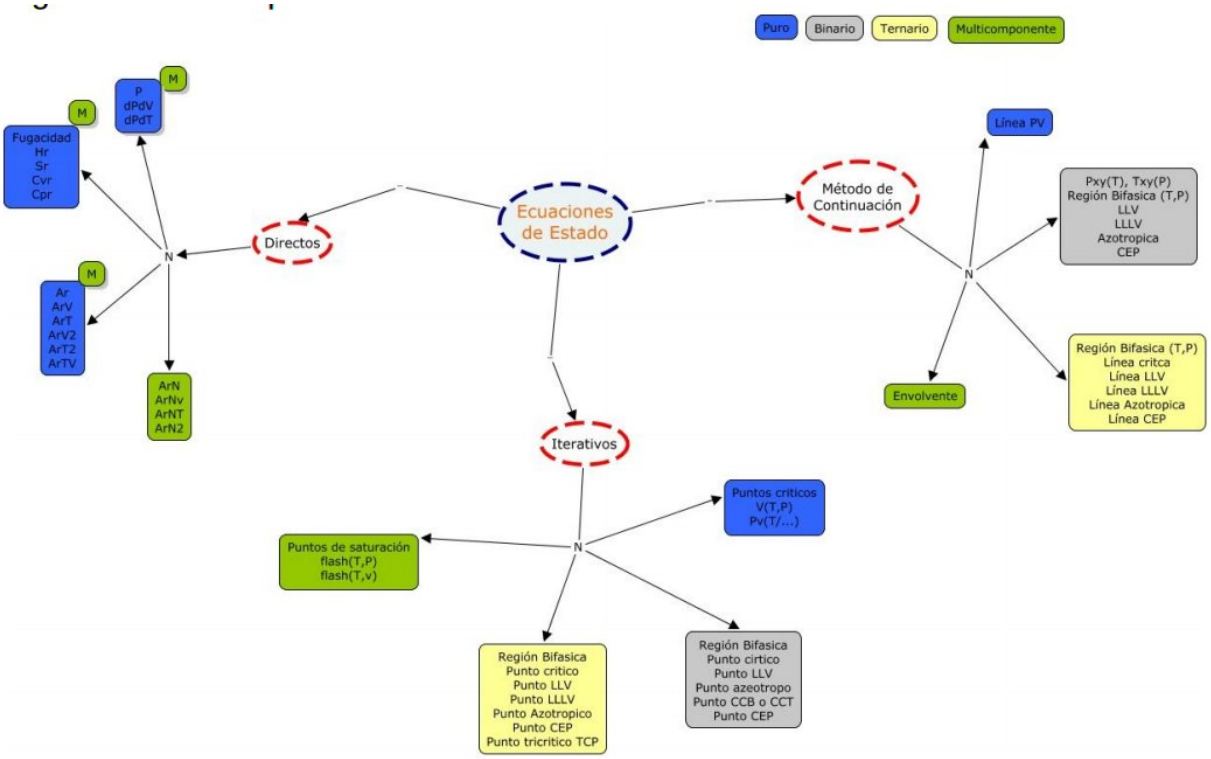
[7] Nature Editorial. The digital toolbox. Nature 513, 6 (04 September 2014) doi:10.1038/513006b

[8] Cismondi, M. And Mollerup, J., Development and application of the three-parameter RK-PR equiation of state, Fluid Phase Equilibria, 232 (2005) 74-89.

[9] Cismondi , M., Michelsen, M.L., Global phase equilibrium calculations: Criticallines, critical end points and liquid-liquid-vapour equilibrium in binary mixtures, J.Supercrit. Fluids. 39 (2007) 287–295

[10] Cismondi , M., Michelsen, M.L., Zabaloy, M.S., AUTOMATED GENERATION OF PHASE DIAGRAMS FOR SUPERCRITICAL FLUIDS FROM EQUATIONS OF STATE, 11th Eur. Meet.Supercrit.Fluids. (2008).

[11] Sommerville, Ian (2004) Software Enginnering, 7th edition, Pretince Hall Traducción al español por el Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Alicante (2005).



Este trabajo se apoya en el amplio conocimiento que tiene el grupo de investigación dirigido por el Dr. Martín Cismondí, en el desarrollo e implementación de algoritmos de métodos numéricos en lenguajes de programación (principalmente Python) para simular el comportamiento de fases de fluidos supercríticos (Cismondi et al, 2007; Rodríguez et al, 2011[12]; Cismondi et al, 2008)[13] incluyendo sistemas que involucran precipitación de sólidos (Rodríguez et al, 2011).

Dentro de los principales métodos numéricos que se consideran para el desarrollo de este trabajo, se encuentran los métodos de continuación por homotopía (Allgower and Georg, 1990)[14], para solucionar los sistemas de ecuaciones altamente no lineales y sensibles a los estimados iniciales de las variables del modelo matemático, además de utilizar el concepto de homotopía termodinámica para automatizar la solución del modelo desde condiciones de baja presión y temperatura hasta la solución del modelo en condiciones supercríticas, lo cual reduciría el esfuerzo y tiempo de computo requerido (Pisoni, 2014)[15].

2.4 2.3. Referencias.

[12] Rodriguez-Reartes S.B., Cismondi M., Zabaloy M.S., Computation of solid-fluidfluid equilibria for binary asymmetric mixtures in wide ranges of conditions, J. Supercrit. Fluids. 57 (2011) 9–24.

[13] Rodriguez-Reartes S.B., Cismondi M., Zabaloy M.S., Modeling approach for the high pressure solid-fluid equilibrium of asymmetric systems, Ind. Eng. Chem. Res. 50 (2011) 3049–3059.

[14] Allgower, E. L. and Georg, K. Introduction to numerical continuation methods. USA: Colorado StateUniversity/Springer, 1990. 388 p.

[15] Pisoni, G., Cismondi, M., Cardozo-Filho, L., Zabaloy, M.S. Generation of characteristic maps of the fluid phase behavior of ternary systems. Fluid Phase Equilibria 362 (2014) 213–226

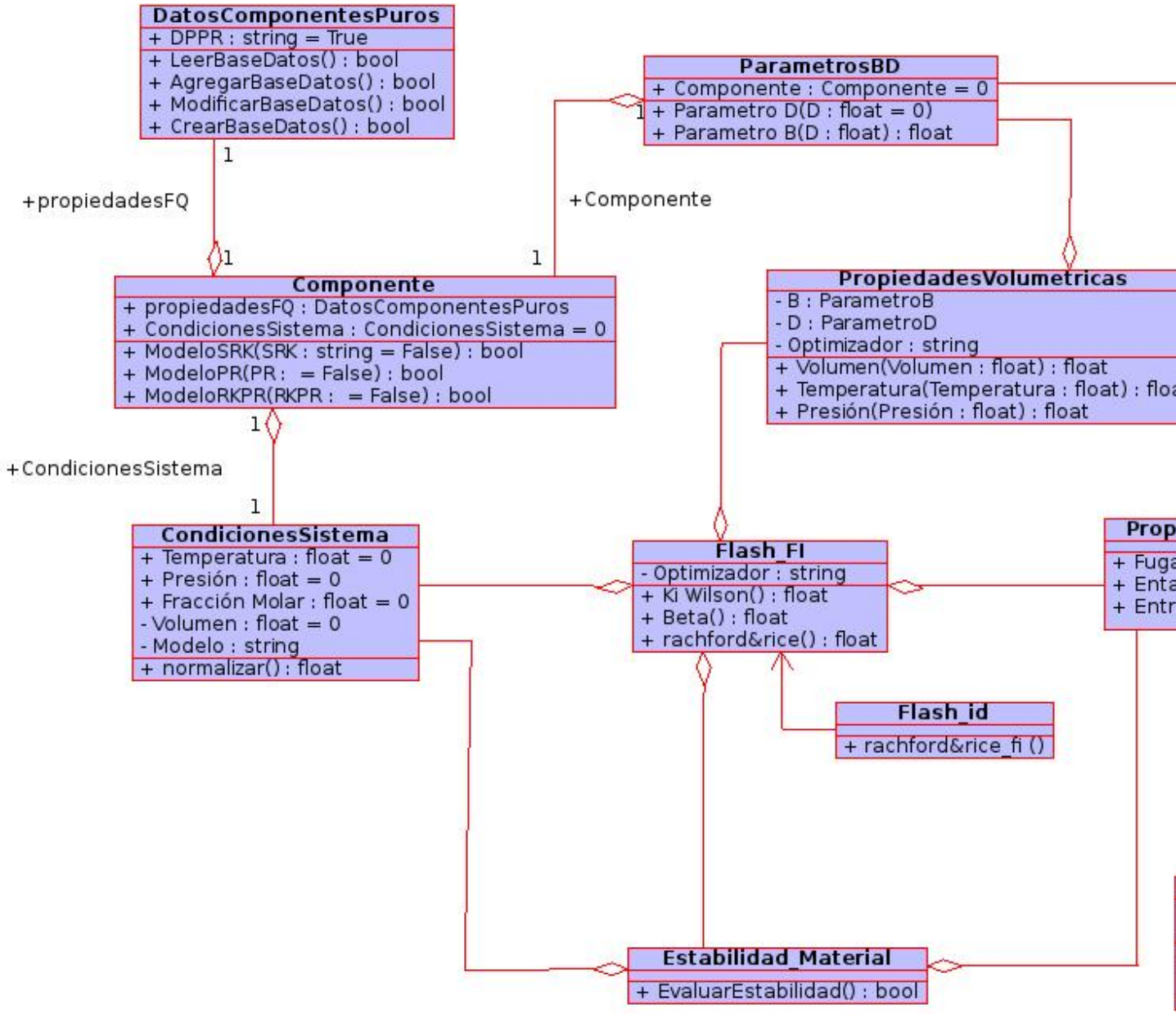
CAPÍTULO 3

3. Diagrama de Clases UML

3.1 3.1 Introduccion

En esta sección se presenta una breve explicación de la primera parte del diagrama de clases que corresponde al cálculo de la fugacidad de un componente puro. Por tanto, los nombres de clases, atributos y métodos están sujetos a modificaciones, actualizaciones y correciones, junto con la estructura de las relaciones entre clases presentadas aquí, así como también las que se puedan adicionar como resultado de la continua revisión del diagrama de clases.

Figura 1. Diagrama de Clases Pyther



La primera parte del diagrama de clases corresponde a:

1. **DatosComponentesPuros**
2. **CondicionesSistema**
3. **Componente**
4. **ParametrosBD**
5. **PropiedadesVolumetricas**
6. **ModulosMM**

7. **PropiedadesTermodinamicas**

La segunda parte del diagrama de clases que será comentado en el siguiente avance corresponde a:

8. **SolidoPuro**
9. **Solido Fluido**
10. **RegresionParametros**
11. **Flash_i**
12. **Flash_Fi**
13. **Estabilidad_Material**
14. **Interfaz Gráfica**

3.2 3.2 Clase DatosComponentesPuros

En la primera clase **DatosComponentesPuros** se tiene:

- Atributos

DIPPR = Este atributo es una variable tipo string que corresponde al nombre que tiene el archivo que actualmente hace de «base de datos» provisional y se verifica que el nombre del archivo coincida con el preestablecido **DPPR** para mostrar por pantalla si se ha cargado o no los datos correctamente. Cuando se adicione la posibilidad de otras «bases de datos», en esta clase se deberá contar con más atributos para manipularlas adecuadamente.

- Métodos

LeerBaseDatos() = Carga los datos del archivo «DPPR» en una variable del sistema para poder manipular dichos datos a conveniencia.

AgregarBaseDatos() = Carga los datos de un archivo con nombre diferente al archivo por defecto «DPPR». Nota: Falta generalizar el formato en el que se pretatarian los diferentes archivos con datos supuestos para que se puedan manipular dentro del sistema.

ModificarBaseDatos() = Crea una copia del archivo «DPPR» en el cual se modifica uno o más valores de los registros del archivo o adiciona un campo nuevo cuyo nombre es especificado por el usuario. Falta generalizar la opción dehacer una agrupación de componentes de acuerdo a un criterio para crear dichos «nuevos» pseudocomponentes.

CrearBaseDatos() = Crea un archivo con datos obtenidos durante la realización de cálculos, por ejemplo la regresión de parametros o puntos importantes de diagrama de fases por mencionar algunas posibilidades para que dicha información se almacene de forma estructurada para su uso en calculos posteriores sin requerir realizar de nuevo el calculo. Actualmente en pruebas.

3.3 3.3 Clase CondicionesSistema

En la segunda clase **CondicionesSistema**

Esta clase tiene como objetivo capturar del usuario las condiciones del sistema al cual se realizará los cálculos, como lo son temperatura, presión, fracción molar, volumén (según sea el caso), el modelo (ecuación de estado/modelo sólido puro) y el componente.

- Atributos

Se tienen los siguientes atributos

1 Temperatura 2 Presión 3 Fracción Molar 4 Volumen 5 Modelo 6 Componentes

3.4 3.4 Clase Componente

Esta clase tiene como objetivo la definición del o los componentes que se manejaran para realizar un cálculo con base a los registros (que se identifican con el nombre de una sustancia química) seleccionados de la clase **DatosComponentesPuros** a las condiciones establecidas en la clase **CondicionesSistema**. Luego se crea cada componente de acuerdo al modelo especificado en la clase **CondicionesSistema**), por ejemplo METHANE-SRK.

- Atributos

propiedadesFQ = Corresponde a un array que contiene las propiedades (temperatura critica, presión critica y factor acentrico) que se definió en la selección del nombre de la sustancia química que se quiere utilizar.

CondicionesSistema = Corresponde a un array que contiene la definición de la temperatura, presión fracción molar, modelo y nombre de la sustancia química que se quiere utilizar.

- Métodos

ModeloSRK ModeloPR ModeloRKPR

Los métodos (ModeloSRK, ModeloPR, ModeloRKPR) corresponden al cálculo de los parametros requeridos para los modelos SKR, PR, RKPR según sea el caso que se especifique en la clase **CondicionesSistema**.

3.5 3.5 Clase ParametrosBD

Esta clase obtiene la información del o los **componentes**, por ejemplo («METHANE SRK»), para calcular los parámetros B y D correspodientes.

- Atributos

componente = es un array que contiene los parámetros necesarios para cálcular las variables B y D

- Métodos

Parametro B = Calcula el parametro B con la información provista en **componente** Parametro D = Calcula el parametro D con la información provista en **componente**

3.6 3.6 Clase PropiedadesVolumetricas

Esta clase tiene como objetivo la manipulación de la ecuación de estado cúbica para determinar la presión, temperatura o volúmen según sea el caso de las especificaciones dadas en la clase **CondicionesSistema**. Por ejemplo, al especificar P, T y ni, encontrar el V en dichas condiciones y un modelo y parametros determinados. Esta clase se separa de de la clase **Modulos MM** (se muestra a continuación) para aprovechar el enfoque modular y acceder al calculo de propiedades volumetricas de forma independiente del calculo de propiedades termodinámicas y sus correspondientes modulos (funcion de helmholtz, primeras derivas y segundas derivadas), según sean requeridas (las propiedades volumetricas).

- Atributos

Parametro B = parametro B determinado en la clase **ParametrosBD** Parametro D = parametro D determinado en la clase **ParametrosBD** Optimizador = corresponde a la selección y especificación de los parámetos requeridos para acceder y ejecutar un método ńumérico de resolución de ecuaciones no lineales de la librería Scipy.

- Métodos

Volumen = cálcua el volumén con una ecuación de estado para una P, T y ni especificados Temperatura = cálcua la temperatura con una ecuación de estado para una P, V y ni especificados (Falta por implementar). Presión = cálcua la presión con una ecuación de estado para una T, V y ni especificados

En caso de especificiar el volumen V, se calcula la presión P para la temperatura T y ni especificada. Para el caso contrario de especificar la presión P, se determina el volumen V para la temperatura T y ni especificada.

3.7 3.7 Clase ModulosMM

Esta clase se tiene como objetivo calcular la función de energía de Helmholtz siguiendo el enfoque modular de Michelsen & Mollerup, partiendo de los parametros B y D obtenidos en la clase **ParametrosBD** y la propiedad volumetrica «volumen» o «presión» según sea el caso especificado (Esta clase tiene la capacidad de navegar y acceder a los otros atributos como lo son la temperatura, fracción molar). En esta clase se tienen tres métodos, que calculan la función de energía de Helmholtz ya mencionada, las primeras derivadas de esta función con respecto a las variables como son: emperatTura, Presión, Volumen y Número de moles (para el caso de la fracción molar hay relaciones que permiten obtener las derivadas en función de las fracciones molares a partir de las derivadas del númerod de moles), así mismo para el caso de las segundas derivadas de la función de energía de Helmholtz.

- Atributos

Parametro B = parametro B determinado en la clase **ParametrosBD** Parametro D = parametro D determinado en la clase **ParametrosBD** Volumen = corresponde al volumén calculado con una ecuación de estado para una P, T y ni especificados Presión = corresponde a la presión con una ecuación de estado para una T, V y ni especificados

En esta clase los atributos de presión P, volumen V se acceden desde la clase **PropiedadesVolumetricas**y como ya se ha mencionado estos pueden ser una especificación o calculados según sea el caso.

- Métodos

funciónHelmholtz = este método calcula la función de energía de Helmholtz con los parametros indicados para la especificación del modelo (por ejemplo METHANE SKR) y las condiciones del sistema.

primerasDerivadas = este método calcula las primeras derivadas de la función de energía de Helmholtz con respecto a las variables como son: Temperatura, Presión, Volumen y Número de moles (para el caso de la fracción molar hay relaciones que permiten obtener las derivadas en función de las fracciones molares a partir de las derivadas del númerod de moles), a las vcon los parametros indicados para la especificación del modelo (por ejemplo METHANE SKR) y las condiciones del sistema.

segundasDerivadas = este método calcula las segundas derivadas de la función de energía de Helmholtz con respecto a las variables como son: Temperatura, Presión, Volumen y Número de moles (para el caso de la fracción molar hay relaciones que permiten obtener las derivadas en función de las fracciones molares a partir de las derivadas del númerod de moles), , a las vcon los parametros indicados para la especificación del modelo (por ejemplo METHANE SKR) y las condiciones del sistema.

3.8 3.8 Clase PropiedadesTermodinamicas

En esta clase se tiene los métodos para calcular las propiedades termodinámicas siguiendo el enfoque modular de Michelsen & Mollerup. Esta clase no tiene atributos y sus métodos corresponden a las propiedades termodinámicas como: Fugacidad, Entalpía y Entropía. (Se está implementando para el método de la energía libre de Gibbs)

- Atributos

No tiene atributos.

- Métodos

Fugacidad = este método calcula la fungacidad de un componente puro o mezcla multicomponente, según sea la especificación (puro o multicomponente) siguiendo el enfoque modular de Michelsen & Mollerup partiendo de los métodos de la clase **ModulosMM**, que ya contienen toda la información pertinente para realizar el calculo de la propiedad termodinámica.

Entalpía = este método calcula la entalpía de un componente puro o mezcla multicomponente, según sea la especificación (puro o multicomponente) siguiendo el enfoque modular de Michelsen & Mollerup partiendo de los métodos de la clase **ModulosMM** para el calculo de las primeras y segundas derivadas de la función de energía de Helmholtz, que ya contienen toda la información pertinente para realizar el calculo de la propiedad termodinámica.

Entropía = este método calcula la entropía de un componente puro o mezcla multicomponente, según sea la especificación (puro o multicomponente) siguiendo el enfoque modular de Michelsen & Mollerup partiendo de los métodos de la clase **ModulosMM** para el calculo de las primeras y segundas derivadas de la función de energía de Helmholtz, que ya contienen toda la información pertinente para realizar el calculo de la propiedad termodinámica.

Nota: para el caso de las propiedades termodinámica aún no se han terminado de realizar las pruebas que corroboren que los calculos implementados tienen resultados correctos.

3.9 3.9 Clase Estabilidad_Material

En esta clase falta por empezar a documentarla.

CAPÍTULO 4

4. Thermodynamics correlations for pure components

En esta sección se muestra la clase *Thermodynamics_correlations()*, la cual permite realizar el cálculo de propiedades termodinámicas de sustancias puras como una función de la temperatura. En este caso se pueden tener 6 situaciones:

1. Especificar una sustancia pura sin especificar una temperatura. En este caso por defecto la propiedad termodinámica se calcula entre el intervalo mínimo y máximo de validez experimental para cada correlación.
2. Especificar una sustancia pura y especificar una temperatura.
3. Especificar una sustancia pura y especificar varias temperaturas.
4. Especificar varias sustancias puras sin especificar una temperatura.
5. Especificar varias sustancias puras y especificar una temperatura.
6. Especificar varias sustancias puras y especificar varias temperaturas

la clase *Thermodynamics_correlations()* es usada para calcular 13 propiedades termodinámicas de sustancias puras en función de la temperatura y se sigue la siguente convención para identificar las propiedades termodinámicas:

property thermodynamics = name property, units, correlation, equation

Las correlaciones termodinámicas implementadas son:

-**Solid_Density** = «Solid Density», «[kmol/m^3]», «A+B*T+C*T$^{2+D*T\^3+E*T}$4», 0

-**Liquid_Density** = «Liquid Density», «[kmol/m^3]», «A/B:sup:*(1+(1-T/C)*D)», 1

-**Vapour_Pressure** = «Vapour Pressure», «[Bar]», «exp(A+B/T+C*ln(T)+D*T^E) * 1e-5», 2

-**Heat_of_Vaporization** = «Heat of Vaporization», «[J/kmol]», «A*(1-Tr):sup:*(B+C*Tr+D*Tr*2)», 3

-**Solid_Heat_Capacity** = «Solid Heat Capacity», «[J/(kmol*K)]», «A+B*T+C*T$^{2+D*T\^3+E*T}$4», 4

-**Liquid_Heat_Capacity** = «Liquid Heat Capacity», «[J/(kmol*K)]», «A:sup:*2/(1-Tr)+B-2*A*C*(1-Tr)-A*D*(1-Tr):sup:‘2-C*2*(1-Tr)‘3/3-C*D*(1-Tr):sup:*4/2-D*2*(1-Tr)^5/5», 5

-**Ideal_Gas_Heat_Capacity** = «Ideal Gas Heat Capacity» «[J/(kmol*K)]», «A+B*(C/T/sinh(C/T))^2+D*(E/T/cosh(E/T))^2», 6

-**Second_Virial_Coefficient** = «Second Virial Coefficient», «[m^3/kmol]», «A+B/T+C/T:sup:*3*+*D/T*8+E/T^9», 7

-**Liquid_Viscosity** = «Liquid Viscosity», «[kg/(m*s)]», «exp(A+B/T+C*ln(T)+D*T^E)», 8

-**Vapour_Viscosity** = «Vapour Viscosity», «[kg/(m*s)]», «A*T:sup:*B/(1+C/T+D/T*2)», 9

-**Liquid_Thermal_Conductivity** = «Liquid Thermal Conductivity», «[J/(m*s*K)]», «A+B*T+C*T$^{2+D*T^3+E*T}$4», 10

-**Vapour_Thermal_Conductivity** = «Vapour Thermal Conductivity», «[J/(m*s*K)]», «A*T:sup:*B/(1+C/T+D/T*2)», 11

-**Surface_Tension** = «Surface Tension», «[kg/s^2]», «A*(1-Tr):sup:*(B+C*Tr+D*Tr*2)», 12

Para empezar se importan las librerías que se van a utilizar, que en este caso son numpy, pandas, pyther ademas de especificar que las figuras generadas se muestren dentro de las celdas de Jupyter Notebook

```
import numpy as np
import pandas as pd
import pyther as pt
import matplotlib.pyplot as plt
%matplotlib inline
```

4.1 16.1 Especificar una sustancia pura sin especificar una temperatura.

Luego se carga el archivo que contine las constantes de las correlaciones de las propiedades termodinamicas, que se llama en este caso *«PureFull_mod_properties.xls»* y se asigna a la variable *dppr_file*.

Creamos un objeto llamado **thermodynamic_correlations** y se pasan como parametros las variables **component** y **property_thermodynamics** que en el ejemplo se especifica para el componente METHANE la Vapour_Pressure

```
dppr_file = "PureFull_mod_properties.xls"

thermodynamic_correlations = pt.Thermodynamic_correlations(dppr_file)

component = ['METHANE']
property_thermodynamics = "Vapour_Pressure"

Vapour_Pressure = thermodynamic_correlations.property_cal(component, property_
↪thermodynamics)
print("Vapour Pressure = {0}". format(Vapour_Pressure))
```

```
----------------------------------------------------------------------
Pure substance without temperature especific: ['METHANE']
----------------------------------------------------------------------
Vapour Pressure = [  0.11687017   0.13272851   0.15029231   0.1696935    0.19106965
   0.21456383   0.24032459   0.26850587   0.29926689   0.33277204
   0.36919081   0.40869762   0.45147173   0.49769708   0.54756216
   0.60125987   0.65898737   0.72094595   0.78734085   0.85838113
   0.93427952   1.01525227   1.101519     1.19330257   1.29082892
   1.39432695   1.50402838   1.62016765   1.74298174   1.87271013
   2.00959463   2.1538793    2.30581038   2.46563616   2.63360692
   2.80997486   2.99499402   3.18892023   3.39201109   3.60452587
   3.82672552   4.05887261   4.30123136   4.55406758   4.8176487
   5.09224376   5.37812343   5.67556002   5.98482753   6.30620166
   6.63995987   6.98638141   7.34574742   7.71834093   8.10444699
   8.5043527    8.91834734   9.34672242   9.78977179  10.24779173
```

(continues on next page)

(proviene de la página anterior)

```
  10.7210811   11.20994139  11.71467689  12.23559478  12.7730053
  13.32722183  13.89856107  14.48734319  15.09389194  15.71853484
  16.36160334  17.02343294  17.70436342  18.40473898  19.1249084
  19.86522527  20.62604814  21.40774072  22.21067207  23.03521683
  23.88175537  24.75067404  25.64236538  26.55722832  27.4956684
  28.45809802  29.44493665  30.45661106  31.49355559  32.55621234
  33.64503148  34.76047146  35.90299928  37.07309076  38.2712308
  39.49791367  40.75364324  42.03893333  43.35430794  44.7003016 ]
```

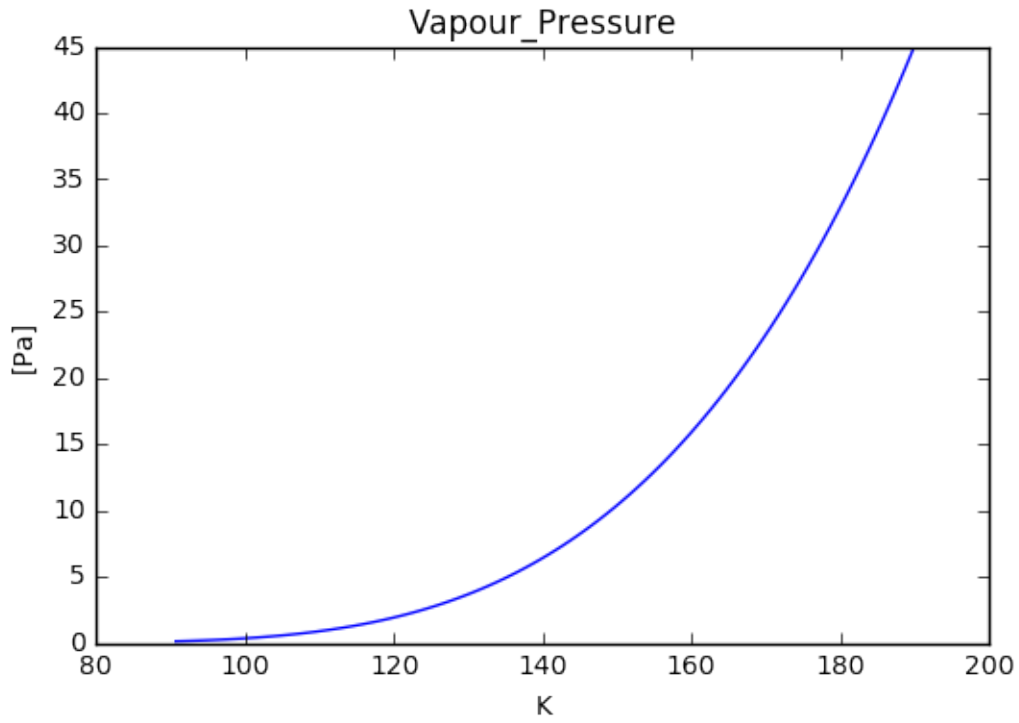
para realizar un gráfico simple de la propiedad termodinámica se utiliza el método **graphical(temperature, property_thermodynamics, label_property_thermodynamics, units)**.

