Escuela Politécnica Superior

Universidad De Sevilla

Trabajo Fin de Grado

Doble Grado en Ingeniería Mecánica e Ingeniería Eléctrica

USO DE PYTHON PARA LA RESOLUCIÓN DEL PROBLEMA DE FLUJO DE POTENCIA MEDIANTE COORDENADAS CÓNICAS

Autor:

Óscar López López

Tutor:

Álvaro Rodríguez del Nozal

RESUMEN

El presente trabajo aborda la resolución del problema de flujo de potencia en redes eléctricas de distribución radiales mediante el uso de programación cónica de segundo orden. Se desarrolla un modelo que reformula las ecuaciones no lineales en un sistema convexo, garantizando una solución eficiente y estable. La implementación en Python, utilizando programación orientada a objetos, permite la creación de un programa generalizado que se adapta a cualquier red eléctrica, simulando redes de 4, 33 y 69 nodos como ejemplos. Los resultados muestran la adecuación de la formulación usada para la resolución de este tipo de problemas.

ÍNDICE

1. MOTIVACIÓN E INTRODUCCIÓN

1.1. Justificación del tema

Las redes de distribución en baja tensión, tradicionalmente, son operadas con poca monitorización, siendo incluso nula en algunos casos. En su mayoría, se tratan de sistemas sobredimensionados, donde la potencia demandada por los usuarios es muy inferior a la potencia nominal instalada en los centros de transformación. Es por ello que las distribuidoras se basan en el número de clientes conectados a la red para fijar la toma del transformador del centro de transformación a partir de la estimación de la energía demandada. Al fijar la toma del transformador en un valor adecuado, se asegura que la tensión se mantenga dentro de unos rangos admisibles.

En el presente, debido al auge de la instalación de recursos de energía renovable distribuidos en las redes de baja tensión, han aparecido retos adicionales en la operación de estas redes. Principalmente, el problema asociado a este hecho es la sobretensión sufrida en las horas punta en las redes con una alta integración de plantas fotovoltaicas residenciales. Si el operador de la red de distribución modifica la toma del transformador con el fin de disminuir el valor de la tensión para solventar este problema, puede darse el escenario en el que la red sufra subtensiones en las últimas horas del día donde la demanda es notoriamente superior y la generación de energía renovable es prácticamente nula.

1.2. Objetivos del trabajo

Como punto de partida para abordar este problema, este proyecto desarrolla una herramienta para la resolución eficiente del problema de flujo de potencia. Esta herramienta es la base para posteriores desarrollos en el análisis de la problemática propuesta mediante la formulación de flujos de potencia que permitan gestionar de manera óptima la red de distribución radial.

Se propone el uso de la programación cónica de segundo orden, enfoque que permite convertir un problema basado en coordenadas cartesianas con ecuaciones no lineales ni convexas en un problema basado en coordenadas cónicas, con

ecuaciones no lineales pero convexas, lo que garantiza que haya solución. Esto se logra mediante un cambio de variables.

Se busca que el problema sea convexo para garantizar que cualquier solución local óptima sea también una solución global óptima. Esto simplifica radicalmente el análisis y la resolución, ya que elimina la necesidad de evaluar múltiples puntos para verificar si son óptimos. Además, los problemas convexos resultan menos sensibles a errores numéricos en comparación con los no convexos. Un problema compuesto por un sistema de ecuaciones no lineales no será convexo si las ecuaciones que definen el espacio de soluciones no lo son. Por ejemplo, definiendo el siguiente sistema de ecuaciones no lineales:

$$x^2 + y^2 - 4 = 0\ ,$$

$$x \cdot y - 1 = 0\ ,$$

El sistema define un conjunto de puntos en R^2, definido como el espacio bidimensional (x, y) de números reales, pero este conjunto no es convexo, ya que el espacio de soluciones no forma una región continua. Además, no cumplen con la propiedad de convexidad, la cual expone que, para cualquier par de puntos dentro del conjunto, todos los puntos en línea recta que los une también pertenecen al conjunto. Para que el sistema sea convexo, habría que reformular las ecuaciones o restringir el espacio de soluciones a un subconjunto de R^2. Como una solución posible, se podría añadir la restricción $x \cdot y > 0$, eliminando las partes negativas del espacio, logrando un subconjunto más manejable y robusto numéricamente [1].

La finalidad de este proyecto es establecer un punto de partida válido para el análisis de redes de distribución radiales, que, acompañado de monitorización y modelado de las mismas, permitan la resolución de flujos de potencia.

2. FORMULACIÓN DEL PROBLEMA

2.1. Descripción del problema

El programa de flujo de potencia es una herramienta esencial para que la operación y el control de la red de distribución sea eficiente. Los sistemas de distribución son caracterizados por su naturaleza radial y su alto ratio R/X. Esto da lugar al mal condicionamiento del problema de flujo de potencia.

Por ello, en este proyecto se desarrolla una solución basada en la programación cónica, donde se formulan coordenadas cónicas con el fin de sustituir las expresiones tradicionales de las tensiones basadas en las coordenadas cartesianas. Con este cambio de variable, se alcanzará un resultado convergente eficiente en un corto período de tiempo.

Las magnitudes del problema de flujo de potencia se trabajan en por unidad para conseguir la compatibilidad de este enfoque con cualquier escenario de red eléctrica, además de simplificar y agilizar los cálculos.

2.2. Fundamentos teóricos

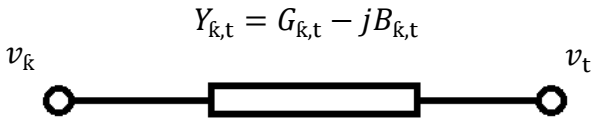

Figura 2.1: Línea de distribución inductiva (elaboración propia)

En este proyecto se tratan redes de distribución cuyas líneas se modelan como una resistencia y una reactancia en serie. Quedan fuera del alcance de este proyecto las líneas con carácter capacitivo, características de redes de distribución subterráneas de media tensión.

Estas líneas inductivas están compuestas por un nodo inicial (ƙ) y un nodo final (t). En estos nudos, la expresión del fasor tensión en coordenadas cartesianas de la que partimos es la siguiente:

$$v_{ƙ} = e_k + jf_k,$$

donde $e_{ƙ}$ es la parte real de la tensión del nudo ƙ y $f_{ƙ}$ es la parte imaginaria del mismo. Además, la admitancia de la línea ($Y_{ƙ,t}$) está compuesta por:

-Parte real: conductancia, $G_{ƙ,t}$.

-Parte imaginaria: susceptancia, $B_{ƙ,t}$.

Para modificar el comportamiento no lineal de la línea, se introduce el siguiente cambo de variables:

$$c_{ƙ,ƙ} = e_k^2 + f_k^2 ,$$

$$c_{ƙ,t} = e_{ƙ} \cdot e_t + f_{ƙ} \cdot f_t ,$$

$$s_{ƙ,t} = e_{ƙ} \cdot f_t - f_{ƙ} \cdot e_t ,$$

Cabe destacar que para cada par de nudos, se cumple que $c_{ƙ,t} = c_{t,ƙ}$, pero $s_{ƙ,t} = -s_{t,ƙ}$.

Por ejemplo, en una línea formada por un nudo inicial ‘0’ y un nudo final ‘1’, estas expresiones resultan:

$$c_{0,0} = e_0^2 + f_0^2 ,$$

$$c_{0,1} = e_0 \cdot e_1 + f_0 \cdot f_1 , siendo\ c_{1,0} = e_1 \cdot e_0 + f_1 \cdot f_0 = c_{0,1}$$

$$s_{0,1} = e_0 \cdot f_1 - f_0 \cdot e_1 , siendo\ s_{1,0} = e_1 \cdot f_0 - f_1 \cdot e_0 = -s_{0,1}$$

Usando la notación descrita anteriormente y considerando el circuito equivalente de una línea simple, el flujo de potencia existente entre el nodo ƙ y el nodo t puede expresarse de forma lineal como:

$$P_{ƙ,t} = G_{ƙ,t} \cdot c_{ƙ,ƙ} - G_{ƙ,t} \cdot c_{ƙ,t} - B_{ƙ,t} \cdot s_{ƙ,t} , \qquad (2.1)$$

$$Q_{ƙ,t} = B_{ƙ,t} \cdot c_{ƙ,ƙ} - B_{ƙ,t} \cdot c_{ƙ,t} + G_{ƙ,t} \cdot s_{ƙ,t} ,$$

De esta manera, el sistema de ecuaciones que describen el flujo de potencias del problema viene dado por:

$$P_{g,k} - P_{l,k} = \sum_{\forall t \in \aleph_k} P_{k,t} \ , \qquad \forall k \in V,$$

$$Q_{g,k} - Q_{l,k} = \sum_{\forall t \in \aleph_k} Q_{k,t} \ , \qquad \forall k \in V,$$

donde $P_{g,k}$ y $Q_{g,k}$ son las potencias activa y reactiva inyectadas en el nodo k, respectivamente, mientras que $P_{l,k}$ y $Q_{l,k}$ son las potencias activa y reactiva demandadas en el nodo k. Se define $\aleph_k$ como el conjunto de nodos conectados directamente en el nodo k y V como el conjunto de nodos totales.