En donde se pasan como argumentos la temperatura a la cual se claculó la propiedad termodinamica, los valores calculados de la propiedad termodinamica, el label de la propiedad termodinámica y las unidades correspondientes de temperatura y la propiedad termodinámica en cada caso.

```
temperature_vapour = thermodynamic_correlations.temperature
units = thermodynamic_correlations.units
print(units)

thermodynamic_correlations.graphical(temperature_vapour, Vapour_Pressure, property_
↪thermodynamics, units)
```

```
('K', '[Pa]')
```



4.2 16.2 Especificar una sustancia pura y una temperatura.

Siguiendo con la sustacia pura *METHANE* se tiene el segundo caso en el cual ademas de especificiar el componente se especifica también solo un valor de temperatura, tal como se muestra en la variable *temperature*.

Dado que cada correlación de propiedad termodinámica tiene un rango mínimo y máximo de temperatura en la cual es valida, al especificar un valor de temperatura se hace una verificación para determinar si la temperatura ingresada se encuentra entre el intervalo aceptado para cada componente y cada propiedad termodinámica. En caso contrario la temperatura se clasifica como invalida y no se obtiene valor para la propiedad termodinámica seleccionada.

```
component = ['METHANE']
property_thermodynamics = "Vapour_Pressure"
temperature = [180.4]

Vapour_Pressure = thermodynamic_correlations.property_cal(component, property_
↪thermodynamics, temperature)
print("Vapour Pressure = {0} {1}". format(Vapour_Pressure, units[1]))
```

```
------------------------------------------------------------------------
Pure substance with a temperature especific: ['METHANE']
------------------------------------------------------------------------
Temperature_enter = [180.4]
Temperature_invalid = []
Temperature_valid = [180.4]
------------------------------------------------------------------------
Vapour Pressure = [ 33.32655377] [Pa]
```

4.3 16.3 Especificar una sustancia pura y especificar varias temperaturas.

Ahora se tiene la situación de contar con un solo componente «METHANE» sin embargo, esta vez se especifica varios valores para la temperatura en las cuales se quiere determinar el correspondiente valor de una proiedad termodinámica, que como en los casos anteriores es la *Vapour_Pressure*.

```
component = ['METHANE']
property_thermodynamics = "Vapour_Pressure"
temperature = [180.4, 181.4, 185.3, 210, 85]

Vapour_Pressure = thermodynamic_correlations.property_cal(component, "Vapour_Pressure
↪", temperature)
print("Vapour Pressure = {0} {1}". format(Vapour_Pressure, units[1]))
```

```
------------------------------------------------------------------------
Pure substance with a temperature especific: ['METHANE']
------------------------------------------------------------------------
Temperature_enter = [180.4, 181.4, 185.3, '210 K is a temperature not valid', '85 K␣
↪is a temperature not valid']
Temperature_invalid = ['210 K is a temperature not valid', '85 K is a temperature not␣
↪valid']
Temperature_valid = [180.4, 181.4, 185.3]
------------------------------------------------------------------------
Vapour Pressure = [ 33.32655377  34.43422601  39.01608023] [Pa]
```

Se debe notar que al ingresar una serie de valores de temperatura, en este caso 5 valores, se obtienen solo 3 valores de la propiedad termodinámica. Esto se debe a que para este caso 2 valores de temperatura no se encuentran en el valor mínimo y máximo en donde es valida la correlación termodinámica. Por tanto, esto se avisa por medio del mensaje: *Temperature_invalid = [“210 K is a temperature not valid”, “85 K is a temperature not valid”]*

4.4 16.4 Especificar varias sustancias puras sin especificar una temperatura.

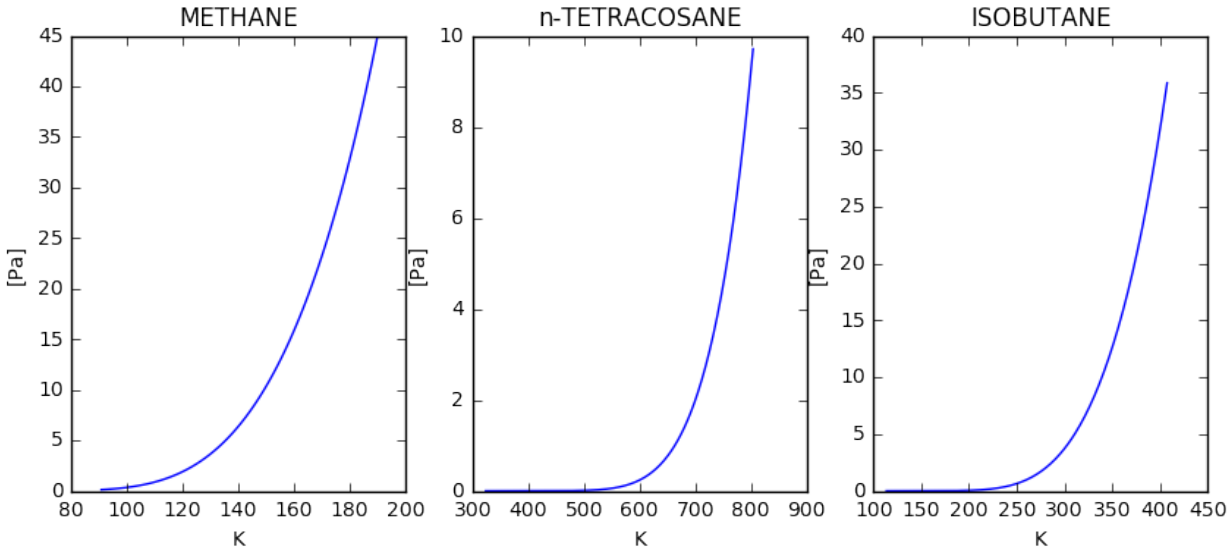
Otra de las posibilidades que se puede tener es la opción de especificar varios componentes para una misma propiedad termodinámica sin que se especifique una o más valores de temperatura. En esta opción se pueden ingresar multiples componentes sin un limite, siempre y cuando estén en la base de datos con la que se trabaja o en dado caso sean agregados a la base de datos nuevas correlaciones para sustancias puras *Ver sección base de datos*. Para este ejemplo se utiliza una *list components* con 3 sustancias puras por cuestiones de visibilidad de las gráficas de *Vapour_Pressure*.

```
components = ["METHANE", "n-TETRACOSANE", "ISOBUTANE"]
property_thermodynamics = "Vapour_Pressure"

Vapour_Pressure = thermodynamic_correlations.property_cal(components, property_
↪thermodynamics)
temperature_vapour = thermodynamic_correlations.temperature
```

por medio del método *multi_graphical(components, temperature, property_thermodynamics)* al cual se pasan los parámetros correspondiente a las sustancias puras, la temperatura a la cual se realiza el calculo de la propiedad termodinámica y los valores de la propiedad termodinámica de cada sustancia pura, para obtener la siguiente figura.

```
thermodynamic_correlations.multi_graphical(components, temperature_vapour, Vapour_
↪Pressure)
```

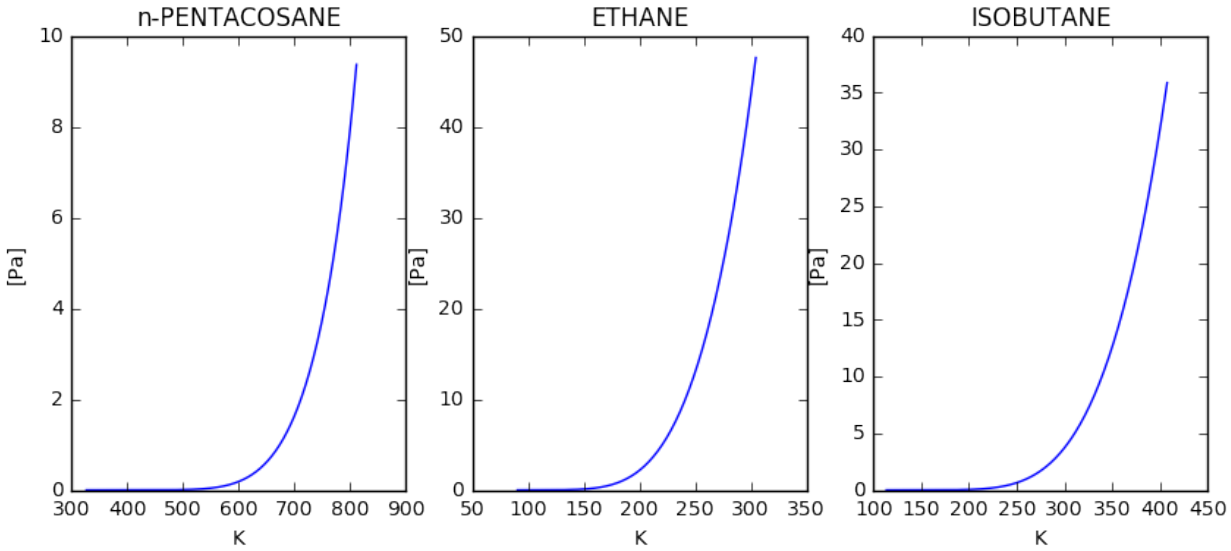


sin embargo como se menciono anteriormente, es posible calcular una propiedad termodinámica para un gran número de sustancias puras y luego realizar las gráficas correspondientes dependiendo de las necesidades de visualización entre otros criterios. Para ejemplificar esto, ahora se tienen 7 sustancias puras y se quiere gráficar la propiedad termodinámica de solo: *n-PENTACOSANE, ETHANE y el ISOBUTANE.*

```
components = ["METHANE", "n-TETRACOSANE", "n-PENTACOSANE", "ETHANE", "ISOBUTANE",
↪"PROPANE", "3-METHYLHEPTANE"]
property_thermodynamics = "Vapour_Pressure"

Vapour_Pressure = thermodynamic_correlations.property_cal(components, property_
↪thermodynamics)
temperature_vapour = thermodynamic_correlations.temperature
```

```
thermodynamic_correlations.multi_graphical(components[2:5], temperature_vapour[2:5],␣
↪Vapour_Pressure[2:5])
```



4.5 16.5 Especificar varias sustancias puras y una temperatura.

Como en el caso anterios, en este ejemplo se espcifican 3 sustancias puras pero con la especificación de un solo valor de temperatura. Esta temperatura será común para las sustancias puras con las que se trabaje por tanto puede darse el caso de que sea una temperatura valida para algunas sustancias puras mientras que para otras no dependiendo del intervalo de valides de cada correlación termodinámica.

```
dppr_file = "PureFull_mod_properties.xls"

thermodynamic_correlations = pt.Thermodynamic_correlations(dppr_file)

components = ["METHANE", "n-TETRACOSANE", "ISOBUTANE"]
property_thermodynamics = "Vapour_Pressure"
temperature = [180.4]

Vapour_Pressure = thermodynamic_correlations.property_cal(components, property_
↪thermodynamics, temperature)
print("Vapour Pressure = {0} {1}". format(Vapour_Pressure, units[1]))
```

```
------------------------------------------------------------------------
Pure substances with a temperature especific: ['METHANE', 'n-TETRACOSANE', 'ISOBUTANE
↪']
------------------------------------------------------------------------
```

(continues on next page)

(proviene de la página anterior)

```
[180.4]
Temperature_enter = [[180.4], ['180.4 K is a temperature not valid'], [180.4]]
Temperature_invalid = [[], ['180.4 K is a temperature not valid'], []]
Temperature_valid = [array([ 180.4]), array([], dtype=float64), array([ 180.4])]
vapour_Pressure =  [array([ 33.32655377]) array([], dtype=float64) array([ 0.
↪0074373])] (3,)
3
Vapour Pressure = [array([ 33.32655377]) array([], dtype=float64) array([ 0.
↪0074373])] [Pa]
```

en este caso se tiene como resultado un con 2 valores de presión de vapor, uno para METHANE y otro para ISOBUTANE, mientras que se obtiene un array vacio en el caso «de n-TETRACOSANE, puesto que la temperatura de 180 K especificada no se encuentra como valida.

para verificar tanto los valores de las constantes como los valores mínimos y máximos de cada correlación termodinámica para cada una de las sustancias puras que se especifique se utiliza el atributo *component_constans* tal como se muestra a continuación

```
thermodynamic_correlations.component_constans
```

4.6 16.6 Especificar varias sustancias puras y especificar varias temperaturas

En esta opción se puede manipular varias sustancias puras de forma simultanea con la especificación de varios valores de temperaturas, en donde cada valor de temperatura especificado será común para cada sustancia pura, de tal forma que se obtendra valores adecuados para aquellos valores de temperatura que sean validos para cada caso considerado.

```
import numpy as np
import pandas as pd
import pyther as pt
import matplotlib.pyplot as plt
%matplotlib inline
```

```
dppr_file = "PureFull_mod_properties.xls"

thermodynamic_correlations = pt.Thermodynamic_correlations(dppr_file)

#components = ["METHANE", "n-TETRACOSANE", "ISOBUTANE"]
components = ["METHANE", "n-TETRACOSANE", "n-PENTACOSANE", "ETHANE", "ISOBUTANE",
↪"PROPANE", "3-METHYLHEPTANE"]
property_thermodynamics = "Vapour_Pressure"
temperature = [180.4, 181.4, 185.3, 210, 800]

Vapour_Pressure = thermodynamic_correlations.property_cal(components, property_
↪thermodynamics, temperature)
print("Vapour Pressure = {0}". format(Vapour_Pressure))
```

```
-----------------------------------------------------------------------
Pure substances with a temperature especific: ['METHANE', 'n-TETRACOSANE', 'n-
↪PENTACOSANE', 'ETHANE', 'ISOBUTANE', 'PROPANE', '3-METHYLHEPTANE']
-----------------------------------------------------------------------
[180.4, 181.4, 185.3, 210, 800]
```

(continues on next page)

(proviene de la página anterior)

```
Temperature_enter = [[180.4, 181.4, 185.3, '210 K is a temperature not valid', '800 K␣
↪is a temperature not valid'], ['180.4 K is a temperature not valid', '181.4 K is a␣
↪temperature not valid', '185.3 K is a temperature not valid', '210 K is a␣
↪temperature not valid', 800], ['180.4 K is a temperature not valid', '181.4 K is a␣
↪temperature not valid', '185.3 K is a temperature not valid', '210 K is a␣
↪temperature not valid', 800], [180.4, 181.4, 185.3, 210, '800 K is a temperature␣
↪not valid'], [180.4, 181.4, 185.3, 210, '800 K is a temperature not valid'], [180.4,
↪ 181.4, 185.3, 210, '800 K is a temperature not valid'], [180.4, 181.4, 185.3, 210,
↪'800 K is a temperature not valid']]
Temperature_invalid = [['210 K is a temperature not valid', '800 K is a temperature␣
↪not valid'], ['180.4 K is a temperature not valid', '181.4 K is a temperature not␣
↪valid', '185.3 K is a temperature not valid', '210 K is a temperature not valid'], [
↪'180.4 K is a temperature not valid', '181.4 K is a temperature not valid', '185.3␣
↪K is a temperature not valid', '210 K is a temperature not valid'], ['800 K is a␣
↪temperature not valid'], ['800 K is a temperature not valid'], ['800 K is a␣
↪temperature not valid'], ['800 K is a temperature not valid']]
Temperature_valid = [array([ 180.4,  181.4,  185.3]), array([800]), array([800]),␣
↪array([ 180.4,  181.4,  185.3,  210. ]), array([ 180.4,  181.4,  185.3,  210. ]),␣
↪array([ 180.4,  181.4,  185.3,  210. ]), array([ 180.4,  181.4,  185.3,  210. ])]
7
Vapour Pressure = [array([ 33.32655377,  34.43422601,  39.01608023]) array([ 9.
↪23391967])
 array([ 7.9130031])
 array([ 0.80394112,  0.85063572,  1.05335836,  3.33810867])
 array([ 0.0074373 ,  0.00816353,  0.01160766,  0.07565701])
 array([ 0.05189654,  0.05605831,  0.07505225,  0.35872729])
 array([  2.09878094e-07,   2.50494222e-07,   4.89039104e-07,
         1.75089920e-05])]
```

como se muestra en los resultados anteriores, se comienza a complicar la manipulación de los datos conforme incrementa el número de sustancias puras y temperaturas involucradas en el analisis, por tal motivo conviene utilizar las bondades de librerías especializadas para el procesamiento de datos como *Pandas* para obtener resultados más eficientes.

El método *data_temperature(components, temperature, Vapour_Pressure, temp_enter)* presenta un DataFrame con los resultados obtenidos luego de calcular la propiedad termodinámica indicada, señalan que para las temperaturas invalidas en el intervalo de aplicación de la correlación termodinámica, el resultado será *NaN*, tal como se muestra con el ejemplo a continuación.

```
temp_enter = thermodynamic_correlations.temperature_enter
thermodynamic_correlations.data_temperature(components, temperature, Vapour_Pressure,␣
↪temp_enter)
```

4.7 16.7 Trabajo futuro

- Actualmente PyTher se encuentra implementando la opción de multiples propiedades termodinámicas de forma simultanea para el caso de multiples sustancias puras con multiples opciones de temepratura.
- Dar soporte a la manipulación de bases de datos por parte de usuarios para agregar, modificar, eliminar, renombrar sustancias puras y/o correlaciones termodinámicas.

4.8 16.8 Referencias

Numpy

CAPÍTULO 5

5. Modelos y parametros para sustancia puras

En todos los casos se realiza el calculo (o el reclaculo) del parametro *rk* de la ecuación RKPR.

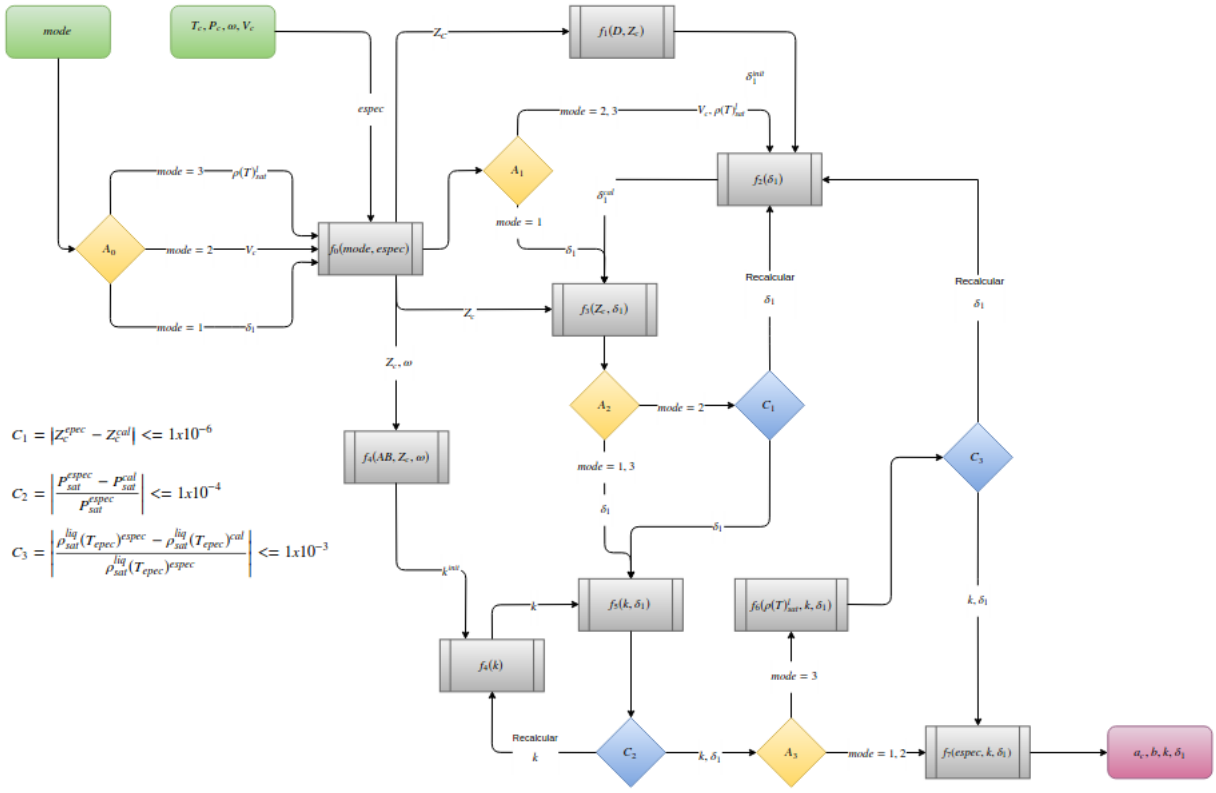
Se tiene la posibilidad de especificar un par (presión de vapor, temperatura) para realizar los calculos, en caso contrario de no hacer esta especificación, se toma por defecto el valor de la presión de vapor correspondiente a una temperatura reducida *Tr = 0.7* junto con el correspondiente valor del factor acentrico según cada sustancia pura.

- Temperatura critica
- Presión critica
- factor acentrico Omega
- Volumen critico

En esta sección se presenta la forma de obtener los parámetros correspondientes para las ecuaciones de estado SRK, PR y RKPR con las diferentes especificaciones que se pueden realizar.

Para el caso de las ecuaciones de estado SRK y PR se tienen como parámetros a las constantes a_c y b. Estos parámetros se determinan a partir de las constantes criticas para cada sustancia pura.

En el caso de la ecuación de estado RKPR se tiene 2 parámetros adicionales a los ya mencionados ac y b. Son los parámetros d1 y k, que se muestran en las siguientes ecuaciones:



importar las linrerías requeridas, en este caso se trata de las librerías numpy, pandas junto con pyther

```
import numpy as np
import pandas as pd
import pyther as pt
```

En los ejemplos siguientes se utilizan los datos termodísicos de la base de datos DPPR. Para el caso se tiene como especificación la ecuación de estado RKPR y las constantes criticas para el componente 3-METHYLHEPTANE a continuación.

```
properties_data = pt.Data_parse()

dppr_file = "PureFull.xls"
component = "3-METHYLHEPTANE"

NMODEL = "RKPR"
ICALC = "constants_eps"

properties_component = properties_data.selec_component(dppr_file, component)
pt.print_properties_component(component, properties_component)
dinputs = np.array([properties_component[1]['Tc'], properties_component[1]['Pc'],
                    properties_component[1]['Omega'], properties_component[1]['Vc']])

component_eos = pt.models_eos_cal(NMODEL, ICALC, dinputs)

#ac = component_eos[0]
print(component_eos)
```

```
Component = 3-METHYLHEPTANE
Acentric_factor = 0.3718
Critical_Temperature = 563.67 K
Critical_Pressure = 25.127 Bar
Critical_Volume = 0.464 cm3/mol
Compressibility_factor_Z = 0.252
del1ini = 6.038268203938681
Zc = 0.24877058378575795
The NMODEL is eos_RKPR and method ICALC is constants_eps
params = [ac, b, rm, del1]
[46.430578671675555, 0.10935115096084358, 2.5860921512475117, 6.0431253541984447]
```

De esta forma se observa el calculo simple de los parámetros para la sustancia pura 3-METHYLHEPTANE_RKPR

A continuación se realiza el mismo tipo de calculo pero tomando una serie de 9 sustancias puras, que se pueden extender facilmente a n sustancias, para obtener sus parámetros de nuevo con la ecuación de estado RKPR.

```
properties_data = pt.Data_parse()

dppr_file = "PureFull.xls"
components = ["ISOBUTANE", "CARBON DIOXIDE", 'METHANE', "ETHANE", "3-METHYLHEPTANE",
↪"n-PENTACOSANE",
              "NAPHTHALENE", "m-ETHYLTOLUENE", "2-METHYL-1-HEXENE"]

NMODEL = "RKPR"
ICALC = "constants_eps"
component_eos_list = np.zeros( (len(components),4) )

for index, component in enumerate(components):

    properties_component = properties_data.selec_component(dppr_file, component)
    pt.print_properties_component(component, properties_component)
    dinputs = np.array([properties_component[1]['Tc'], properties_component[1]['Pc'],
                        properties_component[1]['Omega'], properties_component[1]['Vc
↪']])

    component_eos = pt.models_eos_cal(NMODEL, ICALC, dinputs)
    component_eos_list[index] = component_eos


components_table = pd.DataFrame(component_eos_list, index=components, columns=['ac',
↪'b', 'rm', 'del1'])

print(components_table)
```

```
Component = ISOBUTANE
Acentric_factor = 0.18080000000000002
Critical_Temperature = 408.14 K
Critical_Pressure = 36.003 Bar
Critical_Volume = 0.2627 cm3/mol
Compressibility_factor_Z = 0.28200000000000003
del1ini = 3.9722378008963446
Zc = 0.27871152548257544
The NMODEL is eos_RKPR and method ICALC is constants_eps
params = [ac, b, rm, del1]
Component = CARBON DIOXIDE
Acentric_factor = 0.22360000000000002
```

(continues on next page)

(proviene de la página anterior)

```
Critical_Temperature = 304.21 K
Critical_Pressure = 72.865 Bar
Critical_Volume = 0.094 cm3/mol
Compressibility_factor_Z = 0.274
del1ini = 4.462908059336361
Zc = 0.2707937660977233
The NMODEL is eos_RKPR and method ICALC is constants_eps
params = [ac, b, rm, del1]
Component = METHANE
Acentric_factor = 0.0115
Critical_Temperature = 190.564 K
Critical_Pressure = 45.389 Bar
Critical_Volume = 0.09860000000000001 cm3/mol
Compressibility_factor_Z = 0.28600000000000003
del1ini = 3.7519407434981633
Zc = 0.2824567739174239
The NMODEL is eos_RKPR and method ICALC is constants_eps
params = [ac, b, rm, del1]
Component = ETHANE
Acentric_factor = 0.0995
Critical_Temperature = 305.32 K
Critical_Pressure = 48.083 Bar
Critical_Volume = 0.14550000000000002 cm3/mol
Compressibility_factor_Z = 0.279
del1ini = 4.161423913263858
Zc = 0.2755907402334964
The NMODEL is eos_RKPR and method ICALC is constants_eps
params = [ac, b, rm, del1]
Component = 3-METHYLHEPTANE
Acentric_factor = 0.3718
Critical_Temperature = 563.67 K
Critical_Pressure = 25.127 Bar
Critical_Volume = 0.464 cm3/mol
Compressibility_factor_Z = 0.252
del1ini = 6.038268203938681
Zc = 0.24877058378575795
The NMODEL is eos_RKPR and method ICALC is constants_eps
params = [ac, b, rm, del1]
Component = n-PENTACOSANE
Acentric_factor = 1.1053
Critical_Temperature = 812 K
Critical_Pressure = 9.376 Bar
Critical_Volume = 1.46 cm3/mol
Compressibility_factor_Z = 0.20500000000000002
del1ini = 10.600246415857843
Zc = 0.20275882073834256
The NMODEL is eos_RKPR and method ICALC is constants_eps
params = [ac, b, rm, del1]
Component = NAPHTHALENE
Acentric_factor = 0.3022
Critical_Temperature = 748.35 K
Critical_Pressure = 39.98 Bar
Critical_Volume = 0.41300000000000003 cm3/mol
Compressibility_factor_Z = 0.269
del1ini = 4.8204311891035925
Zc = 0.2653709654843225
The NMODEL is eos_RKPR and method ICALC is constants_eps
```

(continues on next page)

(proviene de la página anterior)

```
params = [ac, b, rm, del1]
Component = m-ETHYLTOLUENE
Acentric_factor = 0.3226
Critical_Temperature = 637.15 K
Critical_Pressure = 28.029 Bar
Critical_Volume = 0.49 cm3/mol
Compressibility_factor_Z = 0.263
del1ini = 5.246526144851435
Zc = 0.2592551086535563
The NMODEL is eos_RKPR and method ICALC is constants_eps
params = [ac, b, rm, del1]
Component = 2-METHYL-1-HEXENE
Acentric_factor = 0.3094
Critical_Temperature = 538 K
Critical_Pressure = 28.325 Bar
Critical_Volume = 0.398 cm3/mol
Compressibility_factor_Z = 0.255
del1ini = 5.784189965441039
Zc = 0.2520206003977051
The NMODEL is eos_RKPR and method ICALC is constants_eps
params = [ac, b, rm, del1]
                          ac         b         rm        del1
ISOBUTANE           15.743219  0.064343  2.205509   4.000470
CARBON DIOXIDE       4.409808  0.022801  2.280728   4.492210
METHANE              2.696405  0.024259  1.282178   3.777713
ETHANE               6.649597  0.035503  1.673541   4.190762
3-METHYLHEPTANE     46.430579  0.109351  2.586092   6.043125
n-PENTACOSANE      289.947431  0.320522  4.581358  10.628260
NAPHTHALENE          49.312554  0.099495  2.591582   4.847168
m-ETHYLTOLUENE       51.786960  0.117115  2.565531   5.267361
2-METHYL-1-HEXENE    37.214555  0.094214  2.338038   5.794610
```