Para que el cambio de variables sea lógico, se implementa la siguiente restricción:

$$c_{k,t}^2 + s_{k,t}^2 = c_{k,k} \cdot c_{t,t} \ ,$$

Si desarrollamos esta expresión sustituyendo las componentes de las tensiones, comprobamos que el cambio de variables ciertamente es correcto:

$$(e_k \cdot e_t + f_k \cdot f_t)^2 + (e_k \cdot f_t - f_k \cdot e_t)^2 = (e_k^2 + f_k^2) \cdot (e_t^2 + f_t^2) \ ,$$

$$(e_k \cdot e_t)^2 + (f_k \cdot f_t)^2 + 2 \cdot e_k \cdot e_t \cdot f_k \cdot f_t + (e_k \cdot f_t)^2 + (f_k \cdot e_t)^2 - 2 \cdot e_k \cdot e_t \cdot f_k \cdot f_t = e_k^2 \cdot e_t^2 + e_k^2 \cdot f_t^2 + f_k^2 \cdot e_t^2 + f_k^2 \cdot f_t^2 \ ,$$

$$e_k^2 \cdot e_t^2 + f_k^2 \cdot f_t^2 + e_k^2 \cdot f_t^2 + f_k^2 \cdot e_t^2 = e_k^2 \cdot e_t^2 + f_k^2 \cdot f_t^2 + e_k^2 \cdot f_t^2 + f_k^2 \cdot e_t^2 \ ,$$

Como se observa, la igualdad se cumple.

Se modifica la restricción de igualdad para dar lugar a una inecuación cuadrática. Esta inecuación está relacionada con la formación de un cono, formulándose de la siguiente forma:

$$c_{k,t}^2 + s_{k,t}^2 \leq c_{k,k} \cdot c_{t,t} \ , \ \forall (k,t) \in \varepsilon, \qquad (2.2)$$

se define ε como el conjunto de líneas conectadas a los nodos. El objetivo es maximizar el valor de $c_{k,t}$ con la finalidad de convertir el carácter no convexo de la restricción en una inecuación que sí es convexa. Con este hecho, la resolución del sistema de ecuaciones resulta sustituido por un problema de optimización. Sin embargo, ambos puntos son equivalentes.

Como se aprecia en el análisis, obtenemos dos ecuaciones lineales correspondientes a cada nodo (excluyendo el nodo Slack) y una ecuación no lineal para cada línea de la red que corresponde a la restricción cónica de segundo orden.

Como vemos en la formulación del problema (sistema de ecuaciones 2.1), el conjunto de las variables objetivo puede apilarse en un vector columna $\chi = [col(c_{k,k})^T, col(c_{k,t})^T, col(s_{k,t})^T]^T$, donde $col(c_{k,k})$ representa un operador encargado de apilar todas las variables $c_{k,k}$ en un vector columna, resultando una definición idéntica para las demás variables. De esta manera, el problema puede representarse de forma compacta:

$$A \cdot \chi = b\,, \tag{2.3}$$

$$\chi^T \cdot Q^{(r,t)} \cdot \chi^T = 0\,, \qquad \forall (r,t) \in \varepsilon,$$

donde las ecuaciones lineales de flujo de potencia están integradas en $A * \chi$; y $Q^{(r,t)}$ conforma un conjunto de matrices que constituye las restricciones cónicas del problema.

3. ENTORNO DE PROGRAMACIÓN

3.1. Herramientas utilizadas

Este enfoque se ha llevado a cabo mediante el lenguaje de programación Python, específicamente a través de la interfaz de desarrollo Spyder. Se trata de un software abierto y no requiere de licencia de pago para su uso, lo que lo convierte en una opción muy accesible. El código desarrollado sigue la estructura de la programación orientada a objetos (POO), ya que ofrece multitud de ventajas, como la herencia, el polimorfismo, la abstracción y el encapsulamiento. Estas características permiten reducir la complejidad, mejorar la eficiencia y minimizar la aparición de errores. Este tipo de programación sigue la siguiente estructura:

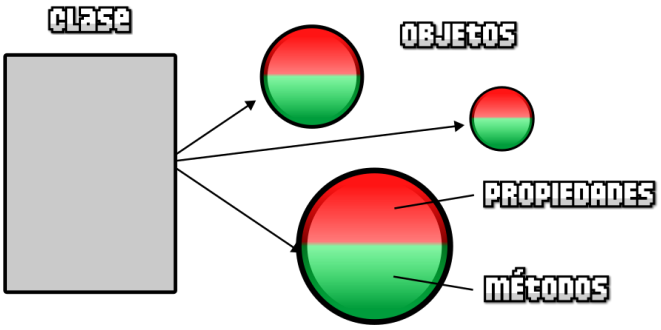

Figura 3.1: Estructura de POO [2]

En la Figura 3.1, se representa el encapsulamiento que tiene lugar en este tipo de programación. Comparándolo con un sistema jerárquico, en lo más alto se encuentra la denominada ‘clase’. Es una plantilla que define las propiedades y comportamientos de un tipo específico de entidad. Por sí misma, una clase es un concepto abstracto que no ocupa memoria. Para poder utilizarla en un programa, es necesario instanciarla, que consiste en crear un nuevo ‘objeto’ concreto de la misma. Un ‘objeto’, por lo tanto, es una instancia de una ‘clase’ que tiene una existencia real en el programa, ocupando memoria y permitiendo su manipulación. Estos se componen de ‘métodos’, que definen su comportamiento mediante funciones, y ‘atributos’ (o ‘propiedades’), que describen las características del ‘objeto’.

Por ejemplo, para crear un programa basado en personas, la estructura resulta como a continuación:

Figura 3.2: Ejemplo de POO [3]

Se puede observar en la Figura 3.2 el uso de este tipo de programación. Como base principal se encuentra la clase ‘Persona’. Esta clase se instancia definiendo los objetos (‘José’ y ‘María’) los cuales representan a cada persona. A su vez, estos se componen de propiedades o atributos (como el sexo o la edad) y de métodos (‘habar’, ‘andar’, ‘comer’ o ‘jugar’) que describen cómo es y qué hace cada una de estas personas.

3.2. Estructura del código

En el caso que nos compete, la programación orientada a objeto se lleva a cabo en el código principal ‘lib’, el cual contiene la mayor parte del contenido esencial para la implementación exitosa del enfoque propuesto.

Su objetivo es organizar y modelar de manera estructurada las entidades principales del sistema del flujo de potencia. Las clases encapsulan datos y comportamientos que se relacionan con las partes fundamentales del problema, facilitando la gestión, el mantenimiento y la posible modificación futura del código.

El código se organiza en cuatro clases principales: ‘Grid’, ‘Node’, ‘Line’ y ‘Prosumer’:

3.2.1. Clase ‘Grid’

La clase ‘Grid’ es el componente principal del código que organiza y gestiona las interacciones entre los nodos, las líneas y los consumidores/generadores. Esta clase permite agregar nodos, líneas y consumidores/generadores al sistema y facilita la resolución global del flujo de potencia en la red. Además, contiene los métodos necesarios para aplicar las ecuaciones de flujo de potencia, como las ecuaciones de Kirchhoff, las matrices definidas en (2.3) y las inecuaciones definidas en (2.2). A su vez, resuelve el sistema de ecuaciones (2.1) y comprueba la validez de las soluciones obtenidas.

3.2.2. Clase ‘Node’

La clase ‘Node’ representa un nodo o punto de conexión en el sistema de flujo de potencia. Cada nodo tiene un identificador único y una lista de líneas y consumidores/generadores asociados. Además, incluye parámetros como la tensión en forma compleja, necesarios para resolver las ecuaciones que definen el flujo de potencia de la red.

3.2.3. Clase ‘Line’

La clase ‘Line’ representa una línea conductora que conecta dos nodos. Las líneas tienen propiedades como la impedancia y los nodos de inicio y fin. Esta clase incluye métodos para calcular la intensidad existente entre los nodos aplicando la ley de Ohm y evaluar las inecuaciones definidas en (2.2) características de la línea mediante las componentes complejas de las tensiones de los nodos asociados.

3.2.4. Clase ‘Prosumer’

La clase ‘Prosumer’ representa los consumidores/generadores asociados a los nodos. Sus propiedades incluyen la potencia activa y reactiva, así como el nodo al que están vinculados. La clase también incluye un método para calcular la intensidad absorbida del nodo basándose en la potencia aparente y la tensión del nodo correspondiente.

4. CASOS DE ESTUDIO.

4.1. Introducción a los casos de estudio

Los casos que se abordan en este proyecto están representados en códigos independientes. Estos códigos se ejecutan en Python a la vez que el código principal 'lib'. La diferencia es que, en este código 'main', los datos del sistema, como las características de los nudos y las líneas, la potencia base y la tensión base, se guardan en variables propias del código. Posteriormente, llevan a cabo una importación de la librería principal 'lib' para así instanciar las clases definidas en esta librería.