Como se observa, los resultados obtenidos son organizados en un DataFrame permitiendo agilizar la manipulación de los datos de una serie de sustancias puras.

```
components_table
```

En el siguiente ejemplo se utiliza la ecuación RKPR pero esta vez con la especificación de la temperatura y densidad de líquido saturado para el CARBON DIOXIDE y de esta forma encontrar el valor del parámetro *delta* que verifica la especificación realizada para la densidad de líquido saturado.

```
properties_data = pt.Data_parse()

dppr_file = "PureFull.xls"
component = "CARBON DIOXIDE"

NMODEL = "RKPR"
ICALC = "density"

properties_component = properties_data.selec_component(dppr_file, component)
pt.print_properties_component(component, properties_component)
#dinputs = np.array([properties_component[1]['Tc'], properties_component[1]['Pc'],
#                    properties_component[1]['Omega'], properties_component[1]['Vc']])

T_especific = 270.0
RHOLSat_esp = 21.4626
```

(continues on next page)

(proviene de la página anterior)

```
# valor initial of delta_1
delta_1 = 1.5

dinputs = np.array([properties_component[1]['Tc'], properties_component[1]['Pc'],
                    properties_component[1]['Omega'], delta_1, T_especific, RHOLSat_
↪esp])


component_eos = pt.models_eos_cal(NMODEL, ICALC, dinputs)

print(component_eos)
```

```
Component = CARBON DIOXIDE
Acentric_factor = 0.22360000000000002
Critical_Temperature = 304.21 K
Critical_Pressure = 72.865 Bar
Critical_Volume = 0.094 cm3/mol
Compressibility_factor_Z = 0.274
The NMODEL is eos_RKPR and method ICALC is density
The parameter delta1(rho,T) = [ 2.65756708]
[ 2.65756708]
```

CAPÍTULO 6

6. Cálculo del Volumen(P,T,n)

6.1 6.1 Introduction

En esta sección se presenta un ejemplo numérico para calcular propiedades termodinámicas y volumetricas utilizando ecuaciones de estado. Para comenzar se desarrolla el procedimiento que permite determinar el volumen de un sistema cuando se especifica la presión **P**, la temperatura **T** y el número de moles **n**, cuya interdependencia entre estás variables es como se muestra en la ecuación (1)

$$P = P(T, V, n) \tag{6.1}$$

La ecuación de presión tradicionalmente se conoce como una ecuación de estado explicita en el termino de la presión, la cual se relaciona con la función residual de **Helmholtz** $A^r(T, V, n)$ que es una función que depende de las variables de estado de una mezcla (T, V, n) menos el equivalente de las mismas variables de estado como una mezcla (T, V, n) de gas ideal, tal como se muestra a continuación en la ecuación (2)

$$A^r(T, V, n) = -\int_{\infty}^{V} \left(P - \frac{nRT}{V} \right) dV \tag{6.2}$$

y recordando la relación simple del número de moles de la mezcla multicomponente en la ecuación (3)

$$n = \sum_i n_i \tag{6.3}$$

Al reorganizar la ecuación (2), se obtiene la tradicional ecuación de estado explicita en la presión como una contribución del negativo de la derivada parcial de la función de Helmohtlz con respecto al volumen **V**, la temperatura **T** y número de moles **n** constante, más el termino de la ecuación de gas ideal, tal como se muestra en la ecuación (4).

$$P = -\left(\frac{\partial A^r(T, V, n)}{\partial V} \right)_{T,V} + \frac{nRT}{V} \tag{6.4}$$

definiciendo la variable **F** como la función de Herlmhotlz residual redicida, como en la ecuación (5)

$$F = \frac{A^r(T, V, n)}{RT} \tag{6.5}$$

se obtiene una función de Helmohtlz residual reducida cuya funcionalidad es como se muestra en la ecuación (6)

$$F = F(n, T, V, B, D) \tag{6.6}$$

es decir, que de esta forma, la ecuación de estado explicita en la presión del sistema se puede reescribir como en la ecuación (7)

$$P = -RT\left(\frac{\partial F}{\partial V}\right)_{T,V} + \frac{nRT}{V} \tag{6.7}$$

ahora es posible obtener las expresiones de las derivadas parciales de la presión con respecto a cada una de las variables del sistema

- Derivada parcial de la presión **P** con respecto al volumen **V**, ecuación (8)

$$\left(\frac{\partial P}{\partial V}\right)_{T,n} = -RT\left(\frac{\partial^2 F}{\partial V^2}\right)_{T,V} - \frac{nRT}{V^2} \tag{6.8}$$

- Derivada parcial de la presión *P** con respecto a la temperatura **T**, ecuación (9)

$$\left(\frac{\partial P}{\partial T}\right)_{V,n} = -RT\left(\frac{\partial^2 F}{\partial T \partial V}\right)_{n} - \frac{P}{T} \tag{6.9}$$

- Derivada parcial de la presión *P** con respecto al número de moles de cada componente n_i, ecuación (10)

$$\left(\frac{\partial P}{\partial n_i}\right)_{T,V} = -RT\left(\frac{\partial^2 F}{\partial V \partial n_i}\right)_{T} + \frac{RT}{V} \tag{6.10}$$

- Derivada parcial del volumen **V** con respecto al número de moles de cada componente n_i, ecución (11)

$$\left(\frac{\partial V}{\partial n_i}\right)_{T,P} = \frac{\left(\frac{\partial P}{\partial n_i}\right)_{T,V}}{\left(\frac{\partial P}{\partial V}\right)_{T,n}} \tag{6.11}$$

6.2 6.2 Método de Solución

Luego de presentar las ecuaciones necesarias en la sección 4.1, ahora se formula la función objetivo con la cual se implementa un método numérico para encoñtran los ceros de una función no lineal, por tanto al especificar la presión P_{esp}, temperatura T y número de moles del sistema n, se quiere encontrar el volumen de la mezcla que cumpla con un valor de la presión determinado usando una ecuación de estado P_{cal}. De esta forma, se plantea la función objetivo $h(T, V, n)$ que se muestra en la ecuación (12)

Función objetivo que se formula para este caso:

$$h(T, V, n) = ln(P_{esp}) - ln(P_{cal}) \tag{6.12}$$

y su primera derivada analítica, se muestra en la ecuación (13)

$$dh(T, V, n) = -\frac{dln(P_{cal})}{dV} \tag{6.13}$$

por tanto, para efectos didacticos se implementa el método de Newton en una sola variable, en este caso para determinar el Volumen **V**, tal como se muestra en la ecuación (14)

$$V^{k+1} = V^k - s^k \frac{h^k}{dh^k} \tag{6.14}$$

por defecto el parametro es s tiene un valor de la unidad, $s = 1$

y la ecuación (12), es resuelta con una tolerancia de error como se muestra en la ecuación (15)

$$error = abs(h(T, V, n)) = abs(ln(P_{esp}) - ln(P_{cal})) \tag{6.15}$$

como ya se había mencionado anteriormente, la presión del sistema está dada por la suma de dos terminos, el primero corresponde a la función de Helmhotlz y el segundo a la parte de la ecuación de gas idea.

Nota: El cálculo de la función de Helmholtz que se muestra a continuación, escrita de la forma que tiene la ecuación (16), es independiente del modelo termodinámico que se utilice: **ecuación de estado**, además de permitir la manipulación modular del sistema de ecauciones.

Función de la energía de **Helmholtz**

$$F = F(n, T, V, B, D) = -ng(V, B) - \frac{D(T)}{T} f(V, B) \tag{6.16}$$

Donde los terminos (g) y (f)de la ecuación (16), se muestran en las ecuaciones (17) y (18), respectivamente

$$g = ln(1 - B/V) = ln(V - B) - ln(V) \tag{6.17}$$

$$f = \frac{1}{RB(\delta_1 - \delta_2)} ln \frac{(1 + \delta_1 B/V)}{(1 + \delta_2 B/V)} = \frac{1}{RB(\delta_1 - \delta_2)} ln \frac{V + \delta_1 B}{V + \delta_2 B} \tag{6.18}$$

6.3 6.3 Derivadas Parciales

Anteriormente se comentó, el enfoque modular de *Michelsen & Mollerup* permite estructurar los diferentes elementos necesarios para el cálculo de propiedades termidinámicas en forma de bloques, por tanto se presenta la forma modular que resultan para las primeras y segundas derivadas parciales de la función de la energía de Helmholtz. Al iniciar, se presenta en la ecuación (19) la primera derivada parcial de la función F

Primera derivada parcial de F con respecto al volumen V, con T y n constantes

$$\left(\frac{\partial F}{\partial V}\right)_{T,n} = F_V = -ng_V - \frac{D(T)}{T} f_V \tag{6.19}$$

de igual forma, en los terminos :math: *g_V* y :math: *f_V*, se muestran en las ecuaciones (20) y (21), respectivamente

$$g_V = \frac{1}{V - B} - \frac{1}{V} = \frac{B}{V(V - B)} \tag{6.20}$$

$$f_V = \frac{1}{RB(\delta_1 - \delta_2)} \left(\frac{1}{V + \delta_1 B} - \frac{1}{V - \delta_2 B}\right) \tag{6.21}$$

la ecuación (21) tiene una forma alternativa más compacta como la que se muestra en la ecuación (22)

$$f_V = -\frac{1}{R(BV + \delta_1 B)(V + \delta_2 B)} \tag{6.22}$$

siguiendo el mismo procedimiento, se obtiene la segunda derivada parcial de la función F con respecto al volumen y esta, se muestra en la ecuación (23)

Segunda derivada parcial de F con respecto al volumen V, con T y n constantes

$$\left(\frac{\partial^2 F}{\partial V^2}\right)_{T,n} = F_{VV} = -ng_{VV} - \frac{D}{T} f_{VV} \tag{6.23}$$

como en el caso anterior, en los terminos g_{VV} y f_{VV}, se muestran en las ecuaciones (24) y (25), respectivamente

$$g_{VV} = -\frac{1}{(V-B)^2} + \frac{1}{V^2} \tag{6.24}$$

$$f_{VV} = \frac{1}{RB(\sigma_1 - \sigma_2))}\left(-\frac{1}{(V+\sigma_1 B)} + \frac{1}{(V+\sigma_2 B)}\right) \tag{6.25}$$

En las ecuaciones anteriores de la función, primera derivada y segunda derivada de Helmhotlz aparecen los parametros **D** y **B** que se expresan como se muestran en las ecuaciones (26) y (28), respectivamente

$$D(T) = \sum_i n_i \sum_j n_j a_{ij}(T) = \frac{1}{2}\sum_i n_i D_i \tag{6.26}$$

donde D_i es la derivada de D con respecto al número de moles n_i de la mezcla, que tiene la forma de la ecuación (27)

Primera derivada parciale del parámetro D con respecto a n_i

$$D_i = 2\sum_j n_j a_{ij} \tag{6.27}$$

en el caso del parámetro B, la ecuación (28) presenta la forma de realizar su cálculo

Parametro **B**

$$nB = \sum_i n_i \sum_j n_j b_{ij} \tag{6.28}$$

Para el caso de un componente puro en el sistema, el parametro B (lij = 0) se calcula como se muestra en la ecuación (29)

$$B = n_i b_{ii} \tag{6.29}$$

y para el caso de una mezcla, la ecuación (29) se reescribe en la orma de la ecuación (30)

$$B = \sum_i n_i b_{ii} \tag{6.30}$$

Las derivadas parciales del parametro B con respecto al número de moles ni, se obtiene al resolver las ecuaciones (31) y (32)

$$B + nB_i = 2\sum_j n_j b_{ij} \tag{6.31}$$

$$B_j + B_i + nB_{ij} = 2b_{ij} \tag{6.32}$$

Resolviendo el sistema de las ecuaciones (31) y (32), se obtiene las ecuaciones (33) y (34)

$$B_i = \frac{2\sum_j n_j b_{ij} - B}{n} \tag{6.33}$$

$$B_{ij} = \frac{2b_{ij} - B_i - B_j}{n} \tag{6.34}$$

De esta manera, ya se cuenta con las ecuaciones necesarias para obtener las primeras y segundas derivadas de la función F con respecto al V a P, T y ni constantes.

6.4 6.4 Ecuación de estado

Hasta acá se ha presentado la manipulacion básica de la función de Herlmhotlz que partiendo de una expresión explicita en la presión como una ecuación de estado, el sistema de ecuaciones se pueda resolver una vez se especificá la presión P, la temperatura T y número de moles n y proceder a la determinación del valor del volumen V correspondiente para un modelo termmodinámico y componentes prestablecidos.

Para este caso se utiliza el modelo de: **Ecuación de estado cúbica**, cuya forma básica se muestra en la ecuación (35)

$$P = \frac{RT}{v-b} - \frac{a(T)}{v(v+b)+b(v-b)} \tag{6.35}$$

en la cual, se requieren los parámetros que se presentan en las ecuaciones (36)-(39)

$$a(T) = a\alpha(T_r, w) \tag{6.36}$$

$$\alpha(T_r, w) = \left(1 + m\left(1 - \sqrt{\left(\frac{T}{T_c}\right)}\right)\right)^2 \tag{6.37}$$

$$a = \Omega_a \frac{R^2 T_c^2}{P_c} \tag{6.38}$$

$$b_c = \Omega_b \frac{RT_c}{Pc} \tag{6.39}$$

estos parámetros, se relacionan con los valores caracteristicos para los modelos **Soave-Redlich-Kwong (SRK)** y **Peng-Robinson (PR)**, en las ecuaciones ()-()

Tabla 1. Parámetros de ecuaciones de estado utilizadas

Soave-Redlich-Kwong	Peng-Robinson
$\sigma_1 = 1$	$\sigma_1 = 1 + \sqrt{2}4$
$\sigma_2 = 0$	$\sigma_2 = 1 - \sqrt{2}$
$m_{SRK} = 0{,}480 + 1{,}574w - 0{,}175w^2$	$m_{PR} = 0{,}37464 + 1{,}5$
$\Omega_{a,SRK} = 0{,}077796070$	$\Omega_{a,PR} = 0{,}45723553$
$\Omega_{b,SRK} = 0{,}086640$	$\Omega_{b,PR} = 0{,}077796070$

6.5 6.5 Resultados

A continuación se presenta un ejemplo numérico del cálculo del volumen de una mezcla multicomponente con las especificaciones de presión P, temperatura T y número de moles n, que se muestran en la Tabla 2.

Tabla 2. Comparación de resultados entre PyTher y GPEC

P = 800.0 Bar T = 368.0 K		
C1 = 0.8224 moles, C3 = 0.0859 moles, C24 = 0.0917 moles		
Variable	PyTherm	GPEC
V	0.097188024166321052	0.09712098988665994

En la tabla 2, se puede observar que para el ejemplo presentado en este documento el procedimiento implementado en Python se puede considerar validado.

Nota: Se requiere probar el código implementado con más casos que involucren un mayor número de componentes, tipos de componentes y otras especificaciones de presión, temperatura y número de moles.

6.6 4.6 Conclusiones

- Se presentó el procedimiento básico para calcular el volumen de una mezcla multicomponente usando el enfoque modular de Michelsen & Mollerup con las ecuaciones de estado SRK y PR.
- Se implmento el algoritmo para el cálculo del volumen en el lenguaje de programación Python en la plataforma Jupyter.

6.7 6.7 Referencias

CAPÍTULO 7

7. Propiedades Termodinámicas

En este documento se presenta el cálculo de las propiedades termodiñámicas de la fugacidad, entalpía y entropía para el caso de un componente puro y una mezcla de C componentes a una presión P, temperatura T, Volumen V y número de moles N utilizando ecuaciones de estado como **Soave-Kwong (SRK)**[1] y **Peng-Robinson (PR)**[1] y las reglas de mezclado de **Van Der Waals (VDW)**[1] siguiendo el enfoque modular presentado por Michelsen and Mollerup[1].

Advertencia: Falta incluir las propiedades termodinámicas entalpía y entropía.

Nota: Falta incluir más modelos de ecuaciones de estado y reglas de mezclado. En el caso de RKPR falta incluir ejemplos en la documentación.

Para desarrollar el trabajo de este documento se utiliza el lenguaje de programación **Python**[2] y la documentación del mismo se desarrolla con la librería **Sphinx 1.3.1**[3]

Nota: En este proyecto, se desarrolla de forma paralela la documentación utilizando la tecnología IPython notebook - Jupyter[4].

7.1 7.1 Implementación básica

De esta forma, la parte inicial del código en el lenguaje de programación **Python**, corresponde a la importación de la librería **Numpy** la cual aporta un tipo de datos denominado **array** que facilita la manipulación de la información para realizar cálculos con **Python**.

[1] Michael L. Michelsen and Jorgen M. Mollerup. Thermodynamics Models: Fundamentals & Computacional aspects. Denmark. Second Edition. 2007.
[2] Python web: https://www.python.org/
[3] Sphinx web: http://sphinx-doc.org/
[4] Jupyter web: https://jupyter.org/

Nota: Se importa la libreria numpy con el alias **np**

Luego se continua con la definición de la clase **Helmholtz():** y la inicialización de la misma, con la lectura de los parametros **eq, w, Tc, Pc, Tr, R** en el método «constructor» **__init__** de la clase, señalando que el parametro **self** no es una palabra reservada del lenguaje Python pero es una convención ampliamente utilizada por la comunidad de usuarios y desarrolladores de código Python bajo el paradigma de programación orientada a objetos:

```
import numpy as np
from scipy import optimize

class Thermophysical_Properties():

    def __init__(self, eq, w, Tc, Pc, Tr, R):
        """
        eq = Ecuación de estado (SRK = 1) (PR = 2)
        w = factor acentrico
        Tc = temperatura critica del componente i
        Pc = presión critica del componente i
        Tr = temperatura reducida del componente i
        R = costante universal de los gases 0.08314472 [=] bar.l/(mol.K)
        """

        self.eq = eq
        self.w = w
        self.Tc = Tc
        self.Pc = Pc
        self.Tr = Tr
        self.R = R
        if self.eq == 1:
            # Soave-Redlich-Kwong (SRK)
            self.s1, self.s2 = 1, 2
            self.m = 0.480 + 1.574 * self.w - 0.175 * self.w ** 2
            self.ac = 0.077796070 * self.R ** 2, self.Tc ** 2 / self.Pc
            self.bc = 0.086640 * self.R * self.Tc / self.Pc
        elif self.eq == 2:
            # Peng-Robinson (PR)
            self.s1, self.s2 = 1 + 2 ** 0.5, 1 - (2 ** 0.5)
            self.m = 0.37464 + 1.54226 * self.w - 0.26992 * self.w ** 2
            self.ac = 0.45723553 * self.R ** 2 * self.Tc ** 2 / self.Pc
            self.bc = 0.077796070 * self.R * self.Tc / self.Pc
            #Martín Cismondí
            #self.ac = np.array([2.4959, 2.4959, 208.4949])
            #self.bc = np.array([0.026802, 0.056313, 0.530667])
            #self.m = np.array([0.392414, 0.603252, 1.716810])
        else:
            print ("Che boludo... Modelo no valido, intentaló de nuevo !!! ")
```

Relación simple del número de moles de la mezcla multicomponente

$$n = \sum_i n_i \tag{7.1}$$

$$D(T) = \sum_i n_i \sum_j n_j a_{ij}(T) = \frac{1}{2} \sum_i n_i D_i \tag{7.2}$$

Donde D_i es la derivada de D con respecto al número de moles n de la mezcla.

a continuación se presentan las primeras derivadas parciales de la función D con respecto a las variables del sistema

$$D_i = 2\sum_j n_j a_{ij} \tag{7.3}$$

$$D_{iT} = 2\sum_j n_j \frac{\partial aij}{\partial T} \tag{7.4}$$

$$D_{ij} = 2a_{ij} \tag{7.5}$$

$$D_T = \frac{1}{2}\sum_i n_i D_{iT} \tag{7.6}$$

$$D_{TT} = \sum_i n_i \sum_j n_j \frac{\partial^2 a_{ij}}{\partial T^2} \tag{7.7}$$

Para determinar el valor del parametro **D** y continuar con el algoritmo se utiliza el siguiente bloque de código en lenguaje de programación Python:

```
def parametro_D(self):
    if self.nC == 1:
        self.D = self.ni ** 2 * self.a_ii
        self.Di = 2 * self.ni * self.a_ii
    elif self.nC > 1:
        di = np.ones((len(self.ni), len(self.ni)))
        self.Di = np.ones((len(self.ni)))
        self.D = np.ones((len(self.ni)))
        for i in range(self.nC):
            for j in range(self.nC):
                di[i, j] = self.ni[j] * self.aij[i, j]
                self.Di[i] = 2 * np.sum(di[i, :])
        self.D = 0.5 * np.sum(ni * self.Di)

    return self.D
```

$$nB = \sum_i n_i \sum_j n_j b_{ij} \tag{7.8}$$

Para el caso de un componente puro en el sistema, el parametro B (lij = 0) se calcula como:

$$B = n_i b_{ii} \tag{7.9}$$

y para el caso de una mezcla:

$$B = \sum_i n_i b_{ii} \tag{7.10}$$

Las derivadas parciales del parametro B con respecto al número de moles, se obtiene de la siguiente forma:

$$B + nB_i = 2\sum_j n_j b_{ij} \tag{7.11}$$

$$B_j + B_i + nB_{ij} = 2b_{ij} \tag{7.12}$$

Resolviendo el sistema de las ecuaciones (9) y (10) se obtiene:

$$B_i = \frac{2\sum_j n_j b_{ij} - B}{n} \tag{7.13}$$

$$B_{ij} = \frac{2b_{ij} - B_i - B_j}{n} \tag{7.14}$$

Para determinar el valor del parametro **B** y continuar con el algoritmo se utiliza el siguiente bloque de código en lenguaje de programación Python:

```
def parametro_B(self):
    if self.nC == 1:
        self.B = self.ni * self.b_ii
    elif self.nC > 1:
        self.aux = np.zeros((len(self.ni)))
        for i in range(self.nC):
            for j in range(self.nC):
                self.aux[i] = self.aux[i] + self.ni[j] * self.bij[i, j]

        self.B = np.sum(self.ni * self.b_ii)

    return self.B
```

La presión **P** del sistema se determina por medio de la ecuación de estado que se eliga de acuerdo a las opciones inicialmente planteadas:

```
def presion(self):
    '''
    Con el metodo presion(), se calcula la Presión P(T, V, N) del sistema
    para una temperatura T, cantidad de moles N y un volumen V
    R = Constante universal de los gases
    nT = Número total de moles en el sistema
    Pcal = Presión calculada con la ecuación de estado
    Arv = Primera derivada parcial de la energía de Helmholz con respecto al
    volumen V, a T y N constantes
    '''
    self.gv = self.R * self.B / (self.V * (self.V - self.B))
    self.fv = - 1 / ((self.V + self.s1 * self.B) * (self.V + self.s2 * self.B))
    self.ArV = -self.nT * self.gv * self.T - self.D * self.fv
    self.Pcal = self.nT * self.R * self.T / self.V - self.ArV
    return self.Pcal
```

Se requiere el calculo de la primera derivadad de la presión con respecto al volumen a temperatura y número de moles constantes:

```
def dP_dV(self):
    self.dPdV = -self.ArV2 - self.R * self.T * self.nT / self.V ** 2
    return self.dPdV
```

Calculo del factor de compresibilidad **Z**:

```
def Z_factor(self, P):
    self.P = P
    self.Z = (self.P * self.V) / (self.nT * self.R * self.T)
    return self.Z
```

Calculo de la presión ideal del sistema:

```
def P_ideal(self, P):
    self.P = P
    self.Pxi = (self.ni * self.P) / self.nT
    return self.Pxi
```

Primera derivada parcial de la energía libre de Helmhotlz reducidad con respecto al volumen a temperatura y número de moles constantes:

```
def dF_dV(self):
    '''
    Primera derivada de F con respecto al volumen Ecu. (68)
    '''
    self.gv = self.R * self.B / (self.V * (self.V - self.B))
    self.fv = - 1 / ((self.V + self.s1 * self.B) * (self.V + self.s2 * self.B))
    self.ArV = -self.nT * self.gv * self.T - self.D * self.fv
    return self.ArV
```

Segunda derivada parcial de la energía libre de Helmhotlz reducidad con respecto al volumen a temperatura y número de moles constantes:

```
def dF_dVV(self):
    '''
    Segunda derivada de F con respecto al volumen Ecu. (74)
    '''
    self.gv2 = self.R * (1 / self.V ** 2 - 1 / (self.V - self.B) ** 2)
    self.fv2 = (- 1 / (self.V + self.s1 * self.B) ** 2 + 1 / (self.V + self.s2 * self.
↪B) ** 2) / self.B / (self.s1 - self.s2)
    self.ArV2 = - self.nT * self.gv2 * self.T - self.D * self.fv2
    return self.ArV2
```

De esta formar se procede a determinar el valor del Volumen **V** para la presión **P**, temperatura **T** y número de moles **N** especificados para el sistema:

```
def volumen_1(self, P):
    '''
    Calculo del volumen V(T,P,n) del fluido a una temperatura T, presión P
    y número de moles totales nT especificados.
    Se utiliza el método de Newton con derivada de la función analitica.
    Pendiente cambiar por una función de Scipy.
    '''
    self.P = P
    self.V = 1.05 * self.B
    lnP = np.log(self.P)
    print "P_esp = ", self.P
    print "V_ini = ", self.V
    Pite = self.presion()
    lnPcal = np.log(Pite)
    #h = self.P - Pite
    h = lnP - lnPcal
    errorEq = abs(h)
    print "ErrorP = ", errorEq
    i = 0
    s = 1.0

    while errorEq > ep:
        self.parametro_D()
        self.parametro_B()
        self.dF_dV()
        self.dF_dVV()
        dPite = self.dP_dV()
        Pite = self.presion()
        lnPcal = np.log(Pite)
        #h = self.P - Pite
        h = lnP - lnPcal
        dh = -dPite
```

(continues on next page)

(proviene de la página anterior)

```
        #print self.nT
        self.V = self.V - s * h / dh
        errorEq = abs(h)
        #print "ErrorP = ", errorEq
        #print "V = ", self.V
        #print "Pite = ", Pite
        i += 1
        if i >= 900:
            pass
            #break
    print "FV = ", dPite

    return self.V
```

Para el cálculo de la función de la energía libre de Helmholtz que se muestra en la ecuación ( ), la cual escrita de esta forma es independiente del modelo termodinámico que se utilice **ecuación de estado**, además de facilitar la manipulación del sistema de ecauciones **modelo** de forma modular.