El objetivo es calcular las tensiones de los nodos mediante las nuevas variables expuestas en puntos anteriores de este trabajo. Para ello, se calcula la matriz columna χ mediante la ecuación (2.3). Luego, se procede al cálculo de las intensidades, tanto de las líneas como de los consumidores/generadores, para comprobar que se cumpla la ley de Kirchhoff en los nodos, asegurando que los resultados sean correctos.

Se presentan tres redes eléctricas de ejemplo en las que se evalúa la resolución del problema de flujo de potencia propuesta: red de 4 nodos, red de 33 nodos y red de 69 nodos. El tipo de red con el que se trabaja es radial con líneas inductivas, ya que son las líneas de distribución objetivo del enfoque de este proyecto.

4.2. Red de 4 nodos

En este caso, la red eléctrica está compuesta por 4 nodos (nodo 1 es Slack), 3 líneas y 3 consumidores/generadores pertenecientes a los nodos que no son Slack.

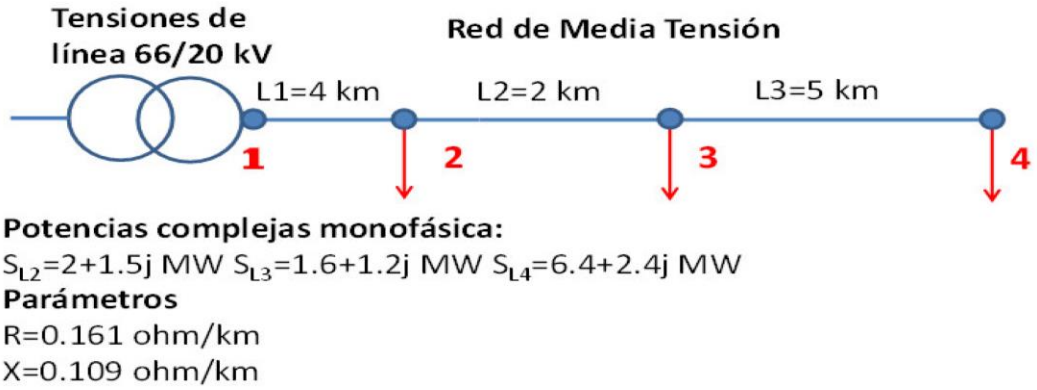

Figura 4.1: Red de 4 nodos

Sbase (MW)	Vbase (kV)
0,1	20

Tabla 4.1: Datos relevantes red de 4 nodos

Se trabajará en valores por unidad para agilizar los cálculos.

Añadiendo los datos en las variables correspondientes a los nudos, líneas y consumidores/generadores del código 'main', se obtienen unos valores de tensión para cada nudo e intensidades en cada línea y consumidor/generador mostrados a continuación:

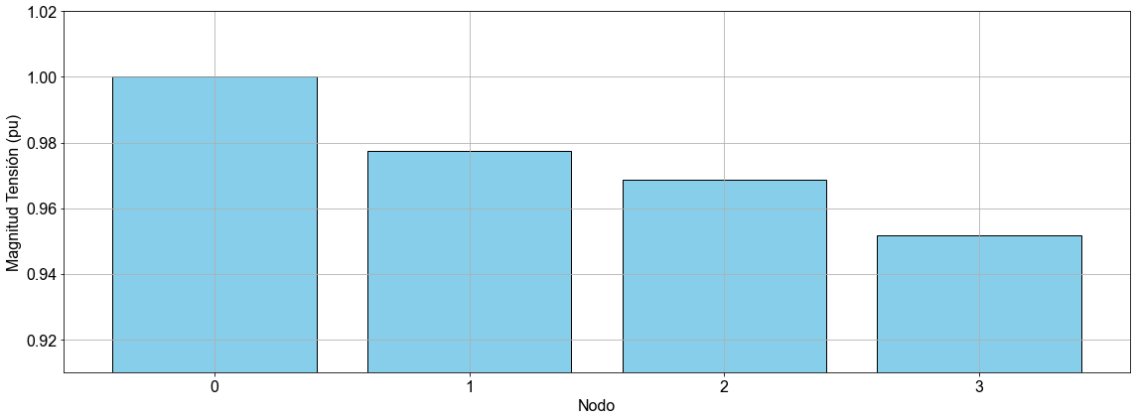

Figura 4.2: Tensión en nodos (elaborado con Python)

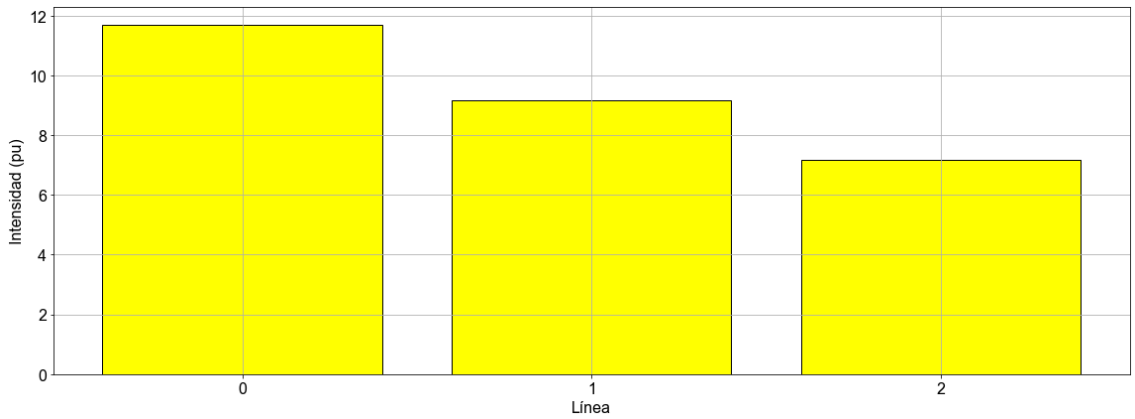

Figura 4.3: Corriente en líneas (elaborado con Python)

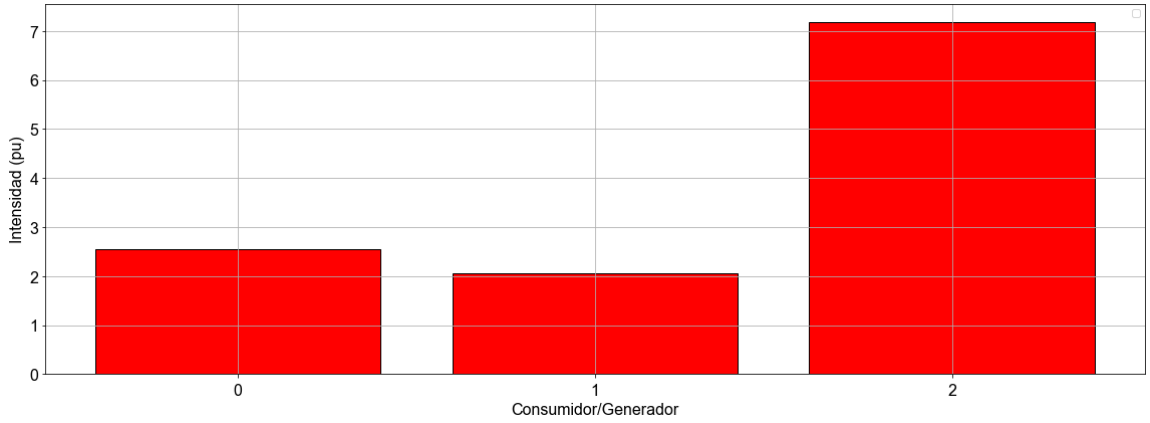

Figura 4.4: Corriente en consumidores/generadores (elaborado con Python)

Se puede observar en la Figura 4.2 cómo la tensión sufre una leve caída a lo largo de la red, manteniéndose bastante uniforme. Estos valores de caída de tensión están dentro de los esperados, ya que son característicos de la presencia de la elevada relación resistencia/reactancia (R/X) en las líneas de distribución de media/baja tensión.

En cuanto a las intensidades, en la Figura 4.3 se aprecia una distribución progresiva de la carga a lo largo de la red, siendo la primera línea la que mayor intensidad soporta. Esto resulta coherente, ya que es la línea saliente del nudo Slack, y, por tanto, del centro de transformación, además de soportar la carga perteneciente a

los nodos posteriores. La Figura 4.4 muestra la cantidad de demanda que presenta cada nodo, siendo notoriamente superior en el nodo 3 (consumidor 2 en la gráfica).

Al realizar la comprobación de la ley de Kirchhoff, se verifica que dicha condición ($\Sigma I = 0$) se cumple, por lo que el sistema se ha resuelto satisfactoriamente. Para ello, se ha considerado la corriente entrante en el nodo con signo positivo y la corriente saliente con negativo, ya sea la de la línea siguiente o la del consumidor.

4.3. Red de 33 nodos

Este caso consiste en una red eléctrica compuesta por 33 nodos mallada. Para poder implementarlo en el código, se han eliminado las líneas necesarias para convertir la red en radial. Estas líneas son las que unen los nodos 7-20, 8-14, 11-21, 17-32 y 24-28.