Función de la energía de Helmholtz

$$F = F(n, T, V, B, D) = -ng(V, B) - \frac{D(T)}{T} f(V, B) \tag{7.15}$$

Donde

$$g = ln(1 - B/V) = ln(V - B) - ln(V) \tag{7.16}$$

$$f = \frac{1}{RB(\delta_1 - \delta_2)} ln \frac{(1 + \delta_1 B/V)}{(1 + \delta_2 B/V)} = \frac{1}{RB(\delta_1 - \delta_2)} ln \frac{V + \delta_1 B}{V + \delta_2 B} \tag{7.17}$$

Calculo de la función de energía F:

```
def funcion_energia_F(self):
    self.g = self.R * np.log(1 - self.B / self.V)
    self.bv = self.B / self.V
    self.f = np.log((self.V + self.s1 * self.B) / (self.V + self.s2 * self.B)) / self.
↪B / (self.s1 - self.s2)
    self.Ar = -self.nT * self.g * self.T - self.D * self.f
    #print (("g = ", self.g))
    #print (("f: ", self.f))
    #print (("Ar: ", self.Ar))
    return self.g, self.f, self.Ar, self.bv
```

Elementos requeridos para calcular las primeras derivadas parciales de la función de energía de Helmholtz $F(n, T, V, B, D)$

$$F_n = -g \tag{7.18}$$

$$F_T = \frac{D(T)}{T^2} f \tag{7.19}$$

$$F_V = -ng_V - \frac{D(T)}{T} f_V \tag{7.20}$$

$$F_B = -ng_B - \frac{D(T)}{T} f_B \tag{7.21}$$

$$F_D = -\frac{f}{T} \tag{7.22}$$

$$g_V = \frac{1}{V-B} - \frac{1}{V} = \frac{B}{V(V-B)} \tag{7.23}$$

$$g_B = -\frac{V}{B} g_V = -\frac{1}{(V-B)} \tag{7.24}$$

$$f_V = \frac{1}{RB(\delta_1 - \delta_2)} \left(\frac{1}{V + \delta_1 B} - \frac{1}{V - \delta_2 B} \right) \tag{7.25}$$

$$f_V = -\frac{1}{R(BV + \delta_1 B)(V + \delta_2 B)} \tag{7.26}$$

$$f_B = -\frac{f + V f_V}{B} \tag{7.27}$$

Primeras derivadas parciales de la función F de Helmhotlz con respecto al número de moles N para temperatura T y volumen V constantes, con respecto a la temperatura para V y N constantes y con respecto al volumen para T y N constantes, respectivamente.

$$\left(\frac{\partial F}{\partial n_i} \right)_{T,V} = F_n + F_B B_i + F_D D_i \tag{7.28}$$

$$\left(\frac{\partial F}{\partial T} \right)_{V,n} = F_T + F_D D_T \tag{7.29}$$

$$\left(\frac{\partial F}{\partial V} \right)_{T,n} = F_V \tag{7.30}$$

Nota: En esl código se muestra solo para la primera derivadas parcial de la función F de Helmhotlz con respecto al número de moles N para temperatura T y volumen V constantes.

calculo de lprimeras derivadas:

```
def primeras_derivadas1(self):

    if nC == 1:
        AUX = self.R * self.T / (self.V - self.B)
        self.fB = -(self.f + self.V * self.fv) / self.B
        self.FFB = self.nT * AUX - self.D * self.fB
        self.Di = 2 * self.nT * self.ac * self.alfa
        self.Bi = self.bc
        self.Arn = -self.g * self.T + self.FFB * self.Bi - self.f * self.Di
    elif nC >= 2:
        # Derivando la ecuación (64) se obtiene la ecuación eq (106)
        self.Bi = np.ones((len(self.ni)))
        for i in range(nC):
            self.Bi[i] = (2 * self.aux[i] - self.B) / self.nT

        AUX = self.R * self.T / (self.V - self.B)
        self.fB = -(self.f + self.V * self.fv) / self.B
        self.FFB = self.nT * AUX - self.D * self.fB
        self.Arn = -self.g * self.T + self.FFB * self.Bi - self.f * self.Di

    print "Bi = ", self.Bi
    print "Di = ", self.Di
    print "fB = ", self.fB
    print "FFB = ", self.FFB
    print "Arn cal = ", self.Arn

    return self.Arn
```

$$ln\hat{\varphi}_i = \left(\frac{\partial F}{\partial n_i}\right)_{T,V} - Z \tag{7.31}$$

Una ve se ha obtenido la primera derivada parcial de la energía libre de Helmholtz, se puede calcular tanto la fugacidad como el coeficiente de fugacidad del sistema:

```
def coeficientes_fugacidad(self):
    self.Z = self.Z_factor(self.P)
    self.lnOi = self.Arn / (self.R * self.T) - np.log(self.Z)
    print "lnOi = ", self.lnOi
    self.Oi = np.exp(self.lnOi)
    print "Oi = ", self.Oi
    return self.Oi
```

$$ln\hat{f}_i = \left(\frac{\partial F}{\partial n_i}\right)_{T,V} - Z + ln(Px_i) \tag{7.32}$$

Calculo de la fugacidad:

```
def fugacidad(self):
    self.Z = self.Z_factor(self.P)
    self.Pxi = self.P_ideal(self.P)
    self.lnFi = self.Arn / (self.R * self.T) - np.log(self.Z) + np.log(self.Pxi)
    self.Fi = np.exp(self.lnFi)
    self.PHILOG = self.Arn / (self.R * self.T) - np.log(self.Z)

    print "Z = ", self.Z
    print "Arn = ", self.Arn
    print "lnFi = ", self.lnFi
    print "Fi = ", self.Fi
    print "PHILOG = ", self.PHILOG

    return self.Fi
```

En el método liquido se accede al cálculo de la **fugacidad** del **fluido** para los parametros y especificaciones determinadas. La fugacidad se guarda en la variable **Fug** que tiene la misma dimensión que el número de componentes nC del sistema:

```
def liquido(self, P):
    self.P = P
    ab = self.parametros(self.ni, self.nT, self.nC, self.V, self.T)
    print (("aij = ", ab[0]))
    print (("bij = ", ab[1]))
    print "................................................................"
    D = self.parametro_D()
    B = self.parametro_B()
    print (("D = ", D))
    print (("B = ", B))
    print "................................................................"
    Vol_1 = self.volumen_1(self.P)
    print (("Vol_1 = ", Vol_1))
    print (("Densidad =", 1 / Vol_1))
    print "................................................................"
    F = self.funcion_energia_F()
    print (("g = ", F[0]))
    print (("f = ", F[1]))
    print (("F = ", F[2]))
    print (("bv = ", F[3]))
    print "................................................................"
```

(continues on next page)

(proviene de la página anterior)

```
    dF = self.primeras_derivadas1()
    print (("dFdni = ", dF[0]))
    print (("dFdT = ", dF[1]))
    print (("dFdV = ", dF[2]))
    print "................................................................."
    Z = self.Z_factor(self.P)
    print "Z =", Z
    Zcal = (self.P * Vol_1) / (self.nT * self.R * self.T)
    print "Zcal =", Zcal
    print "................................................................."
    Pq = self.presion()
    print (("Pcal =", Pq))
    print "................................................................."
    Fug = self.fugacidad()
    #print (("Fug = ", Fug[0]))
    print (("Fug = ", Fug))
    print (("CoeFug = ", Fug / (self.ni * self.P)))
    print (("lnCoeFug = ", np.log(Fug / (self.ni * self.P))))
    print "................................................................."

    return Fug
```

A continucion se muestra la forma en que se ingresan provisonalmente los parametros de inicialización para realizar los calculos. La inicialización corresponde a la especificación del número de componentes **nC**, la temperatura **T** en Kelvin, la presión P en **Bar**, la selección de la ecuación de estado **eq** y la tolerancia para determinar el Volumen **V(P, T, N)** del sistema:

```
#--------------------------- Númuro de componentes -----------------------------
#Número de componentes en el sistema
nC = 3
#---------------------------- Temperatura en K ---------------------------------
# K
T = 299.5
#--------------------------- Presión -----------------------------------------
# Bar
P = 1500.0
#---------------------------- Volumen -----------------------------------------
#--------------- Constante R [=] # bar.l/(mol.K) : 0.08314472-------------------
# bar.l/(mol.K) : 0.08314472
R = 0.08314472
#-------------------------------------------------------------------------------
#-------------------------------------------------------------------------------
# selección de la Ecuación de Estado
# eq = 1, para Ecuación de Estado (SRK)
# eq = 2, para Ecuación de Estado (PR)
eq = 2
#------------------ Criterio de convergencia en línea 215 ---------------------
#------------------ del método def volumen_1(self, P): ------------------------
ep = 1e-6
#-------------------------------------------------------------------------------
#--------------------------- Fugacidad Fluido Puro -----------------------------
#-------------------------------------------------------------------------------
print "................................................................."

# metano - propano - C24
Tcm = np.array([190.56, 369.83, 804.0])
```

(continues on next page)

(proviene de la página anterior)

```
Pcm = np.array([45.99, 41.924, 9.672])
wm = np.array([0.0115, 0.1523, 1.071])
```

Nota: Los parametros de los modelos termodinámicos provisinalmente son escritos en el mismo archivo **.py**, mientras se integra un adminitrador de bases de datos.

Ahora se procede a instanciar la clase **fluido = Helmholtz(eq, w, Tc, Pc, Tr, R)** para luego acceder a los métodos **parametros(ni, nT, nC, V, T)** y **liquido(P)**:

```
#-------------------------------------------------------------------------
# Tempertura reducidad
Tr = T / Tc

nT = np.sum(ni)
print "....................................................................."
fluido = Helmholtz(eq, w, Tc, Pc, Tr, R)
ab = fluido.parametros(ni, nT, nC, V, T)
print ab

flu_1 = fluido.liquido(P)
```

7.2 7.2 Resultados

Mientras se terminan los test para el código implmentado en **Python** para hacerlo de forma programatica, se hace una compración entre los resultados que se obtienen con las rutinas implementadas anteriormente en **FORTRAN** y los obtenidos en esta implmentación en la tabla (1) para un componente puro y en la talba (2) para una mezcla.

Tabla 1. Comparación de resultados entre IPyTherm y GPEC, Macla 1

P = 200.0 Bar T = 368.0 K 1 mol C1		
Variable	PyTherm	GPEC
V	0.14160332	0.141604834257319
g	-0.01744569	-0.01744577009114121
f	6.04150003	6.04143211028481
fB	-29.17898803	-29.1783074191090
FFB	318.78279781	318.778307258157
Arn	-6.67700465	-6.67643301466508
$ln\hat{f}_i$	5.15741367	5.15742167555949

Tabla 2. Comparación de resultados entre IPyTherm y GPEC, Mezcla 2

P = 800.0 Bar T = 368.0 K		
C1 = 0.30 moles, C24 = 0.70 moles		
Variable	PyTherm | GPEC	
Arn		
C1	79.86005173	79.86079
C24	-73.51719121	-73.51722
$ln\hat{f}_i$		
C1	5.74717729	5.74720
C24	1.5816976	1.58170

Tabla 3. Comparación de resultados entre IPyTherm y GPEC, Mezcla 3

P = 800.0 Bar T = 368.0 K		
C1 = 0.8224 moles, C3 = 0.0859 moles, C24 = 0.0917 moles		
Variable	PyTher	GPEC
V	0.097895788494793759	0.09712098988665994
g	-0.12547030006562548	-0.125067142383822
f	6.7115641252706366	6.76716180547646
fB	-19.3589126132	-19.7063420668040
FFB	1635.57161009	1641.91328887125
Ar	-30.818627700503917	-30.9082104588285
Arn		
C1	31.03421463	31.0357268368683
C2	-6.35640646	-6.35637488487487
C24	-95.8172984	-95.8172808890964
$ln\hat{f}_i$		
C1	6.7671523	6.76703848796874
C2	5.5448668	5.54496483049592
C24	2.6217857	2.62114371407445

7.3 7.3 Conclusiones

Se implemento en el lenguaje de programación Python el cálculo de la fugacidad de fluidos puros y mezclas multicomponente siguiendo el enfoque modular de la función de la energía de Helmholtz con ecuaciones de estado **(SRK)** **(PR)** con las reglas de mezclado **(VDW)**.

Al comparar los resultados obtenidos con **IPyTherm 1.0** y **GPEC**, se encuentran concordancia numérica para las variables de los casos de revisión planteados, excepto para el valor de la fugacidad de los componentes de la mezcla 3.

Nota: La diferencia que existe entre el valor de la fugacidad de la mezcla 3 al comparar con los datos de **GPEC**, puede ser debida a errores de transcripción. Pendiente por confirmar.

Este modulo enfocado en el calculo de la fugacidad de fluidos puros y mezclas multicomponente, puede ser integrado para realizar cálculos de fugacidad en sólidos.

7.4 7.4 Referencias

CAPÍTULO 8

8. Equilibrio sólido-fluido para sustancias puras

8.1 Importar las librerías

8.2 Cargar la tabla de datos

```
import scipy as sp
from scipy import optimize
from scipy.optimize import fsolve
import numpy as np
from matplotlib import pyplot
%matplotlib inline
import pandas as pd
from numpy import linalg as LA
from IPython.html import widgets
from IPython.display import display
from IPython.display import clear_output

# encoding: utf-8

from pandas import read_csv
```

```
/home/andres-python/anaconda3/lib/python3.5/site-packages/IPython/html.py:14:␣
↪ShimWarning: The IPython.html package has been deprecated. You should␣
↪import from notebook instead. IPython.html.widgets has moved to ipywidgets.
  "IPython.html.widgets has moved to ipywidgets.", ShimWarning)
```

8.3 Cargar la tabla de datos

```
f = pd.read_excel("PureFull.xls")
f.head()
data2 = pd.DataFrame(f)
data2 = data2.set_index('Name')
data2 = data2.ix[:, 1:12]
Etiquetas = data2.index.get_values()
Etiquetas
```

```
array(['METHANE', 'ETHANE', 'PROPANE', ..., 'TITANIUM', 'PHOSPHORUS',
       'PHOSPHORUS'], dtype=object)
```

```
Componentes_1 = widgets.SelectMultiple(
            description="Component 1",
            options=list(Etiquetas))
display(Componentes_1)
```

```
class Thermophysical_Properties():

    def __init__(self, nameData):
        self.nameData = nameData


    def cargar_Datos(self):
        f = pd.read_excel(self.nameData)
        f.head()
        data2 = pd.DataFrame(f)
        data2 = data2.set_index('Name')
        data2 = data2.ix[:, 1:12]
        self.Etiquetas = data2.index.get_values()

        print("Los datos del archivo: {0}, se han cargado correctamente !!!".
↪format(self.nameData))

        return self.Etiquetas

    def seleccionar_Datos(self):
        Componentes_1 = widgets.SelectMultiple(
            description="Component 1",
            options=list(Etiquetas))
        display(Componentes_1)



    def mostrar_Datos(self):
        print ("Nombre componente: {0}".format(self.Etiquetas))




    def agregar_Datos(self):
        pass

    def borrar_Datos(self):
```

(continues on next page)

(proviene de la página anterior)

```
        pass

    def modificar_Datos(self):
        pass

    def crear_Datos(self):
        pass
```

```
nameData = "PureFull.xls"

propiedades = Thermophysical_Properties(nameData)
propiedades.cargar_Datos()
propiedades.mostrar_Datos()
propiedades.seleccionar_Datos()
```

```
Los datos del archivo: PureFull.xls, se han cargado correctamente !!!
Nombre componente: ['METHANE' 'ETHANE' 'PROPANE' ..., 'TITANIUM' 'PHOSPHORUS'
↪'PHOSPHORUS']
```

```
class System_Conditions():

    def __init__(self, Temperature, Pressure, Volume, Mole_fraction, Model_fluid,␣
↪Model_solid):
        self.Temperature = Temperature
        self.Mole_fraction = Mole_fraction
        pass

    def normalizar(self):
        self.Mole_fraction_normal = Mole_fraction / sum(self.Mole_fraction)
        return self.Mole_fraction_normal

    def convertir(self):
        pass



class Componentes(Thermophysical_Properties, System_Conditions):
    """
    Las variables aux_ se utilizan para presentar de forma más clara y acotada
    las expresiones necesarias en los calculos. Estas, se numeran de acuerdo al orden␣
↪de
    aparición dentro de una clase.

    """

    def __init__(self):
        pass

    def cal_SRK_model(self):
        # Soave-Redlich-Kwong (SRK)
        self.s1, self.s2 = 1, 2
        self.m = 0.480 + 1.574 * self.w - 0.175 * self.w ** 2
        self.ac = 0.077796070 * self.R ** 2, self.Tc ** 2 / self.Pc
        self.bc = 0.086640 * self.R * self.Tc / self.Pc
```

(continues on next page)

(proviene de la página anterior)

```
        return self.m, self.ac, self.bc

    def cal_PR_model(self):
        # Peng-Robinson (PR)
        self.s1, self.s2 = 1 + 2 ** 0.5, 1 - (2 ** 0.5)
        self.m = 0.37464 + 1.54226 * self.w - 0.26992 * self.w ** 2
        self.ac = 0.45723553 * self.R ** 2 * self.Tc ** 2 / self.Pc
        self.bc = 0.077796070 * self.R * self.Tc / self.Pc

        self.alfa = (1 + self.m * (1 - (self.T / self.Tc) ** 0.5)) ** 2
        aux_1 = - (self.m / self.T) * (self.T / self.Tc) ** 0.5
        aux_2 = (self.m * (- (self.T / self.Tc) ** 0.5 + 1) + 1)
        self.dalfadT = aux_1 * aux_2

        aux_3 = 0.5 * self.m ** 2 * (self.T / self.Tc) ** 1.0 / self.T ** 2
        aux_4 = (self.m * (- (self.T / self.Tc) ** 0.5 + 1) + 1) / self.T ** 2
        aux_5 = 0.5 * self.m * (self.T / self.Tc) ** 0.5 * aux_4

        self.d2alfaT2 = aux_3 + aux_5

        self.a_ii = self.ac * self.alfa
        self.b_ii = self.bc
        self.da_iidT = self.ac * self.dalfadT
        d2adT2_puros = self.ac * self.d2alfaT2

        return self.m, self.a_ii, self.b_ii

    def cal_RKPR_model(self):
        pass

    def build_component(self):

        if self.eq == "SRK":
            # Soave-Redlich-Kwong (SRK)
            self.component = self.cal_SRK_model()
        elif self.eq == "PR":
            # Peng-Robinson (PR)
            self.component = self.cal_PR_model()
        elif self.eq == "RKPR":
            # (RKPR)
            #self.component = self.cal_RKPR_model()
            print ("No actualizada, intentalo de nuevo !!! ")
        else:
            print ("Che boludo... Modelo no valido, intentalo de nuevo !!! ")
```

```
Componentes_1 = widgets.SelectMultiple(
    description="Component 1",
    options=list(Etiquetas))

Componentes_2 = widgets.SelectMultiple(
    description="Component 2",
    options=list(Etiquetas))

button = widgets.Button(description="Upload Data")

def cargarDatos(b):
```

(continues on next page)

(proviene de la página anterior)

```
    clear_output()
    print("Component 1: ", Componentes_1.value)
    Nombre = Componentes_1.value
    Propiedades = data2.loc[Nombre]
    Factor_Acentrico_1 = Propiedades[0]
    Temperatura_Critica_1 = Propiedades[1]
    Presion_Critica_1 = Propiedades[2]
    Z_Critico_1 = Propiedades[3]

    #print(Propiedades)
    print ("Acentric Factor = ", Factor_Acentrico_1)
    print ("Critical Temperature = ", Temperatura_Critica_1, "K")
    print ("Critical Pressure = ", Presion_Critica_1, "bar")
    print ("Z_Critical = ", Z_Critico_1, "\n")


    print("Component 2: ", Componentes_2.value)
    Nombre = Componentes_2.value
    Propiedades = data2.loc[Nombre]
    Factor_Acentrico_2 = Propiedades[0]
    Temperatura_Critica_2 = Propiedades[1]
    Presion_Critica_2 = Propiedades[2]
    Z_Critico_2 = Propiedades[3]

    #print(Propiedades)
    print ("Acentric Factor  = ", Factor_Acentrico_2)
    print ("Critical Temperature = ", Temperatura_Critica_2, "K")
    print ("Critical Pressure = ", Presion_Critica_2, "bar")
    print ("Z_Critical = ", Z_Critico_2)

    global TcDato, PcDato, wDato

    TcDato = np.array([Temperatura_Critica_1, Temperatura_Critica_2])
    PcDato = np.array([Presion_Critica_1, Presion_Critica_2])
    wDato = np.array([Factor_Acentrico_1, Factor_Acentrico_2])


button.on_click(cargarDatos)
```

```
class Parameters_BD():

    def __init__(self):
        pass

    def cal_parameters_ij(self):

        if self.nC > 1:
            self.aij = np.ones((len(self.ni), len(self.ni)))
            self.bij = np.ones((len(self.ni), len(self.ni)))
            self.daijdT = np.ones((len(self.ni), len(self.ni)))

            for j in range(self.nC):
                for i in range(self.nC):
                    self.aij[i, j] = (self.a_ii[i] * self.a_ii[j]) ** 0.5
                    self.bij[i, j] = (self.b_ii[i] + self.b_ii[j]) / 2
                    self.bij[i, j] = self.bij[i, j]
```

(continues on next page)

(proviene de la página anterior)

```
                    self.daijdT[i, j] = (self.da_iidT[i] * self.da_iidT[j]) ** 0.5

            for i in range(self.nC):
                for  j in range(self.nC):
                    if i == j:
                        self.aij[i, j] = self.a_ii[i] * (1 - self.kij[i, j])
                        self.daijdT[i, j] = self.da_iidT[i] * (1 - self.kij[i, j])
                    elif i != j:
                        self.aij[i, j] = self.aij[i, j] * (1 - self.kij[i, j])
                        self.daijdT[i, j] = self.daijdT[i, j] * (1 - self.kij[i, j])

        if self.nC == 1:
            return self.a_ii, self.b_ii, self.da_iidT
        else:
            return self.aij, self.bij, self.daijdT

    def cal_parameter_D(self):
        if self.nC == 1:
            self.D = self.ni ** 2 * self.a_ii
            self.Di = 2 * self.ni * self.a_ii
        else:
            di = np.ones((len(self.ni), len(self.ni)))
            self.Di = np.ones((len(self.ni)))
            self.D = np.ones((len(self.ni)))
            for i in range(self.nC):
                for j in range(self.nC):
                    di[i, j] = self.ni[j] * self.aij[i, j]
                    self.Di[i] = 2 * np.sum(di[i, :])
            self.D = 0.5 * np.sum(self.ni * self.Di)

        return self.D

    def cal_parameter_delta_1(self):

        if self.nC == 1:
            self.D1m = np.zeros((len(self.ni)-1))
            self.dD1i = np.ones((len(self.ni)))
            self.dD1ij = np.ones((len(self.ni), len(self.ni)))

            for i in range(self.nC):
                self.D1m = self.D1m + self.ni[i] * self.delta_1[i]

            self.D1m = self.D1m / self.nT

        else:
            self.D1m = np.zeros((len(self.ni)-1))
            self.dD1i = np.ones((len(self.ni)))
            self.dD1ij = np.ones((len(self.ni), len(self.ni)))

            for i in range(self.nC):
                self.D1m = self.D1m + self.ni[i] * self.delta_1[i]

            self.D1m = self.D1m / self.nT

            for i in range(self.nC):
                self.dD1i[i] = (self.delta_1[i] - self.D1m) / self.nT
                for j in range(self.nC):
```

(continues on next page)

(proviene de la página anterior)

```
                    self.dD1ij[i,j] = (2.0 * self.D1m - self.delta_1[i] - self.delta_
↪1[j]) / self.nT ** 2

        return self.D1m, self.dD1i, self.dD1ij

    def cal_parameter_B(self):
        if self.nC == 1:
            self.B = self.ni * self.b_ii
        else:
            self.aux = np.zeros((len(self.ni)))
            for i in range(self.nC):
                for j in range(self.nC):
                    self.aux[i] = self.aux[i] + self.ni[j] * self.bij[i, j]

            self.B = np.sum(self.ni * self.b_ii)
            #print("B = ", self.B)

        return self.B
```

```
class Fugacidad():

    def __init__(self, eq, w, Tc, Pc, Tr, R, ep, ni, nT, nC, V, T, P, kij, lij, delta_
↪1, k, Avsl):
        self.eq = eq
        self.w = w
        self.Tc = Tc
        self.Pc = Pc
        self.Tr = Tr
        self.R = R
        self.ep = ep
        self.ni = ni
        self.nT = nT
        self.nC = nC
        self.V = V
        self.T = T
        self.P = P
        self.kij = kij
        self.lij = lij
        self.delta_1 = delta_1
        self.k = k
        self.Avsl = Avsl

        if self.eq == "SRK":
            # Soave-Redlich-Kwong (SRK)
            self.s1, self.s2 = 1, 2
            self.m = 0.480 + 1.574 * self.w - 0.175 * self.w ** 2
            self.ac = 0.077796070 * self.R ** 2, self.Tc ** 2 / self.Pc
            self.bc = 0.086640 * self.R * self.Tc / self.Pc
        elif self.eq == "PR":
            # Peng-Robinson (PR)
            self.s1, self.s2 = 1 + 2 ** 0.5, 1 - (2 ** 0.5)
            self.m = 0.37464 + 1.54226 * self.w - 0.26992 * self.w ** 2
            self.ac = 0.45723553 * self.R ** 2 * self.Tc ** 2 / self.Pc
            self.bc = 0.077796070 * self.R * self.Tc / self.Pc

            self.alfa = (1 + self.m * (1 - (self.T / self.Tc) ** 0.5)) ** 2
```

(continues on next page)

(proviene de la página anterior)

```
            self.dalfadT = - (self.m / self.T) * (self.T / self.Tc) ** 0.5 * (self.m␣
↪* (- (self.T / self.Tc) ** 0.5 + 1) + 1)
            ter_1 = 0.5 * self.m ** 2 * (self.T / self.Tc) ** 1.0 / self.T ** 2
            ter_2 = 0.5 * self.m * (self.T / self.Tc) ** 0.5 * (self.m * (- (self.T /␣
↪self.Tc) ** 0.5 + 1) + 1) / self.T ** 2

            self.d2alfaT2 = ter_1 + ter_2
            self.a_ii = self.ac * self.alfa
            self.b_ii = self.bc

            self.da_iidT = self.ac * self.dalfadT
            d2adT2_puros = self.ac * self.d2alfaT2

        elif self.eq == "RKPR":
            # (RKPR)
            print ("No actualizada, intentalo de nuevo !!! ")

        else:
            print ("Che boludo... Modelo no valido, intentalo de nuevo !!! ")


    def parametros(self):

        if self.nC > 1:
            self.aij = np.ones((len(self.ni), len(self.ni)))
            self.bij = np.ones((len(self.ni), len(self.ni)))
            self.daijdT = np.ones((len(self.ni), len(self.ni)))

            for j in range(self.nC):
                for i in range(self.nC):
                    self.aij[i, j] = (self.a_ii[i] * self.a_ii[j]) ** 0.5
                    self.bij[i, j] = (self.b_ii[i] + self.b_ii[j]) / 2
                    self.bij[i, j] = self.bij[i, j]
                    self.daijdT[i, j] = (self.da_iidT[i] * self.da_iidT[j]) ** 0.5

            for i in range(self.nC):
                for  j in range(self.nC):
                    if i == j:
                        self.aij[i, j] = self.a_ii[i] * (1 - self.kij[i, j])
                        self.daijdT[i, j] = self.da_iidT[i] * (1 - self.kij[i, j])
                    elif i != j:
                        self.aij[i, j] = self.aij[i, j] * (1 - self.kij[i, j])
                        self.daijdT[i, j] = self.daijdT[i, j] * (1 - self.kij[i, j])

        if self.nC == 1:
            return self.a_ii, self.b_ii, self.da_iidT
        else:
            return self.aij, self.bij, self.daijdT

    def parametro_D(self):
        if self.nC == 1:
            self.D = self.ni ** 2 * self.a_ii
            self.Di = 2 * self.ni * self.a_ii
        else:
            di = np.ones((len(self.ni), len(self.ni)))
            self.Di = np.ones((len(self.ni)))
            self.D = np.ones((len(self.ni)))
```

(continues on next page)

(proviene de la página anterior)

```
            for i in range(self.nC):
                for j in range(self.nC):
                    di[i, j] = self.ni[j] * self.aij[i, j]
                    self.Di[i] = 2 * np.sum(di[i, :])
            self.D = 0.5 * np.sum(self.ni * self.Di)

        return self.D

    def parametro_delta_1(self):

        if self.nC == 1:
            self.D1m = np.zeros((len(self.ni)-1))
            self.dD1i = np.ones((len(self.ni)))
            self.dD1ij = np.ones((len(self.ni), len(self.ni)))

            for i in range(self.nC):
                self.D1m = self.D1m + self.ni[i] * self.delta_1[i]

            self.D1m = self.D1m / self.nT

        else:
            self.D1m = np.zeros((len(self.ni)-1))
            self.dD1i = np.ones((len(self.ni)))
            self.dD1ij = np.ones((len(self.ni), len(self.ni)))

            for i in range(self.nC):
                self.D1m = self.D1m + self.ni[i] * self.delta_1[i]

            self.D1m = self.D1m / self.nT

            for i in range(self.nC):
                self.dD1i[i] = (self.delta_1[i] - self.D1m) / self.nT
                for j in range(self.nC):
                    self.dD1ij[i,j] = (2.0 * self.D1m - self.delta_1[i] - self.delta_
↪1[j]) / self.nT ** 2

        return self.D1m, self.dD1i, self.dD1ij

    def parametro_B(self):
        if self.nC == 1:
            self.B = self.ni * self.b_ii
        else:
            self.aux = np.zeros((len(self.ni)))
            for i in range(self.nC):
                for j in range(self.nC):
                    self.aux[i] = self.aux[i] + self.ni[j] * self.bij[i, j]

            self.B = np.sum(self.ni * self.b_ii)
            #print("B = ", self.B)

        return self.B

    def presion(self):
        '''
        Con el metodo presion(), se calcula la Presión P(T, V, N) del sistema
        para una temperatura T, cantidad de moles N y un volumen V
        R = Constante universal de los gases
```

(continues on next page)

(proviene de la página anterior)

```
        nT = Número total de moles en el sistema
        Pcal = Peos = Presión calculada con la ecuación de estado
        Arv = Primera derivada parcial de la energía de Helmholz con respecto al
        volumen V, a T y N constantes
        '''
        self.gv = self.R * self.B / (self.V * (self.V - self.B))
        self.fv = - 1 / ((self.V + self.s1 * self.B) * (self.V + self.s2 * self.B))
        self.ArV = -self.nT * self.gv * self.T - self.D * self.fv
        self.Pcal = self.nT * self.R * self.T / self.V - self.ArV

        return self.Pcal

    def dP_dV(self):
        self.dPdV = -self.ArV2 - self.R * self.T * self.nT / self.V ** 2
        return self.dPdV

    def Z_factor(self):
        self.Z = (self.P * self.V) / (self.nT * self.R * self.T)
        return self.Z

    def P_ideal(self):
        self.Pxi = (self.ni * self.P) / self.nT
        return self.Pxi

    def dF_dV(self):
        '''
        Primera derivada de F con respecto al volumen Ecu. (68)
        '''
        self.gv = self.R * self.B / (self.V * (self.V - self.B))
        self.fv = - 1 / ((self.V + self.s1 * self.B) * (self.V + self.s2 * self.B))
        self.ArV = -self.nT * self.gv * self.T - self.D * self.fv
        return self.ArV

    def dF_dVV(self):
        '''
        Segunda derivada de F con respecto al volumen Ecu. (74)
        '''
        self.gv2 = self.R * (1 / self.V ** 2 - 1 / (self.V - self.B) ** 2)
        self.fv2 = (- 1 / (self.V + self.s1 * self.B) ** 2 + 1 / (self.V + self.s2 *␣
↪self.B) ** 2) / self.B / (self.s1 - self.s2)
        self.ArV2 = - self.nT * self.gv2 * self.T - self.D * self.fv2
        return self.ArV2

    def volumen_1(self):
        '''
        Calculo del volumen V(T,P,n) del fluido a una temperatura T, presión P
        y número de moles totales nT especificados.
        Se utiliza el método de Newton con derivada de la función analitica.
        Pendiente cambiar por una función de Scipy.
        '''
        self.V = 1.05 * self.B
        lnP = np.log(self.P)
        Pite = self.presion()
        lnPcal = np.log(Pite)
        h = lnP - lnPcal
        errorEq = abs(h)
        i = 0
```

(continues on next page)

(proviene de la página anterior)

```
        s = 1.0

        while errorEq > self.ep:
            self.parametro_D()
            self.parametro_B()
            self.dF_dV()
            self.dF_dVV()
            dPite = self.dP_dV()
            Pite = self.presion()
            lnPcal = np.log(Pite)
            h = lnP - lnPcal
            dh = -dPite
            self.V = self.V - s * h / dh
            errorEq = abs(h)
            i += 1
            if i >= 900:
                pass
                #break

        return self.V

    def funcion_energia_F(self):
        self.g = self.R * np.log(1 - self.B / self.V)
        self.bv = self.B / self.V
        self.f = np.log((self.V + self.s1 * self.B) / (self.V + self.s2 * self.B)) /␣
↪self.B / (self.s1 - self.s2)
        self.Ar = -self.nT * self.g * self.T - self.D * self.f
        return self.g, self.f, self.Ar, self.bv

    def tomar_B(self):
        print ("tomando B =", self.B)
        return self.B + 10

    def derivadas_delta_1(self):
        auxD2 = (1 + 2 / (1 + self.s1) ** 2)

        como_1 = (1 / (self.V + self.s1 * self.B) + 2 / (self.V + self.s2 * self.B) /␣
↪(1 + self.s1) ** 2)
        como_2 = self.f * auxD2
        self.fD1 = como_1 - como_2
        self.fD1 = self.fD1/(self.s1 - self.s2)

        return self.fD1

    def primeras_derivadas1(self):

        if self.nC == 1:
            AUX = self.R * self.T / (self.V - self.B)
            self.fB = -(self.f + self.V * self.fv) / self.B
            self.FFB = self.nT * AUX - self.D * self.fB
            self.Di = 2 * self.nT * self.ac * self.alfa
            self.Bi = self.bc

            if self.eq != "RKPR":
                self.Arn = -self.g * self.T + self.FFB * self.Bi - self.f * self.Di
            else:
                self.Arn = -self.g * self.T + self.FFB * self.Bi - self.f * self.Di -␣
↪self.D * self.fD1 * self.dD1i
```

(continues on next page)

(proviene de la página anterior)

```
        else:
            # Derivando la ecuación (64) se obtiene la ecuación eq (106)
            self.Bi = np.ones((len(self.ni)))
            for i in range(self.nC):
                self.Bi[i] = (2 * self.aux[i] - self.B) / self.nT

            AUX = self.R * self.T / (self.V - self.B)
            self.fB = -(self.f + self.V * self.fv) / self.B
            self.FFB = self.nT * AUX - self.D * self.fB

            if self.eq != "RKPR":
                self.Arn = -self.g * self.T + self.FFB * self.Bi - self.f * self.Di
            else:
                auxD2 = (1 + 2 / (1 + self.s1) ** 2)
                print("B delta1 = ", self.B)
                co_1 = (1 / (self.V + self.s1 * self.B) + 2 / (self.V + self.s2 *␣
↪self.B) / (1 + self.s1) ** 2)
                co_2 = self.f * auxD2
                self.fD1 = co_1 - co_2
                self.fD1 = self.fD1/(self.s1 - self.s2)
                self.Arn = -self.g * self.T + self.FFB * self.Bi - self.f * self.Di -␣
↪self.D * self.fD1 * self.dD1i

        return self.Arn, self.Arn, self.Arn


    def coeficientes_fugacidad(self):
        self.Z = self.Z_factor()
        self.lnOi = self.Arn / (self.R * self.T) - np.log(self.Z)
        self.Oi = np.exp(self.lnOi)
        return self.Oi

    def fugacidad(self):
        self.Z = self.Z_factor()
        self.Pxi = self.P_ideal()
        self.lnFi = self.Arn / (self.R * self.T) - np.log(self.Z) + np.log(self.Pxi)
        self.Fi = np.exp(self.lnFi)
        self.PHILOG = self.Arn / (self.R * self.T) - np.log(self.Z)
        self.PHILOG_i = self.Arn - np.log(self.Z)
        self.FUGLOG = self.Arn / (self.R * self.T) + np.log(self.ni) + np.log((self.
↪nT * self.R * self.T) / self.V)
        return self.Fi

    def exp_sol(self):
        '''
        Este método calcula el factor de corrección de la fugacidad del
        componente fluido para determinar la fugacidad del mismo componente
        en estado sólido.
        Fugacidad del sólido puro
        fi_s(T, P) = fi_l(T, P) * EXP(T, P)
        '''
        Tfus = 323.75
        # Temperatura de fusion de n-tetracosane
        # Unidad de Ti_f en Kelvin
        par_sol = np.array([[-176120.0, 8196.20, -55.911, 0.19357, -0.0002235],
                            [-1.66e6, 8.31e3, 0.0, 0.0, 0.0]])
        par_liq = np.array([[423160.0, 1091.9, 0.0, 0.0, 0.0],
```

(continues on next page)

(proviene de la página anterior)

```
                            [7.01e5, 1.47e3, 0.0, 0.0, 0.0]])
        #print ("par_sol", par_sol)
        #print ("par_liq", par_liq)
        # Las unidades de Cp están en J/Kmol.K
        Cp_solido = par_sol[:, 0] + par_sol[:, 1] * T + par_sol[:, 2] * T ** 2 + par_
↪sol[:, 3] * T ** 3 + par_sol[:, 4] * T ** 4
        #print ("Cp_solido", Cp_solido)
        Cp_liquido= par_liq[:, 0] + par_liq[:, 1] * T + par_liq[:, 2] * T ** 2 + par_
↪liq[:, 3] * T ** 3 + par_liq[:, 4] * T ** 4
        #print ("Cp_liquido", Cp_liquido)
        DeltaCp = (Cp_solido - Cp_liquido) * (1.0 / 1000)
        print ("Delta Cp", DeltaCp)

        #Unidades de Delta H de fusión en Kcal/mol
        DeltaH_f = np.array([13.12, 21.23]) * (1000 / 1.0) * (4.18 / 1.0)
        #print ("Delta H de fusion", DeltaH_f)
        T_f = np.array([323.75, 349.05])
        #print ("Temperaturas de fusion = ", T_f)

        Rp = 8.314
        A = (DeltaH_f / (Rp * Tfus)) * (1 - (Tfus / T))
        B = (DeltaCp / Rp) * (1 - (Tfus / T))
        C = (DeltaCp / Rp) * np.log(Tfus / T)
        self.EXP = np.exp(A - B - C)

        print ("A = ", A)
        print ("B = ", B)
        print ("C = ", C)
        print ("EXP = ", self.EXP)

        return self.EXP

    def exp_sol_1(self):
        '''
        Este método calcula el factor de corrección de la fugacidad del
        componente fluido para determinar la fugacidad del mismo componente
        en estado sólido.
        Fugacidad del sólido puro
        fi_s(T, P) = fi_l(T, P) * EXP(T, P)
        '''
        Tpt = 323.75
        Ppt = 1.38507E-8
        R = 8.314472
        AH = 54894000
        Av = -0.0376300841 #m3/kmol

        a = ((AH / (R * Tpt)) * (1 - (Tpt / self.T))) / 1000
        b = ((Av / (R * self.T)) * (self.P - Ppt)) * 100
        self.EXP_1 = a + b

        return self.EXP_1

    def exp_sol_3(self):
        '''
        Este método calcula el factor de corrección de la fugacidad del
        componente fluido para determinar la fugacidad del mismo componente
        en estado sólido.
```

(continues on next page)

(proviene de la página anterior)

```
        Fugacidad del sólido puro
        fi_s(T, P) = fi_l(T, P) * EXP(T, P)
        '''
        # [=] K
        # [=] bar
        # [m3 / Kmol]
        # Constante R [=] 0.08314472 bar.l/(mol.K)

        Tpt = 323.75
        Ppt = 3.2015002E-8
        #self.Avsl = -0.0565500835

        c1 = -14213.5004
        c2 = 605153.4382
        c3 = -591592.556

        R = 0.08314472

        A1 = c1 * (1 - Tpt / self.T)
        A2 = c2 * (-1 + Tpt / self.T + np.log(self.T / Tpt))
        A3 = c3 * (-1 + self.T / (2 * Tpt) + Tpt / (2 * self.T)) + (Tpt / self.T) *␣
↪(self.P - Ppt)

        FE = (self.Avsl / (self.R * self.T)) * (A1 + A2 + A3)
        self.EXP_3 = np.exp(FE)
        return self.EXP_3


    def fluido(self):
        ab = self.parametros()
        D = self.parametro_D()
        B = self.parametro_B()
        Vol_1 = self.volumen_1()
        F = self.funcion_energia_F()
        dF = self.primeras_derivadas1()
        Z = self.Z_factor()
        Zcal = (self.P * Vol_1) / (self.nT * self.R * self.T)
        Pq = self.presion()
        self.Fug = self.fugacidad()
        self.CoeFug = self.coeficientes_fugacidad()
        return self.Fug

    def solido(self):
        if self.nC == 1:
            Fug = self.fluido()
            #EXP = self.exp_sol()
            #EXP = self.exp_sol_1()
            EXP = self.exp_sol_3()

            FugS = Fug[0] * EXP
        else:
            print ("Aún no se qué hacer para una mezcla de sólidos !!!")
            FugS = 1

        return FugS

#----------------
```

(continues on next page)

(proviene de la página anterior)

```
def calculaFugacidad(x, Pe, nif, nCf, eq, TcDato, PcDato, wDAto, Avsl):
    #-------------------------------------------------------------------------
    # Temperatura en [=] K
    # Presión en [=] bar
    # Constante R [=] 0.08314472 bar.l/(mol.K)
    # x = variable que se cálcula, puede ser T ó P para el equilibrio sólido-fluido
    # Pe = Presión del sistema especificada
    # nif = número de moles del componente (i) en cada fase (f)
    # nCf = número de componentes en una fase (f)
    # eq = modelo de ecuación de estado, SRK, PR, RKPR
    # TcDato = Temperatura critica de la "base de datos"
    # PcDato = Presión critica de la "base de datos"
    # wDato = Factor acentrico de la "base de datos"
    # Avsl = Delta de volumen sólido-fluido

    # ep = Criterio de convergencia del método def volumen_1(self, P)

    T = x # 335.42 # x # 366.78 # 356.429 # 335.42 # 348.89 #327.0
    #print("Temperatura = ", T)
    P = Pe # 2575.0 # 2064.7 # 1524.4 #1164.2 # 865.0
    # 560.3 # x #1054.6 #1560.3 # 2064.7 # 1524.4 # 560.3 # 1164.2 #865.0
    R = 0.08314472
    ep = 1e-5#1e-6
    #-------------------------------------------------------------------------
    Tcm = TcDato
    Pcm = PcDato
    wm = wDato

    nC = nCf

    if nC == 1:
        #print (".....................................................")

        #ni = nif
        ni = np.array([1.0])

        #print ("Número de moles = ", ni)
        # C24
        kij = 0.0
        lij = 0.0

        # Metano - Etano
        delta_1 = np.array([0.85])
        k = np.array([1.50758])
        #C24
        Tc = Tcm[1]
        Pc = Pcm[1]
        w = wm[1]
        print (".....................................................")
    elif nC == 2:
        # metano - C24
        #ni = np.array([1-nif, nif])
        ni = nif #np.array([1-nif, nif])

        #ni = np.array([1 - 0.901, 0.901])
        #--------------------------------
```

(continues on next page)

(proviene de la página anterior)

```
        #ni = np.array([1 - 0.26, 0.26])

        #ni = np.array([1 - 0.104, 0.104])
        #print ("Número de moles = ", ni)

        kij = np.array([[0.000000, 0.083860],
                        [0.083860, 0.000000]])

        kij = np.array([[0.000000, 0.059600],
                        [0.059600, 0.000000]])


        lij = 0.0132


        #kij = np.array([[0.000000, 0.00],
        #                [0.00, 0.000000]])

        #lij = 0.0


        # Metano - C24
        delta_1 = np.array([0.85, 2.40])
        k = np.array([1.50758, 4.90224])

        # metano sigma1 = 0.9253, sigma = 0.85, k = 1.49345, k = 1.50758
        # C24 sigma = 2.40 k = 4.90224

        Tc = Tcm
        Pc = Pcm
        w = wm
        print ("Temperatura Critica = ", Tc, "K")
        print ("Presión Critica = ", Pc, "bar")
        print ("Factor Acentrico = ", w)
        #print (".....................................................................")

    # Tempertura reducidad
    Tr = T / Tc
    # C24 puro
    V = 0.141604834257319
    nT = np.sum(ni)

    fugacidad = Fugacidad(eq, w, Tc, Pc, Tr, R, ep, ni, nT, nC, V, T, P, kij, lij,␣
↪delta_1, k, Avsl)

    print(fugacidad.exp_sol_3())

    if nC == 1:
        SOL = fugacidad.solido()

        return SOL
    else:
        flu_1 = fugacidad.fluido()
        return flu_1
```

(continues on next page)

(proviene de la página anterior)

```
#----------------
```

```
def equilibrioSF(x, Pe, nif, n1, n2, Avsl):

    # fugacidad del sólido puro
    FugS = calculaFugacidad(x, Pe, nif, n1, eq, TcDato, PcDato, wDato, Avsl)
    print(eq, TcDato, PcDato, wDato, Avsl)
    # fugacidad del fluido pesado en la mezcla fluida
    FugF = calculaFugacidad(x, Pe, nif, n2, eq, TcDato, PcDato, wDato, Avsl)

    # Función de igualdad de fugacidades del sólido y el fluido
    eqSF = np.abs(np.abs(np.log(FugS)) - np.abs(np.log(FugF[1])))
    print ("-"*80)
    print ("ln(Fugacidad Sólido) = ", np.log(FugS))
    print ("ln(Fugacidad Fluido) = ", np.log(FugF[1]))
    print ("ln(Fugacidad Sólido) - ln(Fugacidad Fluido) = ", eqSF)

    return eqSF


eq = 'PR'
#Avsl = -0.0565500835
#Avsl = -0.09605965500835

#initial_temperature = [346.5] # T [=] K
#initial_pressure = 136.9 # [=] bar

#Tcal = fsolve(equilibrioSF,initial_temperature,args=(initial_pressure, 1, 2, Avsl),␣
↪xtol=1e-4)
#print(Tcal, "K")

t_exp = [323.65, 326.04, 326.43, 328.12, 329.45, 329.89, 333.43, 335.12, 340.19, 344.
↪58, 346.65, 352.53, 362.45, 362.76, 371.82, 379.74]
temp = np.array(t_exp)

p_exp = [1, 101.0, 136.9, 183.8, 266.2, 266.8, 426.9, 480.3, 718.9, 912.5, 1010.6,␣
↪1277.8, 1778.0, 1825.1, 2323.4, 2736.1]
pres= np.array(p_exp)

pos = np.arange(len(pres))
Tcal = np.ones((len(pres)))
Tcal

Tres = np.array([ 322.65861561,  324.91946742,  325.73456905,  326.80151121,
        328.68045402,  328.69415114,  332.3526483 ,  333.57248076,
        338.99640222,  343.33723415,  345.50684642,  351.28742799,
        361.49784425,  362.4145721 ,  371.63445321,  378.63493779])

Tcal - temp

Avsl = -0.32595074
Avsl


class Flash():
```

(continues on next page)

(proviene de la página anterior)

```
    def __init__(self, zi_F, temperature_f, pressure_f, TcDato_f, PcDato_f, wDato_f):
        self.zi = zi_F
        self.T = temperature_f
        self.P = pressure_f
        self.Tc = TcDato_f
        self.Pc = PcDato_f
        self.w = wDato_f

    def wilson(self):
        # Ecuación wilson
        lnKi = np.log(self.Pc / self.P) + 5.373 * (1 + self.w) * (1 - self.Tc / self.
↪T)
        self.Ki = np.exp(lnKi)
        return self.Ki

    def beta(self):
        # Estimación de la fracción de fase de vapor en el sistema
        self.Ki = self.wilson()
        #Bmin = np.divide((self.Ki * self.zi - 1), (self.Ki - 1))
        Bmin = (self.Ki * self.zi - 1) / (self.Ki - 1)

        #print (("Bmin_inter = ", Bmin))

        Bmax = (1 - self.zi) / (1 - self.Ki)
        #print (("Bmax_inter = ", Bmax))
        self.Bini = (np.max(Bmin) + np.min(Bmax)) / 2
        print("inib =", self.Bini)
        return self.Bini

    def rice(self):
        # Ecuación de Rachford-Rice para el equilibrio líqudo-vapor
        self.fg = np.sum(self.zi * (self.Ki - 1) / (1 - self.Bini + self.Bini * self.
↪Ki))
        self.dfg = - np.sum(self.zi * (self.Ki - 1) ** 2 / (1 - self.Bini + self.Bini␣
↪* self.Ki) ** 2)
        #print g, dg
        return self.fg, self.dfg

    def composicion_xy(self):
        # Ecuación de Rachford-Rice para calcular la composición del equilibrio␣
↪líqudo-vapor
        self.xi = self.zi / (1 - self.Bini + self.Bini * self.Ki)
        self.yi = (self.zi * self.Ki) / (1 - self.Bini + self.Bini * self.Ki)
        self.li = (self.zi * (1 - self.Bini)) / (1 - self.Bini + self.Bini * self.Ki)
        self.vi = (self.zi * self.Bini * self.Ki) / (1 - self.Bini + self.Bini * self.
↪Ki)

        return self.xi, self.yi, self.li, self.vi

    def flash_ideal(self):
        # Solución del flash (T,P,ni) isotermico para Ki_(T,P)
        self.Bini = self.beta()
        self.Ki = self.wilson()
        # print ("Ki_(P, T) = ", self.Ki)
        Eg = self.rice()
        errorEq = abs(Eg[0])
        # Especificaciones del método Newton precario, mientras se cambia por una␣
↪librería Scipy
```

(continues on next page)

(proviene de la página anterior)

```
        i, s, ep = 0, 1, 1e-5

        while errorEq > ep:
            Eg = self.rice()
            self.Bini = self.Bini - s * Eg[0] / Eg[1]
            errorEq = abs(Eg[0])
            i += 1
            if i >= 50:
                break

        xy = self.composicion_xy()
        print ("-"*53, "\n", "-"*18, "Mole fraction", "-"*18, "\n","-"*53)
        print ("\n", "-"*13, "Zi phase composition", "-"*13, "\n")
        print ("{0} = {1} \n {2} = {3} \n {4}={5} \n {6}={7} \n".format(Componentes_
↪f1.value, self.zi[0], Componentes_f2.value, self.zi[1], Componentes_f3.value, self.
↪zi[2], Componentes_f4.value, self.zi[3]))
        print ("Sumatoria zi = {0}".format(np.sum(self.zi)))
        print ("\n", "-"*13, "Liquid phase composition", "-"*13, "\n")
        print ("{0} = {1} \n {2} = {3} \n {4}={5} \n {6}={7} \n".format(Componentes_
↪f1.value, self.xi[0], Componentes_f2.value, self.xi[1], Componentes_f3.value, self.
↪xi[2], Componentes_f4.value, self.xi[3]))
        print ("Sumatoria xi = {0}".format(np.sum(self.xi)))
        print ("\n", "-"*14, "Vapor phase composition", "-"*13, "\n")
        print ("{0} = {1} \n {2} = {3} \n {4}={5} \n {6}={7} \n".format(Componentes_
↪f1.value, self.yi[0], Componentes_f2.value, self.yi[1], Componentes_f3.value, self.
↪yi[2], Componentes_f4.value, self.yi[3]))
        print ("Sumatoria yi = {0}".format(np.sum(self.yi)))
        print ("-"*53, "\n","Beta = {0}".format(self.Bini), "\n")
        print ("\n","Función R&R = {0}".format(Eg[0]), "\n")
        print ("\n","Derivada función R&R = {0}".format(Eg[1]), "\n", "-"*53)


        return #Eg[0], Eg[1], self.Bini

class FlashHP(Fugacidad, Flash):

    def __init__(self, zF):
        Fugacidad.__init__(self, eq, w, Tc, Pc, Tr, R, ep, ni, nT, nC, V, T, P, kij,␣
↪lij, delta_1, k, Avsl)
        self.zF = zF



    def flash_PT(self):
        # Solución del flash (T,P,ni) isotermico para Ki_(T,P,ni)
        flashID = self.flash_ideal()
        print ("flash (P, T, zi)")
        print ("g, dg, B = ", flashID)
        print ("-"*66)

        self.Bini = flashID[2]
        print ("Beta_r ini = ", self.Bini)
        moles = self.composicion_xy()

        self.xi, self.yi = moles[0], moles[1]
        nil, niv = moles[2], moles[3]
```

(continues on next page)

(proviene de la página anterior)

```
        fi_F = self.fugac()

        self.Ki = fi_F[0] / fi_F[1]

        L = 1.0

        self.Ki = self.Ki * L

        Ki_1 = self.Ki
        print ("Ki_(P, T, ni) primera = ", self.Ki)

        print ("-"*66)

        #self.Ki = np.array([1.729, 0.832, 0.640])

        #self.Ki = self.wilson(self.Pc, self.Tc, self.w, self.T)
        #print "Ki_(P, T) = ", self.Ki

        while 1:
            i, s = 0, 0.1

            while 1:
                Eg = self.rice()
                print (Eg)
                self.Bini = self.Bini - s * Eg[0] / Eg[1]
                print (self.Bini)
                errorEq = abs(Eg[0])
                i += 1
                #print i

                #if self. Bini < 0 or self.Bini > 1:
                    #break
                #    self.Bini = 0.5
                if i >= 50:
                    pass
                    #break
                if errorEq < 1e-5:
                    break

            print ("Resultado Real = ", Eg)
            print (" Beta r = ", self.Bini)

            moles = self.composicion_xy(zi, self.Ki, self.Bini)
            self.xi, self.yi = moles[0], moles[1]

            #xy = self.composicion_xy(zi, self.Ki, self.Bini)

            print ("C1 -i-C4 n-C4")
            print ("-"*13, "Composición de fase líquida", "-"*13)
            print ("xi = ", moles[0])
            print ("Sxi = ", np.sum(moles[0]))
            print ("-"*13, "Composición de fase vapor", "-"*13)
            print ("yi = ", moles[1])
            print ("Syi = ", np.sum(moles[1]))

            fi_F = self.fugac()
```

(continues on next page)

(proviene de la página anterior)

```
            self.Ki = fi_F[0] / fi_F[1]
            Ki_2 = self.Ki
            dKi = abs(Ki_1 - Ki_2)
            Ki_1 = Ki_2
            print ("Ki_(P, T, ni) = ", self.Ki)

            fun_Ki = np.sum(dKi)
            print ("fun_Ki = ", fun_Ki)

            if fun_Ki < 1e-5:
                break

        return flashID

url = 'Lectura Juan.xlsx'

class DataGPEC():

    def __init__(self, url):
        self.url = url

    def leerGPEC_1(self):
        """
        El siguiente script python, se puede mejorar generalizando la lectura de␣
↪etiquetas,
        mientras se pasa la transición GPEC librería
        """
        marcas = ['VAP', 'CRI', 'CEP']

        GPEC = pd.read_excel(url)

        """
        Revisar las etiquetas, nombre, roturlos de las figurar generadas con este␣
↪script Python
        para que sean acordes a las variables que se desean gráficar, mientras se␣
↪automatiza este
        proceso.
        """


        #-------------------------------------------------------------------------
↪-
        DatosGPEC = pd.DataFrame(GPEC)
        VAP = DatosGPEC.loc[(DatosGPEC['T(K)'] == marcas[0])]
        etiquetaVAP = VAP.index.get_values()
        inicioVAP = etiquetaVAP[0]+1
        finalVAP = etiquetaVAP[1]-2

        #-------------------------------------------------------------------------
↪-
        self.TemperaturaVAP = np.array([DatosGPEC.ix[inicioVAP:finalVAP,0]], dtype=np.
↪float)
        self.PresionVAP = np.array([DatosGPEC.ix[inicioVAP:finalVAP,1]], dtype=np.
↪float)
        self.VolumenLiqVAP = np.array([DatosGPEC.ix[inicioVAP:finalVAP,2]], dtype=np.
↪float)
        self.VolumenVapVAP = np.array([DatosGPEC.ix[inicioVAP:finalVAP,3]], dtype=np.
↪float)
```

(continues on next page)

(proviene de la página anterior)

```
        #-------------------------------------------------------------------------------
↪-
        CRI = DatosGPEC.loc[(DatosGPEC['T(K)'] == marcas[1])]
        etiquetaCRI = CRI.index.get_values()
        inicioCRI = etiquetaCRI[0]+1
        finalCRI = etiquetaCRI[1]-2
        #-------------------------------------------------------------------------------
↪-
        self.TemperaturaCRI = np.array([DatosGPEC.ix[inicioCRI:finalCRI,0]], dtype=np.
↪float)
        self.PresionCRI = np.array([DatosGPEC.ix[inicioCRI:finalCRI,1]], dtype=np.
↪float)
        self.VolumenLiqCRI = np.array([DatosGPEC.ix[inicioCRI:finalCRI,2]], dtype=np.
↪float)
        self.VolumenVapCRI = np.array([DatosGPEC.ix[inicioCRI:finalCRI,3]], dtype=np.
↪float)
        #-------------------------------------------------------------------------------
↪-
        """
        En la segunda línea critica se tiene como referencia el final de la primera␣
↪línea critica
        y la etiqueta CEP
        """

        CEP = DatosGPEC.loc[(DatosGPEC['T(K)'] == marcas[2])]
        etiquetaCEP = CEP.index.get_values()
        inicioCRI_2 = etiquetaCRI[1]+1
        finalCRI_2 = etiquetaCEP[0]-2

        self.TemperaturaCRI_2 = np.array([DatosGPEC.ix[inicioCRI_2:finalCRI_2,0]],␣
↪dtype=np.float)
        self.PresionCRI_2 = np.array([DatosGPEC.ix[inicioCRI_2:finalCRI_2,1]],␣
↪dtype=np.float)
        self.VolumenLiqCRI_2 = np.array([DatosGPEC.ix[inicioCRI_2:finalCRI_2,2]],␣
↪dtype=np.float)
        self.VolumenVapCRI_2 = np.array([DatosGPEC.ix[inicioCRI_2:finalCRI_2,3]],␣
↪dtype=np.float)

        return self.TemperaturaCRI_2

    def presionVapor(self):
        clear_output()
        pyplot.close("all")
        pyplot.scatter(self.TemperaturaVAP,self.PresionVAP, color = 'red', label =
↪'Presión de Vapor')
        pyplot.title('Temperatura-Presión')
        pyplot.legend(loc="upper left")
        pyplot.xlabel('Temperatura [=] K')
        pyplot.ylabel('Presión [=] bar')

    def densidadPresion(self):
        clear_output()
        pyplot.close("all")
        pyplot.scatter(self.VolumenLiqVAP,self.PresionVAP, color = 'red', label =
↪'Líquido')
        pyplot.scatter(self.VolumenVapVAP,self.PresionVAP, color = 'blue', label =
↪'Vapor')
```