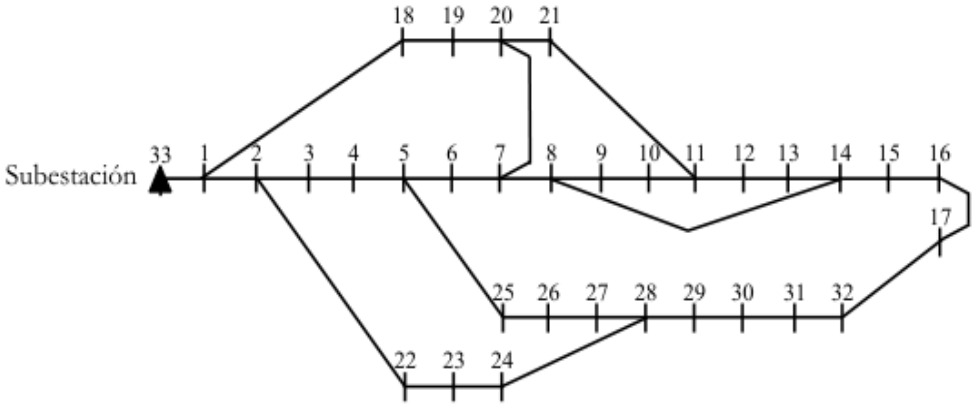

Figura 4.5: Red de 33 nodos

Red de 33 nudos			
Datos básicos	**Potencia activa total de consumo (pu)**	**Potencia reactiva total de consumo (pu)**	**Nº de nodos excepto Slack**
	37,15	23	32
	Sbase (MW)	**Vbase (kV)**	**Nº de bucles**
	0,1	12,66	5
Red radial	**Ramas abiertas**		
	7-20, 8-14, 11-21, 17-32 y 24-28.		

Tabla 4.3: Datos relevantes red de 33 nodos

En la tabla 4.3 se encuentran los datos básicos a introducir en el código 'main'. Además, las características de las líneas y de los consumidores/generadores se encuentran en [4]. Trabajando en valores por unidad, se obtienen los siguientes resultados:

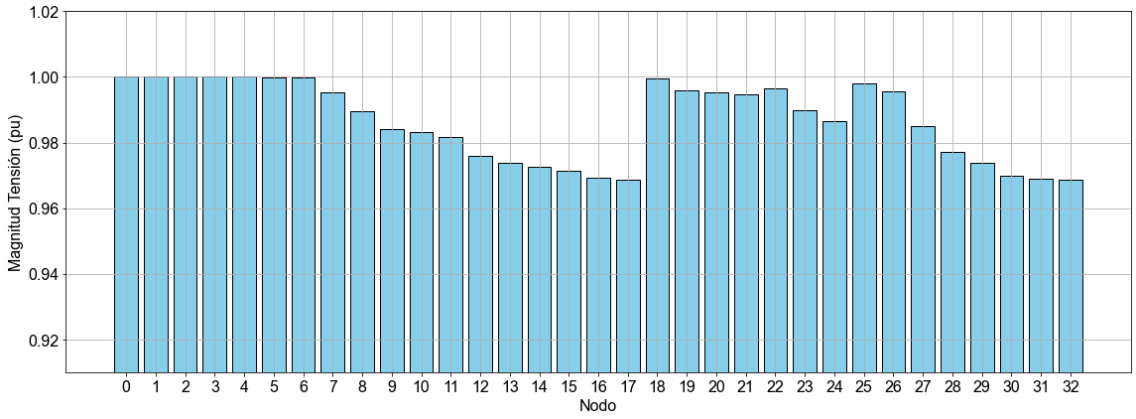

Figura 4.6: Tensión en nodos (elaborado con Pyhton)

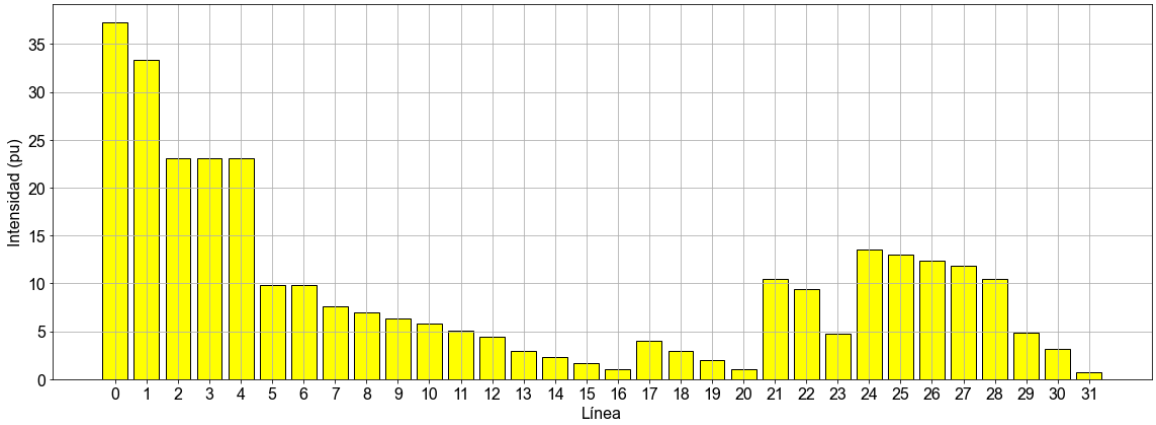

Figura 4.7: Corriente en líneas (elaborado con Pyhton)

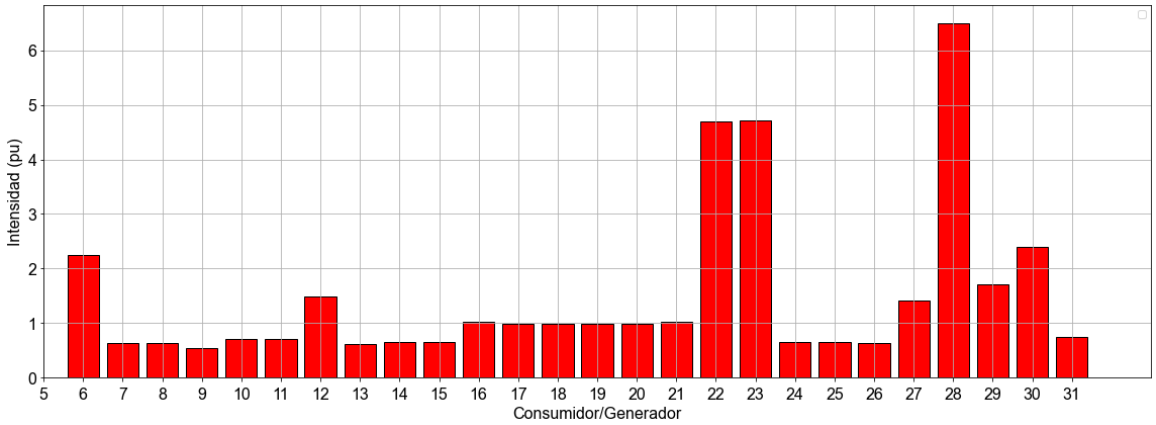

Figura 4.8: Corriente en consumidores/generadores (elaborado con Pyhton)

Se puede observar en la Figura 4.6 cómo la tensión sufre un ligero aumento de valor, por lo que, a lo largo de la red, la tensión se mantiene prácticamente en el mismo valor. Esto indica que el sistema cuenta con un control óptimo del perfil de tensión en toda la red, lo que es crucial para la estabilidad operativa en una red de distribución extensa como la expuesta en este caso.

De la Figura 4.7 cabe destacar que las intensidades de las líneas sufren un patrón decreciente a medida que se avanza hacia nodos más alejados del Slack. Este hecho es coherente con una distribución radial, ya que las líneas iniciales deben

soportar tanto las intensidades salientes del centro de transformación como las que alimentan a los nodos posteriores. El patrón decreciente se lleva a cabo de manera gradual, lo que indica una correcta distribución de carga en las líneas de la red.

En cuanto a los consumidores/generadores, se aprecia en la Figura 4.8 una gran variedad en los niveles de consumo en los nodos de la red, produciéndose picos agresivos en alguno de ellos. Esto puede deberse a una alta demanda energética concentrada en estos nudos. Sin embargo, esto indica la capacidad del modelo para adaptarse a distintos escenarios de carga.

Para verificar que los resultados obtenidos son correctos, comprobamos mediante la ley de Kirchhoff que se cumpla la condición $\Sigma I = 0$. Manteniendo el mismo criterio de signos, comprobamos que la condición se cumple satisfactoriamente.

4.4. Caso 69 nodos

Este caso consiste en una red eléctrica compuesta por 69 nodos mallada. Para poder implementarlo en el código, como anteriormente, se han eliminado las líneas necesarias para convertir la red en radial. Estas líneas son las que unen los nodos 10-68, 12-20, 14-62, 38-48 y 26-54.