(continues on next page)

(proviene de la página anterior)

```
        pyplot.title('Diagrama Densidad-Presión')
        pyplot.legend(loc="upper right")
        pyplot.xlabel('Densidad [=] -')
        pyplot.ylabel('Presión [=] bar')

    def diagramaTPcritico(self):
        clear_output()
        pyplot.close("all")
        pyplot.scatter(self.TemperaturaCRI,self.PresionCRI, color = 'red', label =
↪'Presión Critica')
        pyplot.title('Diagrama Temperatura Cri-Presión Cri')
        pyplot.legend(loc="upper left")
        pyplot.xlabel('Temperatura [=] K')
        pyplot.ylabel('Presión [=] bar')

    def diagramaDensidadCri(self):
        clear_output()
        pyplot.close("all")
        pyplot.scatter(self.VolumenLiqCRI,self.PresionCRI, color = 'red', label =
↪'Líquido')
        pyplot.scatter(self.VolumenVapCRI,self.PresionCRI, color = 'blue', label =
↪'Vapor')
        pyplot.title('Diagrama Densidad Critica')
        pyplot.legend(loc="upper right")
        pyplot.xlabel('Densidad [=] -')
        pyplot.ylabel('Presión [=] bar')

    def diagramaCritico_2(self):
        clear_output()
        pyplot.close("all")
        fig_2= pyplot.scatter(self.TemperaturaCRI_2,self.PresionCRI_2)
        pyplot.scatter(self.TemperaturaCRI_2,self.PresionCRI_2, color = 'red', label␣
↪= 'Presión de Critica 2')
        pyplot.title('Diagrama Critico 2')
        pyplot.legend(loc="upper left")
        pyplot.xlabel('Temperatura [=] K')
        pyplot.ylabel('Presión [=] bar')
#-------------------------------------------------------------------------------
```

8.4 Interfaz «gráfica»

```
Componentes_1 = widgets.SelectMultiple(
    description="Component 1",
    options=list(Etiquetas))

Componentes_2 = widgets.SelectMultiple(
    description="Component 2",
    options=list(Etiquetas))

button = widgets.Button(description="Upload Data")

def cargarDatos(b):
    clear_output()
    print("Component 1: ", Componentes_1.value)
```

(continues on next page)

(proviene de la página anterior)

```
    Nombre = Componentes_1.value
    Propiedades = data2.loc[Nombre]
    Factor_Acentrico_1 = Propiedades[0]
    Temperatura_Critica_1 = Propiedades[1]
    Presion_Critica_1 = Propiedades[2]
    Z_Critico_1 = Propiedades[3]

    #print(Propiedades)
    print ("Acentric Factor = ", Factor_Acentrico_1)
    print ("Critical Temperature = ", Temperatura_Critica_1, "K")
    print ("Critical Pressure = ", Presion_Critica_1, "bar")
    print ("Z_Critical = ", Z_Critico_1, "\n")


    print("Component 2: ", Componentes_2.value)
    Nombre = Componentes_2.value
    Propiedades = data2.loc[Nombre]
    Factor_Acentrico_2 = Propiedades[0]
    Temperatura_Critica_2 = Propiedades[1]
    Presion_Critica_2 = Propiedades[2]
    Z_Critico_2 = Propiedades[3]

    #print(Propiedades)
    print ("Acentric Factor  = ", Factor_Acentrico_2)
    print ("Critical Temperature = ", Temperatura_Critica_2, "K")
    print ("Critical Pressure = ", Presion_Critica_2, "bar")
    print ("Z_Critical = ", Z_Critico_2)

    global TcDato, PcDato, wDato

    TcDato = np.array([Temperatura_Critica_1, Temperatura_Critica_2])
    PcDato = np.array([Presion_Critica_1, Presion_Critica_2])
    wDato = np.array([Factor_Acentrico_1, Factor_Acentrico_2])


button.on_click(cargarDatos)
#display(button)

page1 = widgets.VBox(children=[Componentes_1, Componentes_2, button], padding=4)


#VBox([VBox([Button(description='Press'), Dropdown(options=['a', 'b']),␣
↪Button(description='Button')]),
#      VBox([Button(), Checkbox(), IntText()]),
#      VBox([Button(), IntSlider(), Button()])], background_color='#EEE')


ecuacionEstado = widgets.Dropdown(description='Fluid :', padding=4, options=['SRK',
↪'PR', 'RKPR'])
modeloSolido = widgets.Dropdown(description='Solid :', padding=4, options=['Model I',
↪'Model II', 'Model III'])

button = widgets.Button(description="Upload Models")

def cargarModelos(b):
    clear_output()
    global eq
```

(continues on next page)

(proviene de la página anterior)

```
    eq = ecuacionEstado.value

    print("Component 1: ", Componentes_1.value)
    print("Component 2: ", Componentes_2.value)

    print("Fluid Model : ", ecuacionEstado.value)
    print("Solid Model : ", modeloSolido.value)



button.on_click(cargarModelos)

page2 = widgets.Box(children=[ecuacionEstado, modeloSolido, button], padding=4)


Temp_ini = widgets.Text(description='Initial', padding=4, value="0.0")
Temp_fin = widgets.Text(description='Final', padding=4, value="0.0")

Pres_ini = widgets.Text(description='Initial', padding=4, value="0.0")
Pres_fin = widgets.Text(description='Final', padding=4, value="0.0")

n1 = widgets.Text(description='Mole light component', padding=4, value="0.0")
n2 = widgets.Text(description='Mole heavy component', padding=4, value="0.0")

#button = widgets.Button(description="Cargar Condiciones")

titulo = widgets.HTML(value="<C><H1> System Conditions <H1>")
tempe_info = widgets.HTML(value="<C><H3> Temperature <H3>")
press_info = widgets.HTML(value="<C><H3> Pressure <H3>")
fluid_info = widgets.HTML(value="<C><H3> Mole fracction in the fluid <H3>")

button = widgets.Button(description="Upload Conditions")

def cargarParametros(b):
    clear_output()

    global initial_temperature, initial_pressure, nif

    initial_temperature = float(Temp_ini.value)
    initial_pressure = float(Pres_ini.value)
    nif = np.array([float(n1.value), float(n2.value)])

    print("Component 1: ", Componentes_1.value)
    print("Component 2: ", Componentes_2.value)

    print("Fluid Model : ", ecuacionEstado.value)
    print("solid Model : ", modeloSolido.value)

    print("Initial_temperature = ", initial_temperature, type(initial_temperature))
    print("Final_temperature = ", Temp_fin.value)

    print("Initial_pressure =", initial_pressure, type(initial_pressure))
    print("Final_pressure =", Pres_fin.value)

    print("Mole fraccion light component n1 =", n1.value)
    print("Mole fraccion heavy component n2 =", n2.value)
```

(continues on next page)

(proviene de la página anterior)

```
    print("Mole fracction in the fluid ", nif)

    print(initial_temperature, type(initial_temperature))


button.on_click(cargarParametros)

page3 = widgets.Box(children=[titulo, tempe_info, Temp_ini, Temp_fin, press_info,␣
↪Pres_ini, Pres_fin, fluid_info, n1, n2, button], padding=4)







button = widgets.Button(description="Solid-Fluid")
#display(button)


nnCC_1 = 1
nnCC_2 = 2

def calcularSolidoFluido(b):
    clear_output()
    #Tcal = fsolve(equilibrioSF,guess,args=(Pe, nnCC_1, nnCC_2), xtol=1e-4)

    #initial_temperature = [346.5] # T [=] K
    #initial_pressure = 137.9 # [=] bar

    Tcal = fsolve(equilibrioSF,initial_temperature,args=(initial_pressure, nif, 1, 2,␣
↪Avsl), xtol=1e-4)
    print("Temperature ESF = ", Tcal, "K")


button.on_click(calcularSolidoFluido)
#display(button)


page4 = widgets.Box(children=[button], padding=4)

button = widgets.Button(description="Diagram Solid-Fluid")

def DiagramaSolidoFluido(b):
    clear_output()
    #Tcal = fsolve(equilibrioSF,guess,args=(Pe, 1, 2), xtol=1e-4)
    #Tcal = fsolve(equilibrioSF,guess,args=(Pe, nnCC_1, nnCC_2), xtol=1e-4)
    initial_temperature =346.5 # T [=] K
    initial_pressure = 136.9 # [=] bar

    #346.5 136.9
    # n1, n2 = 1, 2 por defecto para el equilibrio sólido-fluido
    Tcal = fsolve(equilibrioSF,initial_temperature,args=(initial_pressure, nif, 1, 2,␣
↪Avsl), xtol=1e-4)

    print(Tcal, "K")
```

(continues on next page)

(proviene de la página anterior)

```
    pyplot.scatter(Tres,pres, color = 'red', label = 'PR')
    pyplot.scatter(temp,pres, label = 'Data')
    pyplot.title('Temperature Equilibrium Solid Liquid')
    pyplot.legend(loc="upper left")
    pyplot.xlabel('Temperature [=] K')
    pyplot.ylabel('Pressure [=] bar')


button.on_click(DiagramaSolidoFluido)


page5 = widgets.Box(children=[button], padding=4)




DatosTemperatura_Exp = np.array([323.65, 326.04, 326.43])
#DatosTemperatura_Exp = np.array([323.65, 326.04, 326.43, 328.12])

DatosPresionp_Exp = np.array([1.0, 101.0, 136.9])
#DatosPresionp_Exp = np.array([1.0, 101.0, 136.9, 183.8])


posicion = np.arange(len(DatosPresionp_Exp))
TemperaturasModelo = np.ones((len(DatosPresionp_Exp)))
TemperaturasModelo

Avsl = -0.32595074


button = widgets.Button(description="Regression of Parameters")

def regresionParametros(b):
    clear_output()

    def minimizarVSL(Avsl):
        for T, P, i in zip(DatosTemperatura_Exp, DatosPresionp_Exp, posicion):
            print ("Initial Temperature = ", T, "K", "Pressure = ", P, "bar",
↪"Experimental Data = ", i+1)
            initial_temperature = T # T [=] K
            initial_pressure = P # [=] bar
            # tol=
            TemperaturasModelo[i] = fsolve(equilibrioSF,initial_temperature,
↪args=(initial_pressure, nif, 1, 2, Avsl), xtol=1e-4)

        funcionObjetivo = np.sum((DatosTemperatura_Exp - TemperaturasModelo) ** 2)
        print("modelTemperature = ", TemperaturasModelo)
        print("Objective Function = ", funcionObjetivo)

        return funcionObjetivo

    opt = sp.optimize.minimize(minimizarVSL, Avsl, method='L-BFGS-B')

    print("optimal parameter", opt)


button.on_click(regresionParametros)
```

(continues on next page)

(proviene de la página anterior)

```
page6 = widgets.Box(children=[button], padding=4)

tabs = widgets.Tab(children=[page1, page2, page3, page4, page5, page6])
#display(tabs)

tabs.set_title(0, 'Components')
tabs.set_title(1, 'Models')
tabs.set_title(2, 'Conditions')
tabs.set_title(3, 'Results')
tabs.set_title(4, 'Experimental Data')
tabs.set_title(5, 'Regression of Parameters')

#--------------------- flash Isothermal------------------------------

Componentes_f1 = widgets.SelectMultiple(
    description="Component 1",
    options=list(Etiquetas))

Componentes_f2 = widgets.SelectMultiple(
    description="Component 2",
    options=list(Etiquetas))

Componentes_f3 = widgets.SelectMultiple(
    description="Component 3",
    options=list(Etiquetas))

Componentes_f4 = widgets.SelectMultiple(
    description="Component 4",
    options=list(Etiquetas))

button = widgets.Button(description="Upload Data")

def cargarDatos(b):
    clear_output()
    print("Component 1: ", Componentes_f1.value)
    Nombre = Componentes_f1.value
    Propiedades = data2.loc[Nombre]
    Factor_Acentrico_1 = Propiedades[0]
    Temperatura_Critica_1 = Propiedades[1]
    Presion_Critica_1 = Propiedades[2]
    Z_Critico_1 = Propiedades[3]

    #print(Propiedades)
    print ("Acentric Factor = ", Factor_Acentrico_1)
    print ("Critical Temperature = ", Temperatura_Critica_1, "K")
    print ("Critical Pressure = ", Presion_Critica_1, "bar")
    print ("Z_Critical = ", Z_Critico_1, "\n")


    print("Component 2: ", Componentes_f2.value)
    Nombre = Componentes_f2.value
    Propiedades = data2.loc[Nombre]
    Factor_Acentrico_2 = Propiedades[0]
    Temperatura_Critica_2 = Propiedades[1]
    Presion_Critica_2 = Propiedades[2]
```

(continues on next page)

(proviene de la página anterior)

```
    Z_Critico_2 = Propiedades[3]

    #print(Propiedades)
    print ("Acentric Factor  = ", Factor_Acentrico_2)
    print ("Critical Temperature = ", Temperatura_Critica_2, "K")
    print ("Critical Pressure = ", Presion_Critica_2, "bar")
    print ("Z_Critical = ", Z_Critico_2, "\n")

    print("Component 3: ", Componentes_f3.value)
    Nombre = Componentes_f3.value
    Propiedades = data2.loc[Nombre]
    Factor_Acentrico_3 = Propiedades[0]
    Temperatura_Critica_3 = Propiedades[1]
    Presion_Critica_3 = Propiedades[2]
    Z_Critico_3 = Propiedades[3]

    #print(Propiedades)
    print ("Acentric Factor = ", Factor_Acentrico_3)
    print ("Critical Temperature = ", Temperatura_Critica_3, "K")
    print ("Critical Pressure = ", Presion_Critica_3, "bar")
    print ("Z_Critical = ", Z_Critico_3, "\n")


    print("Component 4: ", Componentes_f4.value)
    Nombre = Componentes_f4.value
    Propiedades = data2.loc[Nombre]
    Factor_Acentrico_4 = Propiedades[0]
    Temperatura_Critica_4 = Propiedades[1]
    Presion_Critica_4 = Propiedades[2]
    Z_Critico_4 = Propiedades[3]

    #print(Propiedades)
    print ("Acentric Factor  = ", Factor_Acentrico_4)
    print ("Critical Temperature = ", Temperatura_Critica_4, "K")
    print ("Critical Pressure = ", Presion_Critica_4, "bar")
    print ("Z_Critical = ", Z_Critico_4, "\n")

    global TcDato_f, PcDato_f, wDato_f

    TcDato_f = np.array([Temperatura_Critica_1, Temperatura_Critica_2, Temperatura_
↪Critica_3, Temperatura_Critica_4])
    PcDato_f = np.array([Presion_Critica_1, Presion_Critica_2, Presion_Critica_3,␣
↪Presion_Critica_4])
    wDato_f = np.array([Factor_Acentrico_1, Factor_Acentrico_2, Factor_Acentrico_3,␣
↪Factor_Acentrico_4])


button.on_click(cargarDatos)
#display(button)

page_f1 = widgets.VBox(children=[Componentes_f1, Componentes_f2, Componentes_f3,␣
↪Componentes_f4, button], padding=4)


#------------------ page_f2
ecuacionEstado_f = widgets.Dropdown(description='Fluid :', padding=4, options=['SRK',
↪'PR', 'RKPR'])
```

(continues on next page)

(proviene de la página anterior)

```
button = widgets.Button(description="Upload Models")

def cargarModelos(b):
    clear_output()
    global eq
    eq = ecuacionEstado.value

    print("Component 1: ", Componentes_f1.value)
    print("Component 2: ", Componentes_f2.value)
    print("Component 3: ", Componentes_f3.value)
    print("Component 4: ", Componentes_f4.value)
    print("Fluid Model : ", ecuacionEstado_f.value)


button.on_click(cargarModelos)

page_f2 = widgets.Box(children=[ecuacionEstado_f, button], padding=4)

#------------------ page_f2

#------------------ page_f3
Temp_ini_f = widgets.Text(description='Initial', padding=4, value="0.0")

Pres_ini_f = widgets.Text(description='Initial', padding=4, value="0.0")

n1_f = widgets.Text(description='Component 1', padding=4, value="0.0")
n2_f = widgets.Text(description='Component 2', padding=4, value="0.0")
n3_f = widgets.Text(description='Component 3', padding=4, value="0.0")
n4_f = widgets.Text(description='Component 4', padding=4, value="0.0")


titulo = widgets.HTML(value="<C><H1> System Conditions <H1>")
tempe_info = widgets.HTML(value="<C><H3> Temperature <H3>")
press_info = widgets.HTML(value="<C><H3> Pressure <H3>")
fluid_info = widgets.HTML(value="<C><H3> Mole fracction in the fluid <H3>")

button = widgets.Button(description="Upload Conditions")

def cargarParametros(b):
    clear_output()

    global zi_F, temperature_f, pressure_f, nif

    temperature_f = float(Temp_ini_f.value)
    pressure_f = float(Pres_ini_f.value)
    zi_F = np.array([float(n1_f.value), float(n2_f.value), float(n3_f.value),␣
↪float(n4_f.value)])
    nif = np.array([float(n1_f.value), float(n2_f.value), float(n3_f.value), float(n4_
↪f.value)])

    print("Component 1: ", Componentes_f1.value)
    print("Component 2: ", Componentes_f2.value)
    print("Component 3: ", Componentes_f3.value)
    print("Component 4: ", Componentes_f4.value)

    print("Fluid Model : ", ecuacionEstado_f.value)
```

(continues on next page)

(proviene de la página anterior)

```
    print("Temperature_f = ", temperature_f, type(temperature_f))

    print("Pressure_f = ", pressure_f, type(pressure_f))

    print("Mole fraccion component 1 = ", n1_f.value)
    print("Mole fraccion component 2 = ", n2_f.value)
    print("Mole fraccion component 3 = ", n3_f.value)
    print("Mole fraccion component 4 = ", n4_f.value)

    print("Mole fracction in the fluid = ", zi_F, type(zi_F))

    print(temperature_f, type(temperature_f))


button.on_click(cargarParametros)

page_f3 = widgets.Box(children=[titulo, tempe_info, Temp_ini_f, press_info, Pres_ini_
↪f, fluid_info, n1_f, n2_f, n3_f, n4_f, button], padding=4)
#------------------ page_f3

#------------------ page_f4

button = widgets.Button(description="Flash Calculation")

def calcularFlashPT(b):
    clear_output()
    #Tcal = fsolve(equilibrioSF,guess,args=(Pe, nnCC_1, nnCC_2), xtol=1e-4)

    #initial_temperature = [346.5] # T [=] K
    #initial_pressure = 137.9 # [=] bar
    fhid = Flash(zi_F, temperature_f, pressure_f, TcDato_f, PcDato_f, wDato_f)
    fhid.flash_ideal()


button.on_click(calcularFlashPT)
#display(button)


page_f4 = widgets.Box(children=[button], padding=4)



#------------------ page_f4
flash = widgets.Tab(children=[page_f1, page_f2, page_f3, page_f4])

flash.set_title(0, 'Components')
flash.set_title(1, 'Models')
flash.set_title(2, 'Conditions')
flash.set_title(3, 'Results')
#tabs.set_title(4, 'Experimental Data')
#tabs.set_title(5, 'Regression of Parameters')

#--------------------- GPEC ------------------------------

name_GPEC = widgets.Text(description='File name', padding=4, value=" ")
url = name_GPEC.value
```

(continues on next page)

(proviene de la página anterior)

```
titulo = widgets.HTML(value="<C><H1> Data GPEC <H1>")

button_1 = widgets.Button(description="UpData GPEC")

def upGPEC(b):
    clear_output()

    DGPEC = DataGPEC(url)
    DGPEC.leerGPEC_1()
    print ("Upload {0}".format(url))


button_1.on_click(upGPEC)

button_2 = widgets.Button(description="Vapor pressure")

def diagram_1(b):
    clear_output()
    DGPEC = DataGPEC(url)
    DGPEC.leerGPEC_1()
    DGPEC.presionVapor()

button_2.on_click(diagram_1)


button_3 = widgets.Button(description="Diagram Density-Pressure")

def diagram_2(b):
    clear_output()
    DGPEC = DataGPEC(url)
    DGPEC.leerGPEC_1()
    DGPEC.densidadPresion()


button_3.on_click(diagram_2)

page_G1 = widgets.Box(children=[titulo, name_GPEC, button_1, button_2, button_3],␣
↪padding=4)

gpec = widgets.Tab(children=[page_G1])

gpec.set_title(0, 'Upload Data')



accord = widgets.Accordion(children=[tabs, flash, gpec], width=400)
display(accord)




accord.set_title(0, 'Pure Solid-Binary Fluid')
accord.set_title(1, 'Isothermal Flash Calculation')
accord.set_title(2, 'Data GPEC')
#accord.set_title(3, 'Regression of Parameters Solid-Fluid')
#accord.set_title(4, 'Pure Fluid')
```

(continues on next page)

(proviene de la página anterior)

```
#346.5 136.9
# Lectura Juan.xlsx
```

```
url = 'Lectura Juan.xlsx'
```

CAPÍTULO 9

14. Diagrama de fases de sustancias puras

En esta sección se presenta una forma de obtener las ecuaciones necesarias para realizar el cálculo del diagrama de fases de usa sustancia pura utilizando un algoritmo de continuación (Allgower & Georg, 2003, Cismondi & Michelsen, 2007, Cismondi et. al, 2008).

La implementación de este algoritmo de solución de las ecuaciones resultantes del equilibrio de fases son implementadas como un método de la librería pyther.

9.1 9.1 Sistema de Ecuaciones

Se parte de las ecuaciones que surgen de la condición de equilibrio de fases para una sustancia pura, sin embargo, el enfoque que se utiliza corresponde a tener como variables del sistema de ecuaciones al logaritmo natural de la temperatura T y los volumenes de líquido V_l y vapor V_v. Adicionalmente, se tiene una ecuación correspondiente a la especificación de un valor de alguna de las variables del sistema de ecuaciones danto lugar a un sistema de 3 ecuaciones con la forma que se muestra a continuación:

$$F = \begin{bmatrix} ln\left(\frac{P^l(T,V^l)}{P^v(T,V^v)}\right) \\ lnf_l(T,V^l) - lnf_v(T,V^v) \\ X_S - S \end{bmatrix}$$

Por tanto la solución del sistema de ecuaciones se puede obtener como:

$$Jx \begin{bmatrix} \Delta lnT \\ \Delta lnV^l \\ \Delta lnV^v \end{bmatrix} + F = 0$$

siendo

$$\Lambda = \begin{bmatrix} \Delta lnT \\ \Delta lnV^l \\ \Delta lnV^v \end{bmatrix}$$

en donde cada elemento de la matriz Jx, salvo la última fila que son cero, tienen la siguiente forma:

$$Jx_{1,1} = T\left(\frac{\left(\frac{\partial P_x}{\partial T}\right)}{P_l} - \frac{\left(\frac{\partial P_y}{\partial T}\right)}{P_v}\right)$$

$$Jx_{1,2} = -V_l\left(\frac{\left(\frac{\partial P}{\partial V_x}\right)}{P_l}\right)$$

$$Jx_{1,3} = -V_v\left(\frac{\left(\frac{\partial P}{\partial V_y}\right)}{P_v}\right)$$

$$Jx_{2,1} = T\left(\left(\frac{\partial f^l}{\partial T}\right) - \left(\frac{\partial f^v}{\partial T}\right)\right)$$

$$Jx_{2,2} = V_l\left(\frac{\partial f^l}{\partial V_l}\right)$$

$$Jx_{2,3} = -V_v\left(\frac{\partial f^y}{\partial V_v}\right)$$

Matriz de primeras derivadas parciales

$$J_x = \begin{bmatrix} T\left(\frac{\left(\frac{\partial P_x}{\partial T}\right)}{P_l} - \frac{\left(\frac{\partial P_y}{\partial T}\right)}{P_v}\right) & -V_l\left(\frac{\left(\frac{\partial P_x}{\partial V}\right)}{P_l}\right) & -V_v\left(\frac{\left(\frac{\partial P_y}{\partial V}\right)}{P_y}\right) \\ T\left(\left(\frac{\partial f^l}{\partial T}\right) - \left(\frac{\partial f^v}{\partial T}\right)\right) & V_l\left(\frac{\partial f^l}{\partial V_l}\right) & -V_v\left(\frac{\partial f^y}{\partial V_v}\right) \\ 0 & 0 & 0 \end{bmatrix}$$

una vez que se obtiene la solución del sistema de ecuaciones planteado, se procede con un método de continuación para obtener un valor inicial de un siguiente punto partiendo de la solución previamente encontrada y de esta forma repetir el procedimiento, siguiendo la descripción que se muestra más adelante.

9.2 9.2 Descripción del algoritmo

La descripción del algoritmo es tomada de Pisoni, Gerardo Oscar (2014):

$$Jx\left(\frac{d\Lambda}{dS_{Spec}}\right) + \left(\frac{dF}{dS_{Spec}}\right) = 0$$

Donde J_x es la matriz jacobiana de la función vectorial F, Λ es el vector de variables del sistema $F = 0$, S_{Spec} es el valor asignado a una de las variables del vector Λ, $\frac{d\Lambda}{dS_{Spec}}$ es la derivada, manteniendo la condición $F = 0$, del vector de variables con respecto al parámetro S_{spec}. Observe que si $S_{spec} = \Lambda_i$, entonces $\frac{d\Lambda_i}{dS_{Spec}} = 1$. El vector $\frac{d\Lambda}{dS_{Spec}}$ es llamado "vector de sensitividades".

$\frac{\partial F}{\partial S_{Spec}}$ es la derivada parcial del vector de funciones F con respecto la variable S_{spec}.

La matriz jacobiana J_x debe ser valuada en un punto ya convergido que es solución del sistema de ecuaciones $F = 0$. Observe en los distintos sistemas de ecuaciones presentados en el capítulo 3, que sólo una componente del vector F depende explícitamente de S_{spec}. Por tanto, las componentes del vector $\frac{\partial F}{\partial S_{Spec}}$ son todas iguales a cero, excepto la que depende de S_{spec}, en esta tesis el valor de dicha componente es siempre $1 - 1$.

Conocidos J_x y $\frac{\partial F}{\partial S_{Spec}}$ es posible calcular todas las componentes del vector $\frac{d\Lambda}{dS_{Spec}}$.

Con $\frac{d\Lambda}{dS_{Spec}}$ conocido es posible predecir los valores de todas las variables del vector Λ para el siguiente punto de la "hiper-línea» que se está calculando, aplicando la siguiente ecuación:

$$\Lambda^0_{nextpoint} = \Lambda_{conve.pont} + \left(\frac{d\Lambda}{dS_{Spec}}\right)\Delta S_{Spec}$$

Aquí $\Lambda^0_{nextpoint}$ corresponde al valor inicial del vector Λ para el próximo punto a ser calculado. $\Lambda_{conve.pont}$ es el valor del vector Λ en el punto ya convergido.

Por otra parte, el vector de sensitividades $\frac{d\Lambda}{dS_{Spec}}$ provee información sobre la próxima variable que debe ser especificada en el próximo punto a ser calculado. La variable a especificar corresponderá a la componente del vector $\frac{d\Lambda}{dS_{Spec}}$ de mayor valor absoluto. Supongamos que la variable especificada para el punto convergido fue la presión P, es decir en el punto convergido $S_{spec} = P$.