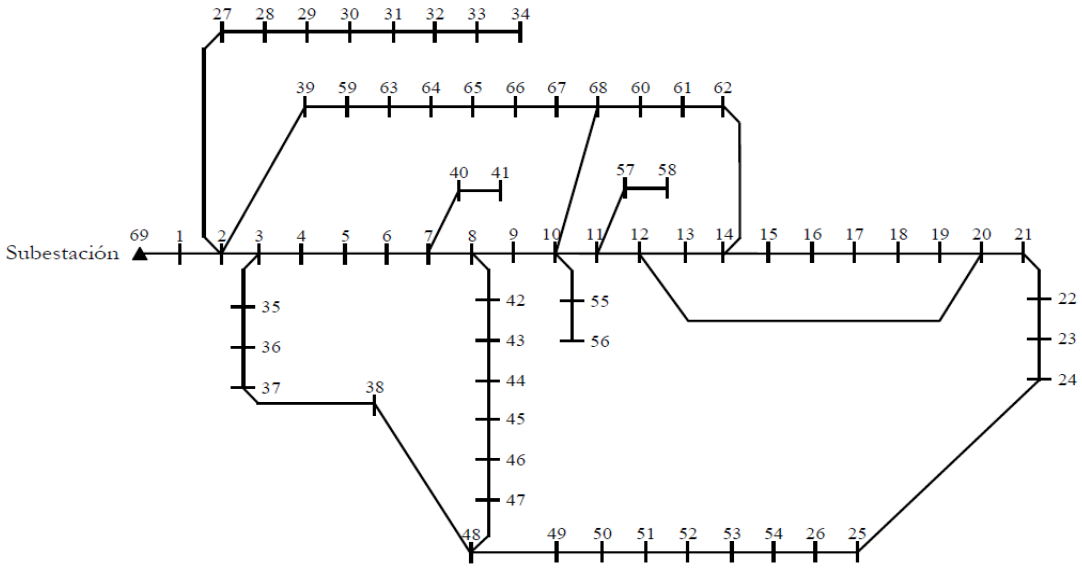

Figura 4.9: Red de 69 nodos

Red de 69 nudos			
Datos básicos	**Potencia activa total de consumo (pu)**	**Potencia reactiva total de consumo (pu)**	**Nº de nodos excepto Slack**
	11,079	8,979	68
	Sbase (MW)	**Vbase (kV)**	**Nº de bucles**
	0,1	12,66	5
Red radial	**Ramas abiertas**		
	10-68, 12-20, 14-62, 38-48 y 26-54.		

Tabla 4.4: Datos relevantes red de 69 nodos

Con los datos aportados en la tabla 4.4, además de las características de las líneas y de los consumidores/generadores recogidas en [4], y trabajando en valores por unidad, se obtienen los siguientes resultados:

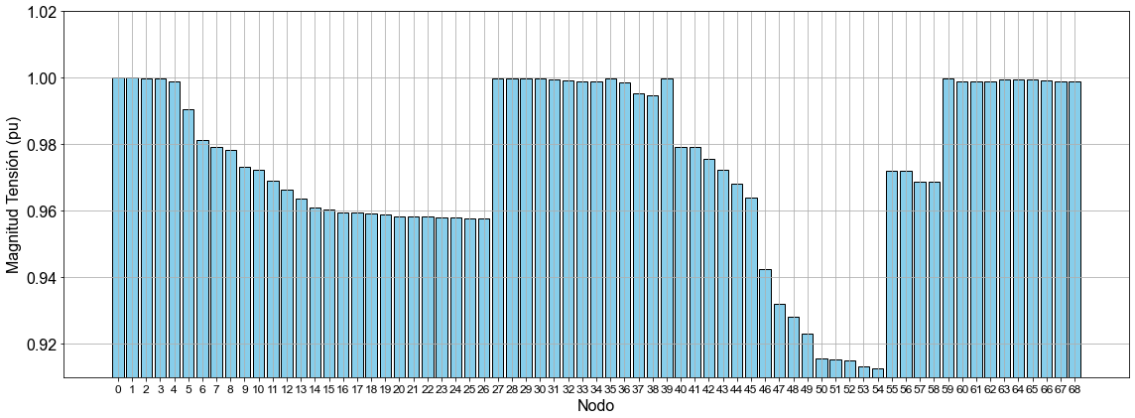

Figura 4.13: Tensión en nodos (elaborado con Python)

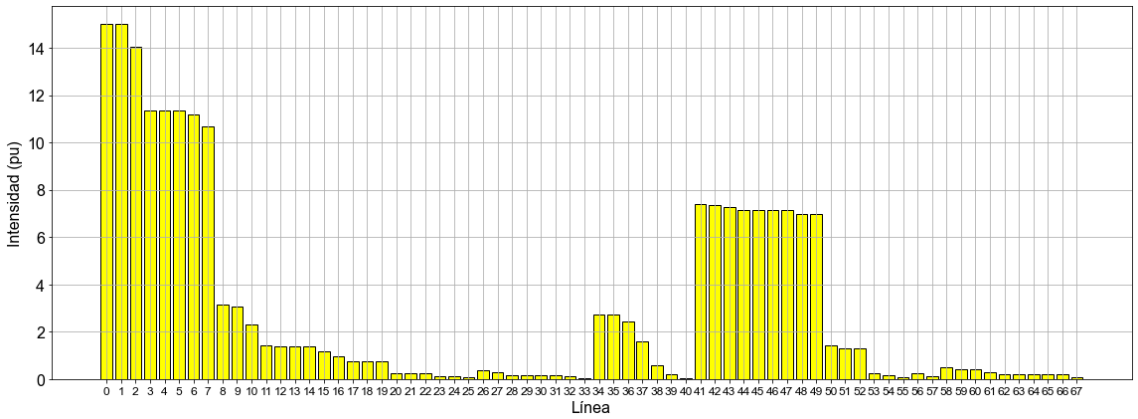

Figura 4.14: Corriente en líneas (elaborado con Python)

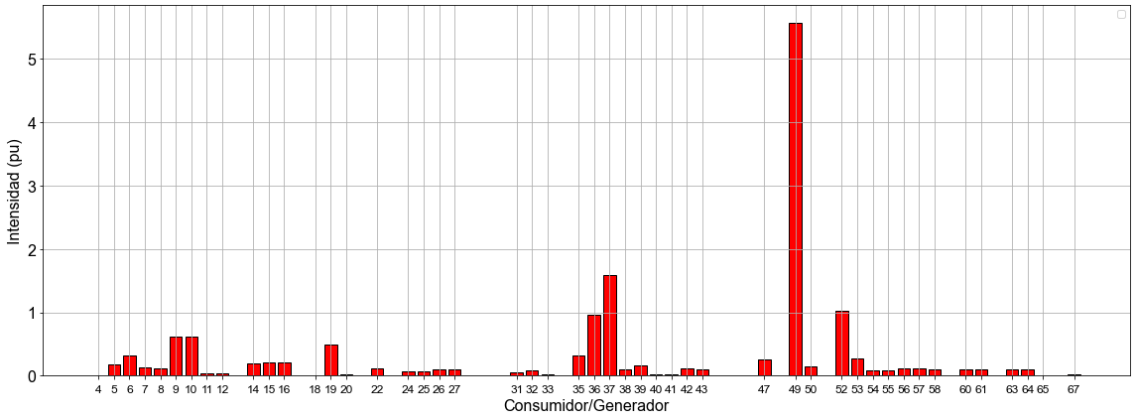

Figura 4.15: Corriente en consumidores/generadores (elaborado con Python)

En la Figura 4.13 se puede apreciar que la tensión sufre ligeras caídas puntuales a lo largo de la red eléctrica. Por lo tanto, se puede concluir con que la red está operando de manera eficiente, logrando que el sistema adopte una excelente regulación de voltaje, siendo imprescindible para el correcto funcionamiento de una red eléctrica de mayor escala como la expuesta en este caso.

Haciendo referencia a las intensidades, en la Figura 4.14 se observa cómo la corriente en las líneas sigue una tendencia decreciente a medida que se alcanzan los nudos más lejanos al Slack. La subida repentina de intensidad es debido a que

las líneas a las cuales pertenece ese valor parten de nodos cercanos al Slack (por ejemplo, la línea 34 parte del nodo 3). En cuanto a los consumidores/generadores, cabe destacar que, a pesar de mostrar picos de intensidad en algunos nodos (Figura 4.15), el sistema no se desestabiliza, por lo que la red no sufrirá ningún problema.

Para verificar que los resultados obtenidos son correctos, se debe comprobar que la ley de Kirchh

off se cumple ($\Sigma I = 0$). Tomando el mismo criterio de signos que en los casos anteriores, se concluye que el criterio se cumple, por lo que el problema está resuelto satisfactoriamente.

5. CONCLUSIONES Y FUTUROS TRABAJOS

5.1. Conclusiones

La implementación de la programación orientada a objeto en el lenguaje de programación Python ha demostrado ser una herramienta eficaz para resolver problemas de flujo de potencia en redes de distribución radiales, gracias al encapsulamiento y modularidad características de este método de programación. Es una herramienta que ofrece una gran variedad de aplicaciones y, en el caso de este proyecto, ha facilitado significativamente la optimización de flujo de potencia en redes eléctricas.