9.3 9.3 Implementación del Algoritmo

A continuación se muestra la forma de utilizar la librería pyther para realizar el diagrama de fases de una sustancia pura.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import pyther as pt
```

Luego de hacer la importación de las librerías que se van a utilizar, en la función main_eos() definida por un usuario se realiza la especificación de la sustancia pura junto con el modelo de ecuación de estado y parámetros que se requieren en la función «pt.function_elv(components, Vc, Tc, Pc, omega, k, d1)» que realiza los cálculos del algoritmo que se describió previamente.

```
def main_eos():
    print("-" * 79)
    components = ["METHANE"]
    MODEL = "PR"
    specification = "constants"
    component_eos = pt.parameters_eos_constans(components, MODEL, specification)
    #print(component_eos)
    #print('-' * 79)

    methane = component_eos[component_eos.index==components]
    #print(methane)
    methane_elv = methane[["Tc", "Pc", "k", "d1"]]
    #print(methane_elv)

    Tc = np.array(methane["Tc"])
    Pc = np.array(methane["Pc"])
    Vc = np.array(methane["Vc"])
    omega = np.array(methane["Omega"])
    k = np.array(methane["k"])
    d1 = np.array(methane["d1"])

    punto_critico = np.array([Pc, Vc])

    print("Tc main = ", Tc)
    print("Pc main = ", Pc)
```

(continues on next page)

(proviene de la página anterior)

```
    print("punto critico = ", punto_critico)

    data_elv = pt.function_elv(components, Vc, Tc, Pc, omega, k, d1)
    #print(data_elv)

    return data_elv, Vc, Pc
```

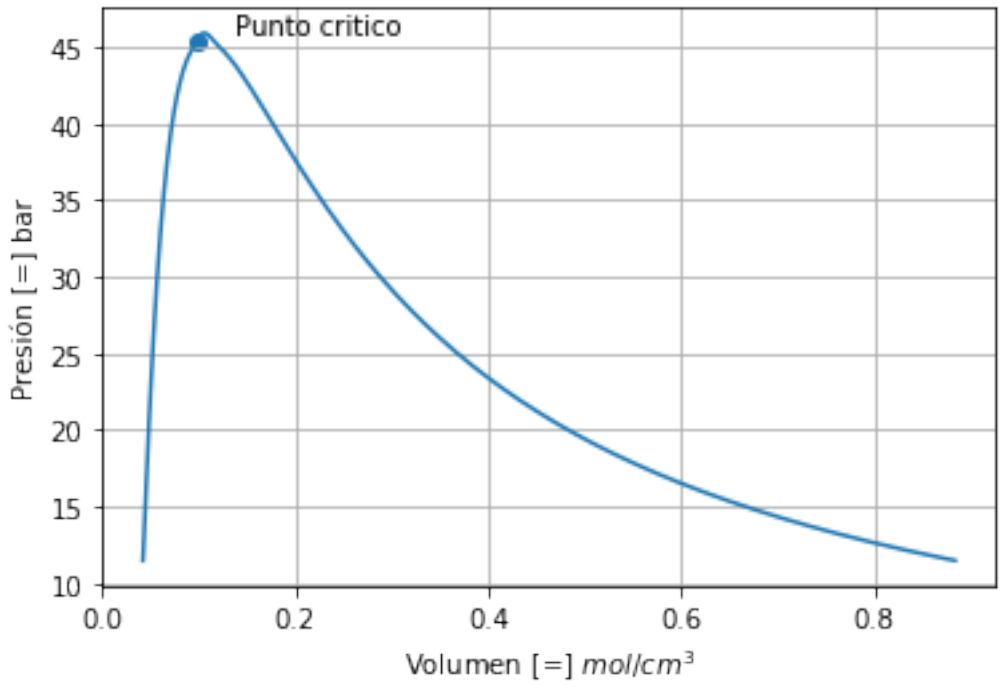
9.4 9.4 Resultados

Se obtiene el diagrama de fases líquido-vapor de una sustancia pura utilizando el método function_elv(components, Vc, Tc, Pc, omega, k, d1) de la librería pyther. Se observa que la función anterior main_eos() puede ser reemplazada por un bloque de widgets que simplifiquen la interfaz gráfica con los usuarios.

```
volumen = envolvente[0][0]
presion = envolvente[0][1]
Vc, Pc = envolvente[1], envolvente[2]

plt.plot(volumen,presion)
plt.scatter(Vc, Pc)

plt.xlabel('Volumen [=] $mol/cm^3$')
plt.ylabel('Presión [=] bar')
plt.grid(True)
plt.text(Vc * 1.4, Pc * 1.01, "Punto critico")
```



9.5 9.5 Referencias

[1] E.L. Allgower, K. Georg, Introduction to Numerical Continuation Methods, SIAM. Classics in Applied Mathematics, Philadelphia, 2003.

[2] M. Cismondi, M.L. Michelsen, Global phase equilibrium calculations: Critical lines, critical end points and liquid-liquid-vapour equilibrium in binary mixtures, Journal of Supercritical Fluids, 39 (2007) 287-295.

[3] M. Cismondi, M.L. Michelsen, M.S. Zabaloy, Automated generation of phase diagrams for binary systems with azeotropic behavior, Industrial and Engineering Chemistry Research, 47 (2008) 9728-9743.

[4] Pisoni, Gerardo Oscar (2014). Mapas Característicos del Equilibrio entre Fases para Sistemas Ternarios (tesis doctoral). Universidad Nacional del Sur, Argentina.

CAPÍTULO 10

10. Sistemas binarios

CAPÍTULO 11

11. Estabilidad Material de las Fases

A temperatura y presión constantes una mezcla de composición z será estable termodinámicamente sólo si se encuentra en su estado o configuración de menor energía de Gibbs posible para tales condiciones. Por lo tanto, ninguna separación posible de una cantidad infinitesimal de una fase de composición w podrá disminuir aun mas la energía libre del sistema. Esto puede expresarse matemáticamente en lo que se conoce como la condición del plano tangente de Gibbs, que se muestra más adelante.

La condición de igualdad de fugacidades de los componentes de una mezcla de diferentes fases solo es una de las condiciones necesarias para establecer el equilibrio termodinámico. De esta forma, una mezcla es estable a (T, P) si y solo si la energía libre de Gibbs total del sistema se encuentra en un **minimo global**. El cambio de la energía libre de Gibbs por la tranferencia de :math:‘ delta n‘ moles de un componente i de una fase líquida a una fase de vapor es:

$$\delta G = (\mu_i^v - \mu_i^l)\delta n_i$$

Por tanto, en el minimo global :math:‘ delta G‘ debe ser cero para cualquier tranferencia de materia, mantienedo el cumplimiento de la igualdad del potencial químico o fugacidad como una condición necesaria del equilibrio de fases termodiámico.

Al considerar una fase de composición z, con potencial químico :math:‘ delta (z)‘ y asumiendo que una cantidad infinitisimal :math:‘ delta e‘ de una nueva fase de composición molar w es formada, entonces elcambio en la energía libre de Gibbs del sistema está expresada como sigue:

$$\delta G = dn \sum_j w_i(\mu_i(w) - \mu_i(z)) \geq 0$$

donde una cantidad :math:‘ w_i delta e‘ del componentes i es transferido, resultando en que una condición necesaria para la estabilidad termodinámica de la fase de composición z es que :math: ‘ delta G‘ no sea negativo para cualquier :math:‘ delta e‘ positivo, expresado en la siguiente desigualdad:

$$\sum_{i=1}^{c} w_i(\mu_i(w) - \mu_i(z)) \geq 0$$

para cualquier composición w, este resultado es denominada como la condición del plano tangente de Gibbs para evaluar la estabilidad termodinámica.

11.1 11.1 Resolución de la condición de estabilidad

Para iniciar se considera una mezcla de C componentes de composición z a una temperatura T y presión P especificada, para escribir la condición suficiente de estabilidad de la mezcla como la función de la distancia del plano tangente *TPD(w)* por sus siglas en inglés:

$$TPD(w) = \sum_{i=1}^{c} w_i(\mu_i(w) - \mu_i(z)) \geq 0$$

que como se mencionó anteriormente, se exige que este sea no negativo para una composición w de una nueva fase en formación. Normalmente, conviene reescribir la TPD en terminos de la fugacidad al utilizar modelos de ecuaciones de estado (puede ser en terminos de otras variables termodinámicas) como sigue:

expresión del potencial químico a T, P y composición w

$$\mu_i(T, P, w) = \mu_i^*(T, P_o) + RTln\left(\frac{f_i(T, P, w)}{P_o}\right)$$

y reemplazando el termino de la fugacidad del componente i en la mezcla a T, P y composición w

$$\mu_i(T, P, w) = \mu_i^*(T, P_o) + RT(ln(w_i) + ln\frac{P)}{P_o} ln\phi_i(T, P, w)))$$

con lo cual se obtiene la expresión de la función de la distancia del plano tangente reducido tpd como se muestra a continuación:

$$tpd(w) = \frac{TPD(w)}{RT} = \sum_{i=1}^{C} w_i(ln(w_i) + ln(\phi_i(w)) - ln(z_i) - ln(\phi_i(z)))$$

agrupando terminos

$$tpd(w) = \frac{TPD(w)}{RT} = \sum_{i=1}^{C} w_i\left(ln(w_i) + ln(\phi_i(w)) - d_i\right)$$

donde

$$d_i = ln(z_i) + ln(\phi_i(z))$$

Por tanto, una apromaximación computacional puede ser basada en el hecho de que la condición del plano tangete es no negativa siempre, si y solo si es no negativa para todos los minimos, en ese sentido la recomendación de Michelsen & Mollurup [1]_, para implementar la evaluación de la condición del plano tangente son:

1. Localizar todos los minimos locales de la distancia del plano tangente.
2. Verificar que el valor de **tpd** es no negavita en todos los minimos. En caso de encontrar un valor negativo de **tpd** durante el procedimiento en alguno de los minimos locales de la función, la mezcla se evaluara como inestable.

11.2 11.1.1 Formas de resolver la función tpd

En primera instancia se puede mencionar los métodos de optimización para encontrar los minimos de la función tpd, sin embargo, en está sección se presentara brevemente la estrategía de expresar este problema como un problema de un sistema de ecuaciónes algebraicas no lineales.

$$tm(W) = 1 + \sum_{i}^{C} W_i(ln(W_i + ln\phi_i(W) - d_i - 1)$$

$$\frac{\partial tm}{\partial W_i} = lnW_i + ln\phi_i(W) - d_i$$

$$tm(W)^{SP} = 1 - W_T$$

$$W_T = \sum_i^C W_i$$

$$tm(W) = 1 + W_T \sum_i^C w_i(ln(W_T + lnw_i + ln\phi_i - d_i - 1)$$

$$tm(W) = (1 - W_T + W_T lnW_T) + W_T tpd(w)$$

Método de solución

$$lnW_i^{k+1} = d_i - ln\phi_i(W^k)$$

$$\phi_i(W) = \phi_i(W_i)$$

$$w_i = \frac{W_i}{W_T}$$

$$g_i = lnW_i + ln\phi_i - d_i$$

Matriz Hessiana

$$H_{ij} = \frac{\partial g_i}{\partial W_j} = \frac{1}{W_i}\sigma_{ij} + \frac{\partial ln\phi_i}{\partial W_i}$$

corrector de Newton

$$H\Delta W + g = 0$$

$$W^{k+1} = w^k \Delta W$$

CAPÍTULO 12

12. Puntos Criticos

$$n_1 = z_1 + s\sqrt{(z_1)}u_1$$

$$n_2 = z_2 + s\sqrt{(z_2)}u_2$$

$$u_1^2 + u_2^2 = 1$$

$$B_{ij} = \sqrt{(z_i z_j)} \left(\frac{\partial ln f_i}{\partial n_j} \right)_{T,V}$$

$$b = \lambda_1 = 0$$

$$c = \left(\frac{\partial \lambda_1}{\partial s} \right)_{s=0} = 0$$

$$c \simeq \frac{\lambda_1(s = \eta) - \lambda_1(s = -\eta)}{2\eta}$$

$$\eta = 0{,}0001$$

$$\lambda = \frac{-\beta \pm \sqrt{\beta^2 - 4}}{2}$$

subroutine bceval(z,T,V,P,b,c) c INPUT: z,T,V c OUTPUT: P,b,c c C The following subroutine must be included in a separate .for file: C XTVTERMO(NTYP,T,V,P,z,FUG,FUGT,FUGV,FUGN) C INPUT: C NTYP: LEVEL OF DERIVATIVES, SEE BELOW C T: TEMPERATURE (K) C V: MOLAR VOLUME (L/MOL) C z: COMPOSITION (MOLES, NEED NOT BE NORMALIZED) C OUTPUT: C P: PRESSURE (bar) C FUG: VECTOR OF LOG FUGACITIES (ALL NTYP) C FUGT: T-DERIVATIVE OF FUG (NTYP = 2, 4 OR 5) C FUGV: V-DERIVATIVE OF FUG (ALL NTYP) C FUGN: MATRIX OF COMPOSITION DERIVATIVES OF FUG (NTYP >=3)

IMPLICIT DOUBLE PRECISION (A-H,O-Z) PARAMETER (nco=2) DIMENSION z(nco) DIMENSION sqz(nco),ym(nco),u(nco),up(nco),y(nco)

COMMON /Pder/ DPDN(nco),DPDT,DPDV COMMON /CAEPcond/ DPDVcri eps=1.0D-4

1 sqz(1)=sqrt(z(1)) sqz(2)=sqrt(z(2)) call eigcalc(z,T,V,P,b,u) dpdvcri=dpdv

c calculation of b at s=eps (e) y(1)=z(1)+eps*u(1)*sqz(1) y(2)=z(2)+eps*u(2)*sqz(2) if(minval(y).lt.0)then

call modifyz(z) go to 1

end if call eigcalc(y,T,V,Pp,bpos,up)

c calculation of b at s=-eps (m) ym(1)=z(1)-eps*u(1)*sqz(1) ym(2)=z(2)-eps*u(2)*sqz(2) if(minval(ym).lt.0)then

call modifyz(z) goto 1

end if call eigcalc(ym,T,V,Pn,bneg,up)

c calculation of c c=(bpos-bneg)/2.0/eps end

c

subroutine modifyz(z)

IMPLICIT DOUBLE PRECISION (A-H,O-Z) DIMENSION z(2)

if(z(1).lt.z(2))then z(1)=2*z(1) z(2)=1.0d0-z(1)

else z(2)=2*z(2) z(1)=1.0d0-z(2)

end if end

c

subroutine eigcalc(z,T,V,P,b,u)

IMPLICIT DOUBLE PRECISION (A-H,O-Z) PARAMETER (nco=2) DIMENSION z(nco),FUG(nco),FUGT(nco),FUGV(nco),FUGN(nco,nco) DIMENSION u(nco) jac=5 ! FUGN is required, but not FLT call XTVTERMO(jac,T,V,P,z,FUG,FUGT,FUGV,FUGN)

bet=-z(1)*FUGN(1,1)-z(2)*FUGN(2,2) gam=z(1)*z(2)*(FUGN(1,1)*FUGN(2,2)-FUGN(1,2)**2) sq=sqrt(bet**2-4*gam) rlam1=(-bet+sq)/2 rlam2=(-bet-sq)/2 if(abs(rlam1).lt.abs(rlam2))then

b=rlam1

else b=rlam2

end if u2=(b-z(1)*FUGN(1,1))/(sqrt(z(1)*z(2))*FUGN(1,2)) ! k=u2/u1=u2 u(1)=sqrt(1/(1+u2*u2)) !normalization u(2)=sqrt(1-u(1)**2) if(u2.lt.0) u(2)=-u(2) end

C C purpose of routine CRITSTABCHECK: C C To find the composition where the tangent plane distance respect to the C critical composition takes on its minimum value at given T and P C C Parameters: C C T (I) Temperature C P (I) Pressure C Xc (I) Composition of the critical point C W (O) Composition of the minimum tpd C tpdm (O) Value of the minimum tpd

CAPÍTULO 13

13. Mezclas multicomponentes

CAPÍTULO 14

14. Cálculo del flash Isotermico (T, P)

Se presenta una implementación del calculo del flash isotermico bifasico utilizando la ecuación de estado *Peng-Robinsong (PR)* [2] junto con las reglas de mezclado de *Van Der Waalls* [2].

El cálculo del flash isotermico bifasico es un cálculo básico en la introducción de los procesos de separación porque es el esqeuma tecnologíco de separación más simple, en el que ingresa una corriente de fluido a un «tanque» calentado por un flujo de calor en el que se obtiene una corriente de salida por cada fase presente en el sistema. En el caso bifasico, una corriente de líquido y otra de vapor, tal como se muestra en la figura 1.

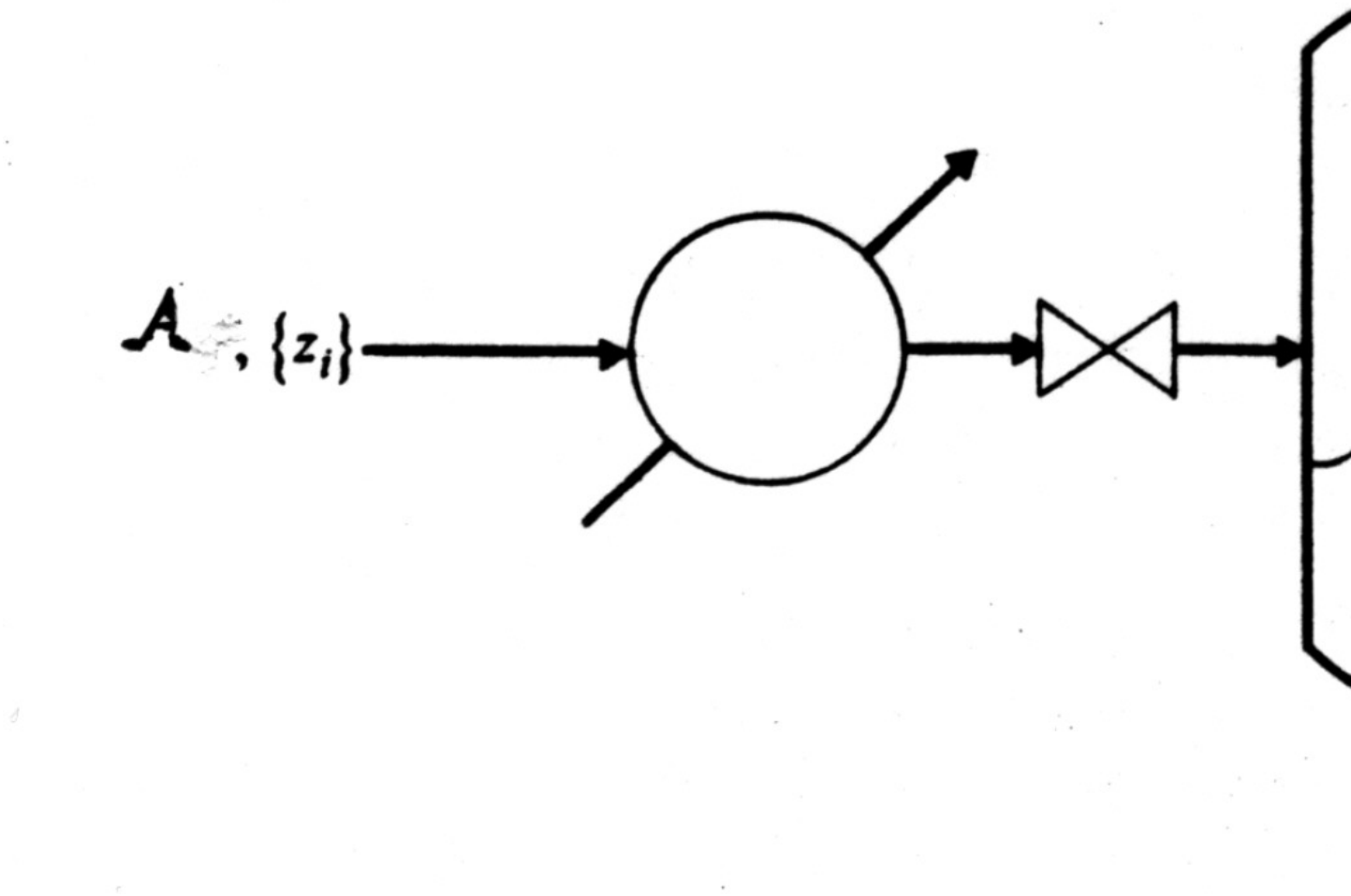


Figura 1. Esquema del cálculo del flash isotermico

14.1 14.1 Modelo flash líquido-vapor

El modelo del flash isotermico bifasico, corresponde al balance de materia global y por componente en el tanque separador que se muestra en la figura (1), junto con la condición de equilibrio de fases líquido-vapor.

Coeficiente de distribución K_i

$$Ki = \frac{yi}{xi}$$

Aproximación de wilson para el coeficiente de distribución K_i

$$lnK_i = ln\left(\frac{Pc_i}{P}\right) + 5{,}373(1+w_i)(1-\frac{Tc_i}{T})$$

Rachford-Rice $g(\beta)$

$$g(\beta) = \sum_{i=1}^{C}(y_i - x_i)$$

$$g(\beta) = \sum_{i=1}^{C}\frac{K_i - 1}{1-\beta+\beta K_i}$$

Derivada de la función *Rachford-Rice* $g(\beta)$

$$\frac{dg}{d\beta} = \sum_{i=1}^{C} z_i \frac{(K_i-1)^2}{(1-\beta+\beta K_i)^2} < 0$$

Valores límites de la función *Rachford-Rice* $g(\beta)$

$$g(0) = \sum_{i=1}^{C}(z_i K_i - 1) > 0$$

$$g(1) = \sum_{i=1}^{C}(1-\frac{z_i}{K_i}) < 0$$

Ecuaciones para calcular las fracciones molares de cada fase

$$y_i \frac{K_i z_i}{1-\beta+\beta K_i}$$

$$x_i = \frac{z_i}{1-\beta+\beta K_i}$$

Relaciones que determinan los valores mínimos y máximos para β

$$1-\beta+\beta K_i >= K_i z_i$$

$$\beta \geq \frac{K - i z_i - 1}{K_i - 1}$$

$$1-\beta+\beta K_i >= z_i$$

$$\beta \leq \frac{z_i - 1}{1 - K_i}$$

Valores extremos de la fracción de vapor en el sistema β

$$\beta_{min} = 0$$

$$\beta_{max} = 1$$

14.2 14.2 Algoritmo

- Especificar la Presión P, Temperatura T y número de moles N de cada componente del sistema
- Calcular el coeficiente de distribución K_i^{wilson} a partir de la relación de Wilson
- Calcular el valor de β_{min}
- Calcular el valor de β_{max}
- Calcular el promedio de beta, usando Beta minimo y Beta máximo
- Resolver la ecuación de *Rachford-Rice* $g(\beta)$, para calcular β con una tolerancia de $1x10^{-6}$
- Calcular las fracciones molares del líquido x_i y del vapor y_i
- Calcular los coeficientes de fugacidad $\hat{\phi}_i$ para las fracciones molares del líquido x_i y del vapor y_i
- Calcular el coeficiente de distribución K_i a partir de los coeficientes de fugacidad del componente i $\hat{\phi}_i$
- Volver a resolver la ecuación de *Rachford-Rice* $g(\beta)$, para calcular β con una tolerancia de $1x10^{-6}$
- Verificar la convergencia del sistema con una tolerancia de $1x10^{-6}$ para $\Delta K_i = \left| K_i^{j+1} - K_i^j \right|$, siendo está situación la convergencia del procedimiento.

14.3 14.2.1 Implementación

En la implementación del cálculo del flash isotermico, se tiene 3 partes importantes:

- Cálculo de los coeficientes de distribución por medio de la ecuación de Wilson
- Cálculo de los valores mínimos y máximos para la fracción β
- Cálculo del *step* para calcular la fracción β

14.3.1 Ecuación de Wilson

```
def Ki_wilson(self):
    """Equation of wilson for to calculate the Ki(T,P)"""
    variable_0 = 5.373 * (1 + self.w) * (1 - self.Tc / self.T)
    lnKi = np.log(self.Pc / self.P) + variable_0
    self.Ki = np.exp(lnKi)
    return self.Ki
```

14.3.2 Cálculo de los valores mínimos y máximos para la fracción β

```
def beta_initial(self):
    self.Ki = self.Ki_wilson()
    self.Bmin = (self.Ki * self.zi - 1) / (self.Ki - 1)
    self.Bmax = (1 - self.zi) / (1 - self.Ki)
    self.Binit = (np.max(self.Bmin) + np.min(self.Bmax)) / 2
    return self.Binit
```

14.3.3 Cálculo del *step* para calcular la fracción β

```
def beta_newton(self):
    iteration, step, tolerance = 0, 1, 1e-5
    while True:
        self.Binit = self.Binit - step * self.rachford_rice()[0] / self.rachford_
↪rice()[1]
        iteration += 1
        while self.Binit < self.Bmin or self.Binit > self.Bmax:
            step = step / 2
        if abs(self.rachford_rice()[0]) <= tolerance or (iteration >= 50):
            break
    return self.Binit
```

14.4 14.3. Resultados

A continuación se muestran los resultados numéricos del calculo del flash isotermico bifasico para una mezcla de los componentes (C3-Ci4-C4), que corresponde al cálculo del flash isotermico propuesto por (Elliott & Lira, 2012) el ejemplo 10.7 de su libro Introductory Chemical engineering thermodynamics. En la tabla 1, se presentan las especificaciones de la presión P, temperatura T y flujo F junto con las fracciones molares del líquido, del vapor y la fracción de fase resultanten usando como modelo termodinámico la ecuación de estado *Peng-robinson (PR)* y las reglas de mezclado de *Van Der Waalls*.

En la tabla 1., se presenta el resultado del cálculo del flash isotermico utilizando solo el K_i^{wilson}

Tabla.1 flash isotermico $K_i(T, P)$ Mezcla ideal

Presión Bar	Temperatura K	Flujo F mol/h
8	320	1

Componente	z_i	líquido x_i	Vapor y_i
C3	0.23	0.18357118	0.37209837
Ci4	0.67	0.70479988	0.56349276
C4	0.10	0.11162895	0.06440887

función g	derivada función $\frac{dg}{d\beta}$	β
6.1017797856749434e-07	-0.20663315922997191	0.24627123315157093

mientras que en la tabla 2, se muestra el resultado del cálculo del flash isotermico utilizando el resultado de K_i^{wilson} como valor inicial para el procedimiento del cálculo del flash isotermico incluyento el cálculo de los coeficientes de fugacidad $\hat{\phi}_i$ con la ecuación de estado PR.

Tabla.2 Flash isotermico $K_i(T, P, x_i, y_i)$ **(PR)**

función g	derivada función $\frac{dg}{d\beta}$	β
-9.7482523918959729e-06	-0.13108663002971882	0.19530673657

De esta forma, se observa que el algoritmo empleando la ecuación de estado **Peng-Robinson (PR)** converge en a una solución *cercana* de la solución que utiliza la aproximación de wilson para el coeficiente de distribución **Ki**, mostrando ser efieciente para casos simples como el presente en este capítulo.

14.5 14.3.1 Efecto de la temperatura y presión sobre β

Para el mismo sistema que se presentó en las tabla 1 y 2, en la figura 2 se muestra la solución del cálculo del flash isotermico para un rango de presión y temperatura en el cual la fracción vaporizada β varia entre 0 y 1. En este caso, al aumentar la presión β disminuye mientras que el efecto de la temperatura es el contrario.

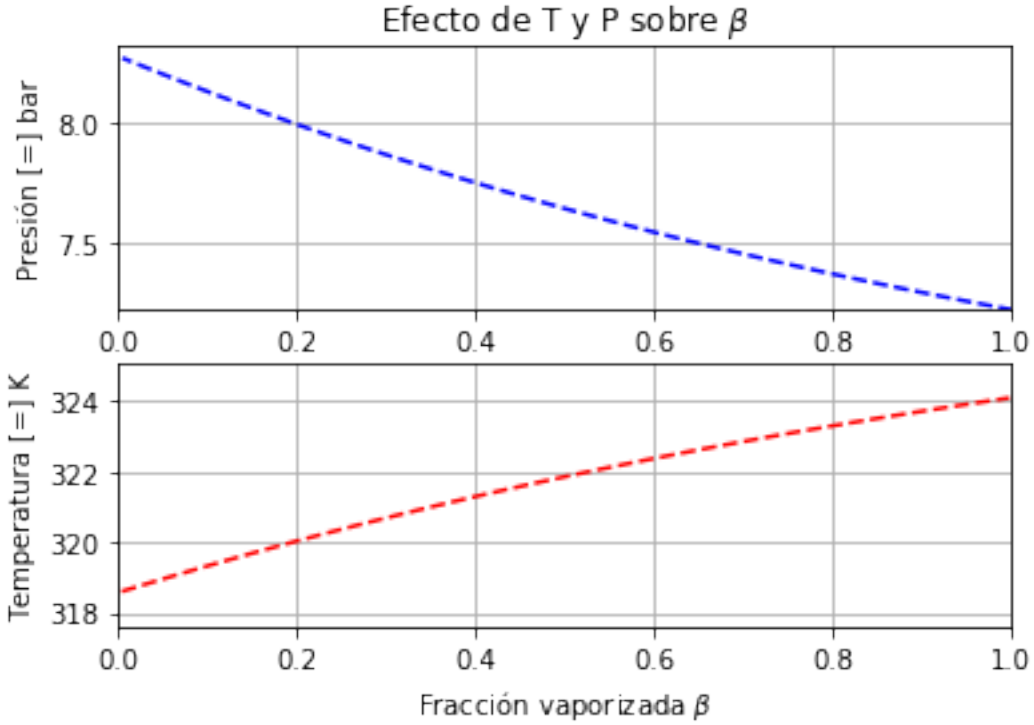


Figura 2. Efecto de la temperatura y presión sobre β

14.6 14.4 Conclusiones

- Se implemento el cálculo del flash isotermico bifasico utilizando la ecuación de estado *Peng-Robinsong (PR)* tomando las recomendaciones planteadas en el curso de termodinámica de fluidos para mejorar la convergencia del cálculo.
- Se encontró que se utilizan en promedio 3 iteraciones para calcular el valor β en cada paso que se mantienen constantes los valores K_i.

14.7 14.5 Referencias

1. Curso de especialización en Termodinámica de fluidos. Ph.D Martín Cismondí. Marzo-Junio (2017)
2. Introductory Chemical engineering thermodynamics. J. Richard Elliott , Carl T. Lira. Prentice Hall (2012)

CAPÍTULO 15

15. Modelos para la energía de gibbs de Exceso

Por lo regular G^E/RT es una función de T, P y de la composición, aunque para líqudios a presiones de bajas a moderadas es una función muy débil de P. Por tanto, es usualmente despreciada la dependencia de la presión de los coneficientes de actividad. En estos términos, para los datos a T constante:

$$\frac{G^E}{RT} = g(x_1, x_2, ..., x_N)$$

a T constante

La ecuación de Margules, es un ejemplo de dicha funcionalidad.

Un número de otras ecuaciones son de uso común para la correlación de los coeficientes de actividad. En los sitemas binarios (especies 1 y 2), la función representada con mayor frecuencia por una ecuación es G^E/x_1x_2RT, la cual es factible expresar como una series de potencias en x_1

$$\frac{G^E}{x_1x_2RT} = a + bx + x_i^2 + ...$$

a T constante

Puesto que $x_2 = 1 - x_1$, la fracción mol $x_1$sirve como la única variable independiente. una serie de potencias equivalentes con ciertas ventajas se conoce como la expansión de Redlisc/Kister

$$\frac{G^E}{RT} = A + B(x_1 - x_2) + C(x_1 - x_2)^2 + ...$$

a T constante

en su aplicación son apropiados diversor truncamientso de esta serie y en cada caso las expresiones especificas para $ln\gamma_1$ y $ln\gamma_2$ se genera con la ecuación

$$ln\gamma_i = \left(\frac{\partial nG^E/RT}{\partial n_i}\right)_{P,T,n_j}$$

Cuando A=B=C=. . . =0, G^R/RT=0, ln _i=0 y la solución es ideal.

Si B = C = . . . = , entonces

$$\frac{G^E}{x_1 x_2 RT} = A$$

donde A es una constante para una termperatura dada. Las ecuaciones correspondientes para $ln\gamma_1$ y $ln\gamma_2$ son:

$$ln\gamma_1 = Ax_2^2$$

$$ln\gamma_2 = Ax_1^2$$

Es evidente la naturaleza simetríca de estas relaciones. Los