A lo largo de este trabajo, se ha llevado a cabo un modelo de flujo de potencia basado en la programación cónica de segundo orden. El enfoque propuesto ha demostrado ser capaz de calcular eficientemente las intensidades de las líneas y las tensiones de los nodos, ofreciendo una solución alternativa a los métodos tradicionales de flujo de potencia. Gracias a su capacidad para obtener soluciones convexas mediante la formulación de coordenadas cónicas y la rapidez en la resolución, este método abre nuevas oportunidades para su aplicación en diversos escenarios del ámbito de las redes eléctricas.

A pesar del éxito obtenido en las simulaciones realizadas en redes radiales, se ha identificado una limitación en su aplicación para redes malladas. La complejidad adicional que presentan estas redes en términos de programación ha impedido, en el alcance de este proyecto, resolver este tipo de sistemas de manera eficiente.

Además, a pesar de tratar con líneas inductivas en los diversos casos analizados, cabe destacar que el modelo desarrollado es totalmente adaptable para incluir cargas capacitivas. La implementación de elementos capacitivos no supondría cambios significativos en la estructura del código, lo que amplía el abanico de aplicaciones potenciales de este método en análisis de redes con características mixtas.

5.2. Futuros trabajos

A lo largo del presente proyecto, se ha desarrollado un modelo que ha demostrado ser eficaz para resolver el flujo de potencia en redes de distribución radiales. Sin embargo, existen varias oportunidades de expansión y mejora que podrían abordarse en futuros trabajos. A continuación, se detallan algunas propuestas para ampliar el alcance y las capacidades del modelo:

5.2.1. Optimización para redes malladas

A pesar de ser la principal limitación identificada en el proyecto, cabe la posibilidad de adaptar el código actual y abordar el análisis de redes malladas. Estas redes, que son más complejas debido a la presencia de bucles y caminos redundantes, son comunes en sistemas de distribución urbanos actuales. En futuros trabajos, se podrían implementar técnicas avanzadas, como la descomposición de Benders o métodos basados en descomposición de dominios, para mejorar la eficiencia en la resolución de estos sistemas. Además, se podrían explorar algoritmos de optimización no lineal para gestionar de forma más efectiva la complejidad adicional que presentan estas tipologías.

5.2.2. Aplicación del modelo en redes de mayor escala

El modelo propuesto en este proyecto ha sido puesto a prueba en redes de hasta 69 nodos, pero resultaría interesante expandir su aplicación a redes de mayores dimensiones, como redes de 84 nodos o incluso 114. En este sentido, se podría optimizar el rendimiento del código mediante paralelización y el uso de software externo para la caracterización del sistema, tal como se ha llevado a cabo en los casos propuestos a lo largo del presente trabajo. Esto ayuda a que la simulación de redes a gran escala alcance un tiempo razonable y no se demore en demasía.

5.2.3. Integración de fuentes de energía renovable

Con el auge de la incorporación de las energías renovables en el presente, como la solar fotovoltaica y la eólica, se ha transformado significativamente la dinámica de operación de las redes de distribución. Para trabajos futuros, resultaría de gran interés extender el modelo desarrollado para incluir la integración de fuentes de energía renovable distribuidas. Esto no solo proporcionaría un análisis más realista y actualizado de los sistemas actuales, sino que también permitiría evaluar el impacto de la generación intermitente en el flujo de potencia y la estabilidad de la red.

6. REFERENCIAS

[1] Definición de problema convexo: https://es.quora.com/Qu%C3%A9-significa-un-problema-convexo.

[2] “Estructura de POO”, disponible en: https://lenguajejs.com/javascript/oop/que-es/.

[3] “Ejemplo de POO”, disponible en: https://hilmer.vip/2022/12/13/programacion-orientada-a-objetos/.

[4] M.E. Baran and F.F. Wu, “Network reconfiguration in distribution systems for los reduction and load balancing”, in IEEE Transactions on Power Delivery, vol 4, no. 2, pp. 1401-1407, April 1989, disponible en:

https://ieeexplore.ieee.org/document/25627.

[5] Repositorio de Github: https://github.com/arnozal/conic_power_flow.

7. ANEXOS

7.1 Código 'lib' de Python

```
1. # Importing required libraries
2. import numpy as np
3. from scipy.optimize import minimize, NonlinearConstraint,
   LinearConstraint
4.
5. class grid:
6.     def __init__(self, nodes, lines, pros):
7.         self.nodes                                          =
   self.add_nodes(nodes)
8.         self.lines = self.add_lines(lines, self.nodes)
9.         self.pros = self.add_pros(pros, self.nodes)
10.             self.n = len(self.nodes)
11.             self.m = len(self.lines)
12.             self.x_size = self.n+(2*self.m)-1
13.             self.obtain_index()
14.
15.         def add_nodes(self, nodes):
16.
17.             nodes_list = list()
18.             for item in nodes:
19.                 nodes_list.append(node(item['id'],
   item['slack']))
20.             return nodes_list
21.
22.         def add_lines(self, lines, nodes):
23.
24.             lines_list = list()
25.             for item in lines:
26.                 lines_list.append(line(item['id'],
   item['From'], item['To'], item['R'], item['X'], nodes))
27.             return lines_list
28.
29.         def add_pros(self, pros, nodes):
30.
31.             pros_list = list()
32.             for item in pros:
33.                 pros_list.append(prosumer(item['id'],
   item['Node'], item['P'], item['Q'], nodes))
34.             return pros_list
```

```
35.
36.         def obtain_index(self):
37.
38.             n_aux = 0
39.             while n_aux < self.n:
40.                 self.nodes[n_aux].index = n_aux -1
41.                 n_aux += 1
42.
43.             n_aux2 = 0
44.             while n_aux2 < self.m:
45.                 self.lines[n_aux2].index.append(n_aux - 1)
46.                 n_aux += 1
47.                 n_aux2 += 1
48.
49.             n_aux3 = 0
50.             while n_aux3 < self.m:
51.                 self.lines[n_aux3].index.append(n_aux - 1)
52.                 n_aux += 1
53.                 n_aux3 += 1
54.
55.             self.X = np.zeros(self.x_size)
56.             self.X[:self.n - 1] = 1
57.
58.         def obtain_A(self):
59.
60.             matrizA         =         np.zeros(((2*self.n)-2,
    (self.n+2*self.m)-1), dtype=float)
61.
62.             n_aux = 0
63.             for i, node in enumerate(self.nodes[1:]):
64.                 matrizA[2*i,  n_aux]  =  np.sum([line.G  for
    line in node.lines])
65.                 matrizA[2*i+1,  n_aux]  =  np.sum([line.B  for
    line in node.lines])
66.                 n_aux += 1
67.
68.                 for j, line in enumerate(node.lines):
69.                     if node == line.nodes[0]:
70.                         matrizA[2*i,    line.index[0]]    =-
    line.G
71.                         matrizA[2*i,    line.index[1]]    =-
    line.B
72.                         matrizA[2*i+1,   line.index[0]]   =-
    line.B
73.                         matrizA[2*i+1,    line.index[1]]    =
    line.G
```

```
74.                     else:
75.                         matrizA[2*i,    line.index[0]]    =-
   line.G
76.                         matrizA[2*i, line.index[1]] = line.B
77.                         matrizA[2*i+1,   line.index[0]]   =-
   line.B
78.                         matrizA[2*i+1,   line.index[1]]   =-
   line.G
79.             self.A = matrizA
80.
81.         def ineq(self, X):
82.
83.             rest = []
84.             for line in self.lines:
85.                 rest.append(line.ineq(X))
86.
87.             return rest
88.
89.         def intensity(self):
90.             intens = []
91.             for line in self.lines:
92.                 intens.append(line.intensity())
93.             return intens
94.
95.         def intensity_pros(self):
96.             intens_pros = []
97.             for pros in self.pros:
98.                 intens_pros.append(pros.intensity())
99.             return intens_pros
100.
101.        def comprobacion_Kirchhoff(self, tolerancia = 1e-3):
102.            Check = []
103.            for node in self.nodes[1:]:
104.                total_intens = 0 + 0j
105.                for line in node.lines:
106.                    if line.nodes[0] == node:
107.                        total_intens                        -=
   line.intensity()  #Sale intensidad del nodo
108.                    elif line.nodes[1] == node:
109.                        total_intens                       +=
   line.intensity()  #Entra intensidad al nodo
110.                for pros in node.pros:
111.                    total_intens += pros.intensity()
112.                if abs(total_intens) < tolerancia:
113.                    Check.append(True)
114.                else:
```

```
115.                     Check.append(total_intens)
116.             return Check
117.
118.         def solve_pf(self):
119.
120.             self.obtain_A()
121.             self.obtain_B()
122.             self.obtain_f()
123.
124.             lc = LinearConstraint(self.A, self.B, self.B)
125.             nlc = NonlinearConstraint(self.ineq, -np.inf, 0)
126.             fo = lambda x: self.f.dot(x)
127.             sol = minimize(fo, self.X, constraints=(lc, nlc))
128.
129.             for index, node in enumerate(self.nodes[1:]):
130.                 node.Ckk = sol.x[index]
131.
132.             return sol
133.
134.         def obtain_B(self):
135.
136.             matrizB = np.zeros(2*self.n-2, dtype=float)
137.
138.             for i, node in enumerate(self.nodes[1:]):
139.
140.                 for x in node.pros:
141.                     matrizB[2*i] += x.P
142.                     matrizB[2*i+1] += x.Q
143.             self.B = matrizB
144.
145.         def obtain_f(self):
146.
147.             f = np.zeros((1, self.x_size))
148.             f[0, self.n - 1:(self.n+self.m) - 1] =- 1
149.             self.f = f
150.
151.         def obtain_volt(self):
152.
153.             self.nodes[0].U = complex(1, 0)
154.             for line in self.lines:
155.                 A   =   np.array([[np.real(line.nodes[0].U),
    np.imag(line.nodes[0].U)],         [-np.imag(line.nodes[0].U),
    np.real(line.nodes[0].U)]], dtype = np.float64)
156.                 b  =  np.array([line.Ckt,  line.Skt],  dtype  =
    np.float64)
157.                 x = np.linalg.solve(A, b)
```

```
158.                 line.nodes[1].U = complex(x[0], x[1])
159.
160.             return [node.U for node in self.nodes]
```

7.2. Código 'main' para red de 4 nodos

```
1.  import numpy as np
2.  import lib
3.
4.
5.  Sbase = 1e6
6.  Ubase = 20e3
7.  Zbase = (Ubase**2)/Sbase
8.
9.  # Nodes
10. Nodes = [{'id': 0, 'slack': True },
11.          {'id': 1, 'slack': False},
12.          {'id': 2, 'slack': False},
13.          {'id': 3, 'slack': False},]
14.
15. # Lines
16. Lines = [{'id': 0,  'From': 0,  'To': 1,  'R': 0.161*4/Zbase, 'X': 0.109*4/Zbase},
17.          {'id': 1,  'From': 1,  'To': 2,  'R': 0.161*2/Zbase, 'X': 0.109*2/Zbase},
18.          {'id': 2,  'From': 2,  'To': 3,  'R': 0.161*5/Zbase, 'X': 0.109*5/Zbase}]
19.
20. # Prosumers
21. Pros = [{'id': 0, 'Node': 1, 'P': -2e6/Sbase, 'Q': -1.5e6/Sbase},
22.         {'id': 1, 'Node': 2, 'P': -1.6e6/Sbase, 'Q': -1.2e6/Sbase},
23.         {'id': 2, 'Node': 3, 'P': -6.4e6/Sbase, 'Q': -2.4e6/Sbase},]
24.
25. # Constructing network and solving power flow
26. net = lib.grid(Nodes, Lines, Pros)
27.
28. sol = net.solve_pf()
29. Volt = net.obtain_volt()
30. I = net.intensity()
31. Check = net.comprobacion_Kirchhoff()
32. I_pros = net.intensity_pros()
```

7.3. Código 'main' para red de 33 nodos

7.3.1 Código de Python

```
1.  import numpy as np
```

```
import lib

Sbase = 0.1e6
Ubase = 12.66e3
Zbase = (Ubase**2)/Sbase

Nodes = []
Lines = []
      Pros = []
      Nodes.append({'id': 0, 'S': 0, 'fdp': 0, 'slack': True})

      flag = False
      data = 'something'
      with open('red32nudos/datos_32.m') as fp:
          data = fp.readline()
          while(data):
              if flag:
                  data = data.split(' ')
                  data = list(filter(lambda x: x != '' and x != '\n' and x != ']', data))
                  if len(data) == 0:
                      break
                  Lines.append({
                                  'id': int(data[0])-1,
                                  'From': int(data[1]),
                                  'To': int(data[2]),
                                  'R': float(data[3])/Zbase,
                                  'X': float(data[4])/Zbase,
                                  'Long':1.0
                                  })
                  if int(data[2]) not in [node['id'] for node in Nodes]:
                      Nodes.append({
                                      'id': int(data[2]),
                                      'S':  np.sqrt((float(data[5])*1e3)**2 + (float(data[6])*1e3)**2)/Sbase,
                                      'fdp': float(data[5])/(np.sqrt(float(data[5])**2    +    float(data[6])**2))    if float(data[5]) != 0 else 0,
                                      'slack': False
                                      })
                  if float(data[5]) != 0 and float(data[6]) != 0:
                      Pros.append({
                                      'id': int(data[2])-1,
                                      'Node': int(data[2]),
                                      'P': -float(data[5])*1000/Sbase,
                                      'Q': -float(data[6])*1000/Sbase
                                      })

```

```
46.             if 'datsis' in data:
47.                 flag = True
48.                 data = data.split('datsis = [')[1]
49.                 data = data.split(' ')
50.                 data = list(filter(lambda x: x != '' and x != '\n' and
    x != ']', data))
51.                 Lines.append({
52.                                 'id': int(data[0])-1,
53.                                 'From': 0,
54.                                 'To': int(data[2]),
55.                                 'R': float(data[3])/Zbase,
56.                                 'X': float(data[4])/Zbase,
57.                                 'Long':1.0,
58.                                 })
59.                 Nodes.append({
60.                                 'id': int(data[2]),
61.                                 'S':   np.sqrt((float(data[5])*1e3)**2   +
    (float(data[6])*1e3)**2)/Sbase,
62.                                 'fdp':
    float(data[5])/(np.sqrt(float(data[5])**2 + float(data[6])**2)),
63.                                 'slack': False
64.                                 })
65.                 Pros.append({
66.                                    'id': int(data[2])-1,
67.                                    'Node': int(data[2]),
68.                                    'P': -float(data[5])*1000/Sbase,
69.                                    'Q': -float(data[6])*1000/Sbase
70.                                    })
71.             data = fp.readline()
72.
73.     Lines = Lines[:32] #Las redes radiales.
74.     net = lib.grid(Nodes, Lines, Pros)
75.
76.     sol = net.solve_pf()
77.     Volt = net.obtain_volt()
78.     I = net.intensity()
79.     Check = net.comprobacion_Kirchhoff()
80.     I_pros = net.intensity_pros()
```

7.3.2. Datos de Matlab

```
1. % Datos del sistema de 32 nudos (articulo_red32nudos.pdf)
```

```
2.
3. %                                    Matriz                                    [
   nºrama  nudo_origen  nudo_destino  R(pu)   X(pu)  Pnudo_destino(MW)  Qnud
   o_destino(MVAr) ]
4.
5. datsis = [  1            69              1        9.36E-7    2.25E-
   6        0                0
6.             2            1               2        9.36E-7    2.25E-
   6        0                0
7.             3            2               3        2.81E-6    6.74E-
   6        0                0
8.             4            3               4        4.70E-5    5.50E-
   5        0                0
9.             5            4               5        6.85E-4    3.49E-
   4        8.78E-4          7.20E-4
10.                  6                5               6          7.13E-
   4   3.63E-4        1.35E-2          9.98E-3
11.                  7            6               7       0.7114    0.235
   1        200.00              100.00
12.                  8            7               8       1.0300    0.740
   0         60.00               20.00
13.                  9            8               9       1.0440    0.740
   0         60.00               20.00
14.                 10            9              10       0.1966    0.065
   0         45.00               30.00
15.                 11           10              11       0.3744    0.123
   8         60.00               35.00
16.                 12           11              12       1.4680    1.155
   0         60.00               35.00
17.                 13           12              13       0.5416    0.712
   9        120.00               80.00
18.                 14           13              14       0.5910    0.526
   0         60.00               10.00
19.                 15           14              15       0.7463    0.545
   0         60.00               20.00
20.                 16           15              16       1.2890    1.721
   0         60.00               20.00
21.                 17           16              17       0.7320    0.574
   0         90.00               40.00
22.                 18            1              18       0.1640    0.156
   5         90.00               40.00
23.                 19           18              19       1.5042    1.355
   4         90.00               40.00
24.                 20           19              20       0.4095    0.478
   4         90.00               40.00
```

```
25.                21          20          21        0.7089    0.937
   3         90.00                40.00
26.                22           2          22        0.4512    0.308
   3         90.00                50.00
27.                23          22          23        0.8980    0.709
   1        420.00               200.00
28.                24          23          24        0.8960    0.701
   1        420.00               200.00
29.                25           5          25        0.2030    0.103
   4         60.00                25.00
30.                26          25          26        0.2842    0.144
   7         60.00                25.00
31.                27          26          27        1.0590    0.933
   7         60.00                20.00
32.                28          27          28        0.8042    0.700
   6        120.00                70.00
33.                29          28          29        0.5075    0.258
   5        200.00               600.00
34.                30          29          30        0.9744    0.963
   0        150.00                70.00
35.                31          30          31        0.3105    0.361
   9        210.00               100.00
36.                32          31          32        0.3410    0.530
   2         60.00                40.00
37.                33           7          20        2.0000    2.000
   0          0.00                 0.00
38.                34           8          14        2.0000    2.000
   0          0.00                 0.00
39.                35          11          21        2.0000    2.000
   0          0.00                 0.00
40.                36          17          32        0.5000    0.500
   0          0.00                 0.00
41.                37          24          28        0.5000    0.500
   0          0.00                 0.00 ];
```