```
# -*- coding: utf-8 -*-
import numpy as np
```

```
def cal_NRTL(nC, T, Xi, Alfa, Aij):

    #-----------------------------------------------------------------------
    s = (len(Xi),len(Xi))
    Tao = G = np.zeros(s)
    Tao = Aij / T
    G = np.exp(-Alfa * Tao)

    print ("\n", "Esta es la Matriz Tao = {0}".format(Tao), "\n")
    print ("Esta es la Matriz G = {0}".format(G), "\n")
    #-----------------------------------------------------------------------
    suma_1 = np.ones([nC, nC], dtype=np.float32)
    suma_2 = np.ones([nC, nC], dtype=np.float32)
    suma_11 = np.zeros([0, nC], dtype=np.float32)
    suma_12 = np.zeros([0, nC], dtype=np.float32)
    #-----------------------------------------------------------------------
    for j in range(nC):
        for i in range(nC):
            suma_1[i, j] = Tao[i, j] * G[i, j] * Xi[i]
            suma_2[i, j] = G[i, j] * Xi[i]

    suma_11 = suma_1.sum(axis=0)
    suma_12 = suma_2.sum(axis=0)
    #-----------------------------------------------------------------------
    print ("Esta es la Matriz suma1 = {0}".format(suma_1), "\n")
    print ("Esta es la Matriz suma2 = {0}".format(suma_2), "\n")
    print ("Esta es la Matriz suma11 = {0}".format(suma_11), "\n")
    print ("Esta es la Matriz suma12 = {0}".format(suma_12), "\n")
    #-----------------------------------------------------------------------
    A = suma_11 / suma_12
    print ("miremos la matriz A = {0}".format(A), "\n")
    #-----------------------------------------------------------------------
    num1 = np.zeros([nC, nC], dtype=np.float32)
    den1 = np.zeros([nC, nC], dtype=np.float32)
    num2 = np.zeros([nC, nC], dtype=np.float32)
    den2 = np.zeros([nC, nC], dtype=np.float32)

    for j in range(nC):
        for i in range(nC):
            num1[i, j] = G[j, i] * Xi[i]
            den1[i, j] = G[i, j] * Xi[i]
            num2[i, j] = Tao[i, j] * G[i, j] * Xi[i]
```

(continues on next page)

(proviene de la página anterior)

```
            den2[i, j] = G[i, j] * Xi[i]

    print ("Esta es la Matriz num1 = {0}".format(num1), "\n")
    print ("Esta es la Matriz den1 = {0}".format(den1), "\n")
    print ("Esta es la Matriz num2 = {0}".format(num2), "\n")
    print ("Esta es la Matriz den2 = {0}".format(den2), "\n")
    #-------------------------------------------------------------------------
    Z = np.zeros([nC, 1], dtype=np.float32)
    W = np.zeros([nC, 1], dtype=np.float32)
    lnGamma = np.zeros([nC, 1], dtype=np.float32)
    ln = np.zeros([nC, nC], dtype=np.float32)

    for i in range(nC):
        Z[i, 0] = np.sum(den1[:, i])
        W[i, 0] = np.sum(num2[:, i])

    for j in range(nC):
        for i in range(nC):
            ln[i, j] = num1[i, j] / Z[i, 0] * (Tao[j, i] - W[i, 0] / Z[i, 0])
    print ("Esta es la Matriz ln = {0}".format(ln))
    #-------------------------------------------------------------------------
    for i in range(nC):
        lnGamma[i, 0] = A[i] + sum(ln[:, i])

    gamma_i = np.exp(lnGamma)

    print ("Esta es la Matriz Z = {0}".format(Z), "\n")
    print ("Esta es la Matriz W = {0}".format(W), "\n")
    print ("Esta es la Matriz ln = {0}".format(ln), "\n")
    print ("Esta es la Matriz lnGamma = {0}".format(lnGamma), "\n")
    print ("Esta es la Matriz gamma_i = {0}".format(gamma_i), "\n")
    #-------------------------------------------------------------------------
    return gamma_i
```

```
import numpy as np
#import NRTL_3
#-------------------------------------------------------------------------
## Definiciones
#-------------------------------------------------------------------------

# nC: Numero de componenetes de la mezcla
# T = Temperatura en K
# Xi = np.matrix([0.25, 0.25, 0.25, 0.25])
# Alfa =
# Aij =

#-------------------------------------------------------------------------
#                 Alcohol  Agua     Acetato    Acido
Alfa = np.array([[0.000, 0.2980, 0.3009, 0.1695],
                 [.2980, 0.0000, 0.2000, 0.2987],
                 [0.3009, 0.2000, 0.0000, 0.2000],
                 [0.1695, 0.2987, 0.2000, 0.0000]])
#-------------------------------------------------------------------------
#             Alcohol  Acetato  Agua       Acido
Aij = np.array([[0.0000, 100.1, -144.8, 178.3],
                [1447.5, 0.0000, 2221.5, 424.018],
```

(continues on next page)

(proviene de la página anterior)

```
                [320.6521, 254.47, 0.0000, 214.55],
                [-316.8, -110.57, -37.943, 0.000]])
#-------------------------------------------------------------------------
nC = 4
T = 300.0
#-------------------------------------------------------------------------
Xi_1 = float(eval(input("Fraccion molar 1: ")))
Xi_2 = float(eval(input("Fraccion molar 2: ")))
Xi_3 = float(eval(input("Fraccion molar 3: ")))
Xi_4 = float(eval(input("Fraccion molar 4: ")))
#-------------------------------------------------------------------------
Xi = np.array([Xi_1, Xi_2, Xi_3, Xi_4])
sumar_Xi = sum(Xi)
Xi = Xi / sumar_Xi
#-------------------------------------------------------------------------

print ("\n", "Composición Xi = {0}".format(Xi),"\n")
print ("Matriz Alfa = {0}".format(Alfa), "\n")
print ("Matriz Aij = {0}".format(Aij), "\n")

#-------------------------------------------------------------------------

#CoeAct_1 = NRTL_3.NRTL(nC, T, Xi, Alfa, Aij)
coeficientes_actividad = cal_NRTL(nC, T, Xi, Alfa, Aij)
```

```
Fraccion molar 1: 0.2
Fraccion molar 2: 0.2
Fraccion molar 3: 0.3
Fraccion molar 4: 0.3

 Composición Xi = [ 0.2  0.2  0.3  0.3]

Matriz Alfa = [[ 0.      0.298   0.3009  0.1695]
 [ 0.298   0.      0.2     0.2987]
 [ 0.3009  0.2     0.      0.2   ]
 [ 0.1695  0.2987  0.2     0.    ]]

Matriz Aij = [[    0.       100.1     -144.8      178.3   ]
 [ 1447.5        0.      2221.5      424.018 ]
 [  320.6521   254.47       0.       214.55  ]
 [ -316.8     -110.57     -37.943      0.    ]]


 Esta es la Matriz Tao = [[ 0.          0.33366667 -0.48266667  0.59433333]
 [ 4.825       0.          7.405       1.41339333]
 [ 1.06884033  0.84823333  0.          0.71516667]
 [-1.056      -0.36856667 -0.12647667  0.        ]]

Esta es la Matriz G = [[ 1.          0.90535091  1.15631058  0.90416854]
 [ 0.2374377   1.          0.22741016  0.65561563]
 [ 0.72497794  0.84396296  1.          0.86672518]
 [ 1.19601118  1.1163795   1.02561797  1.        ]]

Esta es la Matriz suma1 = [[ 0.          0.06041708 -0.11162251  0.1074755 ]
 [ 0.22912738  0.          0.33679447  0.18532856]
 [ 0.2324657   0.21476325  0.          0.18595588]
```

(continues on next page)

(proviene de la página anterior)

```
 [-0.37889633 -0.12343808 -0.03891502  0.        ]]

Esta es la Matriz suma2 = [[ 0.2         0.18107018  0.23126212  0.18083371]
 [ 0.04748754  0.2         0.04548203  0.13112313]
 [ 0.21749339  0.25318888  0.30000001  0.26001754]
 [ 0.35880336  0.33491385  0.30768541  0.30000001]]

Esta es la Matriz suma11 = [ 0.08269677  0.15174225  0.18625693  0.47875994]

Esta es la Matriz suma12 = [ 0.82378429  0.96917295  0.88442957  0.87197441]

miremos la matriz A = [ 0.10038643  0.15656881  0.21059555  0.54905277]

Esta es la Matriz num1 = [[ 0.2         0.04748754  0.14499559  0.23920223]
 [ 0.18107018  0.2         0.16879259  0.2232759 ]
 [ 0.34689316  0.06822305  0.30000001  0.30768541]
 [ 0.27125058  0.19668469  0.26001754  0.30000001]]

Esta es la Matriz den1 = [[ 0.2         0.18107018  0.23126212  0.18083371]
 [ 0.04748754  0.2         0.04548203  0.13112313]
 [ 0.21749339  0.25318888  0.30000001  0.26001754]
 [ 0.35880336  0.33491385  0.30768541  0.30000001]]

Esta es la Matriz num2 = [[ 0.          0.06041708 -0.11162251  0.1074755 ]
 [ 0.22912738  0.          0.33679447  0.18532856]
 [ 0.2324657   0.21476325  0.          0.18595588]
 [-0.37889633 -0.12343808 -0.03891502  0.        ]]

Esta es la Matriz den2 = [[ 0.2         0.18107018  0.23126212  0.18083371]
 [ 0.04748754  0.2         0.04548203  0.13112313]
 [ 0.21749339  0.25318888  0.30000001  0.26001754]
 [ 0.35880336  0.33491385  0.30768541  0.30000001]]

Esta es la Matriz ln = [[-0.02437202  0.2723532   0.17045911 -0.33577991]
 [ 0.03308712 -0.03230978  0.12046131 -0.12097954]
 [-0.27191302  0.55496132 -0.07143436 -0.11726451]
 [ 0.01408571  0.19496277  0.04953417 -0.18889984]]
Esta es la Matriz Z = [[ 0.82378429]
 [ 0.96917295]
 [ 0.88442957]
 [ 0.87197441]]

Esta es la Matriz W = [[ 0.08269677]
 [ 0.15174225]
 [ 0.18625693]
 [ 0.47875994]]

Esta es la Matriz ln = [[-0.02437202  0.2723532   0.17045911 -0.33577991]
 [ 0.03308712 -0.03230978  0.12046131 -0.12097954]
 [-0.27191302  0.55496132 -0.07143436 -0.11726451]
 [ 0.01408571  0.19496277  0.04953417 -0.18889984]]

Esta es la Matriz lnGamma = [[-0.14872578]
 [ 1.14653635]
 [ 0.47961578]
 [-0.21387103]]
```

(continues on next page)

(proviene de la página anterior)

```
Esta es la Matriz gamma_i = [[ 0.86180538]
 [ 3.14727306]
 [ 1.6154536 ]
 [ 0.8074525 ]]
```

15.1 15.3 Modelos para la energía de gibbs de Exceso: UNIFAC

Ejemplo de implementación del modelo de actividad UNIFAC

```
U  = 5;

m = 2;     #Este es el número de moléculas en la mezcla
#g = 3;   #Este es el número de grupos funcionales en la mezcla
g = 7;
#T = 331.15  # K
#T = 328
#     Etanol - n-Hexano
#xj = [0.332 , 0.668]
#xj = [0.383 , 0.617]
##################################################


# Agua - Isoamil alcohol - ácido acético
#     H2O CH3 CH2 CH  OH  COOH  COOCH3
v1 = [1   0   0   0   0   0     0]'; # Agua
v2 = [0   2   2   1   0   0     1]'; # Isoamil acetato
v3 = [0   1   0   0   0   1     0]'; # Ácido acético

v = [v1' ; v2' ; v3']';

v = [v1' ; v3']';
##################################################

# Agua - Isoamil acetato - ácido acético
#     H2O     CH3     CH2     CH      OH     COOH    COOCH3
R = [0.9200 0.9011 0.6744 0.4469 1.0000 1.3013  1.9031]';
Q = [1.4000 0.8480 0.5400 0.2280 1.2000 1.2240  1.7280]';
##################################################

# Agua - Isoamil alcohol - Ácido acético

#    H2O      CH3      CH2      CH       OH       COOH     COOCH3
a = [0        300      300      300     -229.1   -14.09    72.8700;...
     1318     0        0        0        986.5    663.5    232.100;...
     1318     0        0        0        986.5    663.5    232.100;...
     1318     0        0        0        986.5    663.5    232.100;...
     353.5    156.4    156.4    156.4    0        199      101.100;...
    -66.17    315.3    315.3    315.3   -151      0       -256.300;...
     200.800 114.800 114.800 114.800 245.400 660.200 0       ];
##################################################

A = exp(-a./T);

##################################################
```

(continues on next page)

(proviene de la página anterior)

```
for j = 1 : 1 : m

   r(:,j) = sum(R.*v(:,j));

end

r;
#####################################################

for j = 1 : 1 : m

   q(:,j) = sum(Q.*v(:,j));

end

q;
#####################################################

for j = 1 : 1 : m

   J(:,j) = r(1,j)*xj(1,j)/sum(r.*xj);

end

J;
#####################################################

for j = 1 : 1 : m

   L(1,j) = q(1,j)*xj(1,j)/sum(q.*xj);

end

L;
#####################################################

li = 5.*(r - q) - (r - 1);

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

lnYCi = log(J./xj) + 5.*q.*log(L./J) + li - (J./xj).*(sum(xj.*li));

#lnY1C = log(J(1,1)/xj(1,1)) + 5*q(1,1)*log(L(1,1)/J(1,1)) + li(1,1) - (J(1,1)/xj(1,
↪1))*(xj(1,1)*li(1,1) + xj(1,2)*li(1,2))


#lnY2C = log(J(1,2)/xj(1,2)) + 5*q(1,2)*log(L(1,2)/J(1,2)) + li(1,2) - (J(1,2)/xj(1,
↪2))*(xj(1,1)*li(1,1) + xj(1,2)*li(1,2))
#####################################################


# Coeficiente de actividad residual del grupo (k)
# en la molecula (i) ####################
# Fracción molar del grupo funcional (k)
# en la molecula (i)
#####################################################
```

(continues on next page)

(proviene de la página anterior)

```
#######################################################
for i = 1 : 1 : m       #Molécula (i)

   for k = 1 : 1 : g    #Grupo funcional (k)

      xg(k,i) = v(k,i)./sum(v(:,i));

   end

end

xg;

#######################################################
for i = 1 : 1 : m       #Molécula (i)

   for k = 1 : 1 : g    #Grupo funcional (k)

      Lg(k,i) = Q(k,1)*xg(k,i)/sum(Q.*xg(:,i));

   end

end

Lg;

#mor
#######################################################
for i = 1 : 1 : m       #Molécula (i)

   for k = 1 : 1 : g    #Grupo funcional (k)

      ST(k,i) = sum(Lg(:,i).*A(:,k));

   end

end

ST = ST';

#######################################################
for i = 1 : 1 : m       #Molécula (i)

   for k = 1 : 1 : g    #Grupo funcional (k)

      if i == 1
         STa(k,:) = (Lg(:,i)'.*A(k,:));
      elseif i == 2
         STa(k + g,:) = (Lg(:,i)'.*A(k,:));
      elseif i == 3
         STa(k + 2*g,:) = (Lg(:,i)'.*A(k,:));
      end

   end

end
```

(continues on next page)

(proviene de la página anterior)

```
STa;

#######################################################
for i = 1 : 1 : m      #Molécula (i)

   for k = 1 : 1 : g   #Grupo funcional (k)

      if i == 1
         lnTg(i,k) = Q(k,1).*(1 - log(ST(i,k)) - sum(STa(k,:)./ST(i,:)));
      elseif i == 2
         lnTg(i,k) = Q(k,1).*(1 - log(ST(i,k)) - sum(STa(k+g,:)./ST(i,:)));
      elseif i == 3
         lnTg(i,k) = Q(k,1).*(1 - log(ST(i,k)) - sum(STa(k+2*g,:)./ST(i,:)));
      end

   end

end
#lnT(1,k) = Q(k,1).*(1 - log(STg(1,k)) - sum(STga(k,:)./STg));


lnTg;
#######################################################


#mor
#######################################################


# Coeficiente de actividad residual del grupo (k)
# en la mezcla ####################
# Fracción molar del grupo funcional (k)
# en la mezcla
#######################################################

for i = 1 : 1 : m      #Molécula (i)

   STq(:,i) = sum(v(:,i)*xj(:,i));

end

STq = sum(STq);


#######################################################
for k = 1 : 1 : g   #Grupo funcional (k)

   xs(k,:) = (sum(v(k,:).*xj))./(STq);

end

xs;
#######################################################

for k = 1 : 1 : g   #Grupo funcional (k)
```

(continues on next page)

(proviene de la página anterior)

```
   Lgs(k,1) = Q(k,1)*xs(k,1)/sum(Q.*xs);

end

Lgs;
#########################################################



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

for k = 1 : 1 : g   #Grupo funcional (k)

   STg(k,:) = sum(Lgs.*A(:,k));

end

STg = STg';

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

for k = 1 : 1 : g   #Grupo funcional (k)

   STga(k,:) = (Lgs'.*A(k,:));

end

STga;

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

for k = 1 : 1 : g   #Grupo funcional (k)

lnT(1,k) = Q(k,1).*(1 - log(STg(1,k)) - sum(STga(k,:)./STg));

end


lnT;
#####################################################
#Coeficiente de actividad Residual

for i = 1 : 1 : m      #Molécula (i)

   lnYRi(:,i) = sum(v(:,i).*(lnT' - lnTg(i,:)'));



lnYRi;


#Coeficiente de actividad total
lnYi = lnYCi + lnYRi
Yi = np.exp(lnYi)
```

- **Bibliografía**

CAPÍTULO 16

16. Análisis computacíonal de consistencia termodinámica

1. Salazar * , M. Cismondí

Resumen. En este trabajo se presenta la herramienta PyTher, la cual se enfoca en cálculos termodinámicos del comportamiento de fases a través de la plataforma Jupyter para realizar el análisis computacional de la consistencia termodinámica de datos experimentales entre fases líquido- vapor, permitiendo una manipulación eficiente de los datos experimentales para determinar su calidad de forma programática e interactiva.

Palabras clave: PyTher, Termodinámica computacional, consistencia termodinámica, Python, Análisis de datos.

12.1 **Introducción**

Desde hace bastante tiempo se viene trabajando en la generación, recopilación y procesamiento de los datos en el ámbito científico en distintas áreas de forma programática, sin embargo, desde hace 20 años se presenta un crecimiento dramático de datos que incluyen datos experimentales científicos reportados en literatura especializada de dominio público (Frenkel, 2013), que sirve como punto de partida para otras investigaciones, por ejemplo las involucradas en moldeamiento, simulación y optimización. Por tanto, en el campo de la termodinámica los datos referidos a las áreas de termofísica y termoquímica que son una fuente importante de datos tanto para la investigación científica en áreas fundamentales como el desarrollo de tecnología y aplicaciones en nuevos diseños de procesos y productos. Recientemente el Thermodynamics Research Center (TRC) del US National Institute of Standards and Technology (NIST), publico estadísticas referentes al crecimiento de los datos de propiedades termofísicas y termoquímicas (Frenkel , 2015), reportadas en las 5 principales revistas especializadas en esta temática (Journal of Chemical and Engineering Data, The Journal of Chemical Thermodynamics, Fluid Phase Equilibria, Thermochimica Acta, and the International Journal of Thermophysics), mostrando que la cantidad de datos se ha duplicado en los últimos 10 años y viene presentando un crecimiento anual del 7 % en el volumen de datos reportados. Esto se debe entre varias cosas, por el aumento en la eficiencia y capacidad tecnológica de la medición de datos experimentales de propiedades termofísicas y termoquímicas junto con la automatización de sistemas de control y adquisición de datos para la medición de presión, temperatura, concentración entre otras variables, lo que resulta en un aumento en la productividad en la adquisición de datos, sin embargo, este aumento de productividad no ha venido acompañada con el aumento de la capacidad de evaluar la "calidad" de los datos medidos y reportados en la literatura especializada, debido a que los equipos comerciales que tradicionalmente son empleados para realizar las mediciones han sido desarrollados sin involucrar suficientemente personal altamente calificado en cada temática específica, además del uso de software que utiliza metadatos para completar de forma "engañosa" la información de propiedades termodinámicas (Frenkel , 2015), que además tiene un

factor agravante que es la dificultad de la adecuada verificación por parte de los pares evaluadores de la gran cantidad de artículos presentados para su publicación con un tiempo insuficiente para corroborar la calidad de los datos experimentales reportados (Chirico et al, 2013; Frenkel et al, 2006).

En este trabajo se presenta la herramienta PyTher para el procesamiento y visualización de datos experimentales del equilibrio de fases líquido-vapor, la cual se basa en la tecnología de la plataforma IPython que en su tercera versión recibe el nombre de Jupyter. Esta plataforma se desarrolla bajo el concepto del "peper ejecutable" (Pérez and Granger, 2007; Pérez, 2013), puesto que frecuentemente en el desarrollo de una investigación científica actual se requiere de la computación, procesamiento, visualización y presentación de una gran cantidad de información y datos que habitualmente se realiza con diferentes herramientas computacionales que no siempre están adaptadas para funcionar juntas lo que implica un esfuerzo considerable, tener que enfocarse en llevar datos de un formato a otro para poder avanzar en el procesamiento, que principio no hace parte del objetivo de la investigación científica que se está realizando, resultando en un proceso improductivo por el costo de tiempo que involucra la manipulación de herramientas de cálculo científico tradicionalmente implementado en lenguajes como FORTRAN, el cual es limitado para el procesamiento y visualización de grandes cantidades de datos (M. Gaitan et al. 2012).

2. **Consistencia termodinámica**

En esta sección se presenta la manipulación de las ecuaciones para determinar el valor de la derivada parcial de la temperatura con respecto a la fracción molar de un componente en la fase líquida a presión constante, según la definición de la Ec. (1).

CAPÍTULO 17

17. Curso de postgrado: Termodinámica de fluidos

17.1 17.1 Contenido del curso

1. Introducción y presentación del curso.
2. Comportamientos de fases. De sustancias puras a sistemas binarios.
3. Sistemas ternarios. Sistemas multicomponentes.
4. Práctica: con software GPEC y software Fluids o Sur.
5. Ecuaciones generales del equilibrio entre fases. Estabilidad de fases.
6. Celdas de equilibrio. Métodos sistéticos.
7. Métodos analíticos. medición de equilibrios a alta presión.
8. Práctica: Visita a Planta Piloto y equipos.
9. Ecuaciones de estado.
10. Reglas de mezclado y combinación. Traslación de volumen.
11. Contribución grupal (GC-EOS). Ecuaciones con término asociativo.
12. Cálculo de fugacidades. Algoritmo para flash bifásico.
13. Puntos de saturación y construcción de envolventes de fases.
14. Métodos de continuación más allá de las envolventes.
15. Práctica: Implementación de algortimo.
16. Cálculos para sustancias puras.
17. Algoritmo y métodos de cálculo detrás de GPEC(sistemas binarios)
18. Métodos de cálculo para sistemas ternarios.

- Examen: Primer Examen Parcial.

19. Parametrización de compuestos puros.
20. Parametrización de interacciones binarias. enfoques y algoritmos.
21. Práctica: Ajuste de sistemas binarios a elección.
22. Fluidos de reservorio. Clasificación. ensayos PVT.
23. Fluidos sintéticos y modelados con EOS cúbicas. Caracterización.
24. Simulación PVT. aseguramiento de flujo: Parafinas y Asfaltenos.
25. Modelos de GE: Empíricos, van Laar, Soluciones regulares.
26. Consistencia termodinámica. Composiciones Locales: Modelos.
27. Equilibrios Sólido-Fluido: Sustancias puras y sistemas binarios.
28. Enfoques de modelado para multicomponentes.
29. Termodinámica de soluciones de polímeros. Regresión de datos.
30. Biorefinerías. Equilibrio entre fases en el procesamiento de biomasa.
31. Aplicaciones de A-UNIFAC y GCA-EoS en biodiesel y fitoquímicos.
32. Práctica: Utilización de programas.

- Examen: Segundo Examen Parcial.

17.2 17.2 Información

Horario de Clases: Miercoles 10h - 12 h y 13h - 15h en el anfiteatro A de la FCEFyN de la UNC. Marzo 29 - Julio 26 de 2017.

Modalidad: Guias de Problemas - Prácticas con computación.

Evaluación: Los alumnos rendirán 2 examenes.

Profesores: Martín Cismondi, Nicolas Gañan, Gerardo Pisoni, Alfonsina Andreatta, Belén Rodriguez, Juan Ramello.

CAPÍTULO 18

Indices and tables

- genindex
- modindex
- search