7.4. Código ‘main’ para red de 69 nodos

7.4.1 Código de Python

```
import numpy as np
import pandas as pd
import lib

Sbase = 0.1e6
Ubase = 12.66e3
Zbase = (Ubase**2)/Sbase

Nodes = []
      Lines = []
      Pros = []
      Nodes.append({
                    'id': 0,
                    'S': 0,
                    'fdp': 0,
                    'slack': True
                    })

      #    df    =    pd.read_csv(os.path.join('red69nudos','69nudos.csv'), decimal=',')
      df = pd.read_csv('red69nudos/69nudos.csv', decimal=',')
      df.head()
      df.fillna(0.0, inplace=True)
      df['node_from'].replace(69, 0, inplace=True)
      df['node_to'].replace(69, 0, inplace=True)

      for i, row in df.iterrows():
          Lines.append({
                       'id': i,
                       'From': int(row['node_from']),
                       'To': int(row['node_to']),
                       'R': row['r_pu'],
                       'X': row['x_pu'],
                       'Long':1.0,
                       })
          if int(row['node_to']) not in [69] + [node['id'] for node in Nodes]:
              Nodes.append({
                           'id': int(row['node_to']),
                           'S':  np.sqrt((float(row['p_to_MW'])*1e6)**2  + (float(row['q_to_Mvar'])*1e6)**2)/Sbase,
                           'fdp': float(row['p_to_MW'])/(np.sqrt(float(row['p_to_MW'])**2                 + float(row['q_to_Mvar'])**2)) if float(row['p_to_MW']) != 0 else 0,
                           'slack': False
                           })

          if float(row['p_to_MW']) != 0 and float(row['q_to_Mvar']) != 0:
```

```
44.
45.             Pros.append({
46.                         'id': int(row['node_to']-1,
47.                         'Node': int(row['node_to']),
48.                         'P': -float(row['p_to_MW']/Sbase),
49.                         'Q': -float(row['q_to_Mvar']/Sbase),
50.                         })
51.
52.
53.     Lines = Lines[:68] #Las redes radiales.
54.     net = lib.grid(Nodes, Lines, Pros)
55.
56.     sol = net.solve_pf()
57.     Volt = net.obtain_volt()
58.     I = net.intensity()
59.     Check = net.comprobacion_Kirchhoff()
60.     I_pros = net.intensity_pros()
```

7.4.2. Datos de Excel (cvs)

```
1. node_from,node_to,r_pu,x_pu,p_to_MW,q_to_Mvar
2. 69,1,"9,36E-07","2,25E-06","0,00E+00","0,00E+00"
3. 1,2,"9,36E-07","2,25E-06","0,00E+00","0,00E+00"
4. 2,3,"2,81E-06","6,74E-06","0,00E+00","0,00E+00"
5. 3,4,"4,70E-05","5,50E-05","0,00E+00","0,00E+00"
6. 4,5,"6,85E-04","3,49E-04","8,78E-04","7,20E-04"
7. 5,6,"7,13E-04","3,63E-04","1,35E-02","9,98E-03"
8. 6,7,"1,73E-04","8,80E-05","2,49E-02","1,78E-02"
9. 7,8,"9,23E-05","4,70E-05","1,00E-02","7,21E-03"
10.     8,9,"1,53E-03","5,07E-04","9,33E-03","6,67E-03"
11.     9,10,"3,50E-04","1,16E-04","4,85E-02","3,46E-02"
12.     10,11,"1,33E-03","4,40E-04","4,85E-02","3,46E-02"
13.     11,12,"1,93E-03","6,36E-04","2,71E-03","1,82E-03"
14.     12,13,"1,95E-03","6,46E-04","2,71E-03","1,82E-03"
15.     13,14,"1,98E-03","6,54E-04","0,00E+00","0,00E+00"
16.     14,15,"3,68E-04","1,22E-04","1,52E-02","1,02E-02"
17.     15,16,"7,01E-04","2,32E-04","1,65E-02","1,18E-02"
18.     16,17,"8,80E-06","2,99E-06","1,65E-02","1,18E-02"
19.     17,18,"6,13E-04","2,03E-04","0,00E+00","0,00E+00"
20.     18,19,"3,94E-04","1,30E-04","3,16E-04","2,12E-04"
21.     19,20,"6,39E-04","2,11E-04","3,80E-02","2,71E-02"
22.     20,21,"2,62E-05","8,61E-06","1,76E-03","1,18E-03"
```

```
21,22,"2,98E-04","9,85E-05","0,00E+00","0,00E+00"
22,23,"6,48E-04","2,14E-04","9,39E-03","6,67E-03"
23,24,"1,40E-03","4,63E-04","0,00E+00","0,00E+00"
24,25,"5,78E-04","1,91E-04","4,67E-03","3,33E-03"
25,26,"3,24E-04","1,07E-04","4,67E-03","3,33E-03"
2,27,"8,24E-06","2,02E-05","8,67E-03","6,19E-03"
27,28,"1,20E-04","2,93E-04","8,67E-03","6,19E-03"
28,29,"7,45E-04","2,46E-04","0,00E+00","0,00E+00"
29,30,"1,31E-04","4,34E-05","0,00E+00","0,00E+00"
30,31,"6,57E-04","2,17E-04","0,00E+00","0,00E+00"
31,32,"1,57E-03","5,27E-04","4,58E-03","3,26E-03"
32,33,"3,20E-03","1,06E-03","6,50E-03","4,55E-03"
33,34,"2,76E-03","9,12E-04","1,92E-03","1,29E-03"
3,35,"6,36E-06","1,57E-05","0,00E+00","0,00E+00"
35,36,"1,59E-04","3,90E-04","2,64E-02","1,88E-02"
36,37,"5,42E-04","1,33E-03","2,82E-02","9,15E-02"
37,38,"1,54E-04","3,76E-04","1,28E-01","9,15E-02"
2,39,"8,24E-06","2,02E-05","8,67E-03","6,19E-03"
7,40,"1,74E-04","8,85E-05","1,35E-02","9,44E-03"
40,41,"6,21E-04","2,09E-04","1,20E-03","8,94E-04"
8,42,"3,26E-04","1,66E-04","1,45E-03","1,16E-03"
42,43,"3,80E-04","1,94E-04","8,79E-03","6,32E-03"
43,44,"5,32E-04","2,71E-04","8,00E-03","5,71E-03"
44,45,"5,27E-04","2,68E-04","0,00E+00","0,00E+00"
45,46,"2,98E-03","9,99E-04","0,00E+00","0,00E+00"
46,47,"1,47E-03","4,92E-04","0,00E+00","0,00E+00"
47,48,"5,69E-04","1,88E-04","6,67E-04","2,40E-02"
48,49,"7,23E-04","2,19E-04","0,00E+00","0,00E+00"
49,50,"9,50E-04","4,84E-04","4,15E-01","2,96E-01"
50,51,"1,82E-04","9,28E-05","1,07E-02","7,61E-03"
51,52,"2,71E-04","1,38E-04","0,00E+00","0,00E+00"
52,53,"1,33E-03","6,77E-04","7,57E-02","5,39E-02"
53,54,"1,95E-03","9,92E-04","1,97E-02","1,39E-02"
10,55,"3,77E-04","1,14E-04","6,00E-03","4,28E-03"
55,56,"8,80E-06","2,62E-06","6,00E-03","4,28E-03"
11,57,"1,38E-03","4,57E-04","9,33E-03","6,66E-03"
57,58,"8,80E-06","2,99E-06","9,33E-03","6,66E-03"
39,59,"1,20E-04","2,93E-04","8,67E-03","6,19E-03"
59,63,"1,97E-04","2,30E-04","0,00E+00","0,00E+00"
63,64,"5,69E-05","6,64E-05","8,00E-03","5,71E-03"
64,65,"3,37E-06","3,93E-06","8,00E-03","5,71E-03"
65,66,"1,36E-03","1,59E-03","3,92E-04","3,25E-04"
66,67,"5,80E-04","6,78E-04","0,00E+00","0,00E+00"
67,68,"7,67E-05","8,95E-05","2,00E-03","1,43E-03"
68,60,"1,72E-05","2,17E-05","0,00E+00","0,00E+00"
60,61,"2,04E-04","2,57E-04","3,08E-03","8,79E-03"
61,62,"1,68E-06","2,25E-06","3,08E-03","8,79E-03"
```

```
10,68,"9,36E-04","9,36E-04",,
12,20,"9,36E-04","9,36E-04",,
14,62,"1,87E-03","1,87E-03",,
38,48,"3,74E-03","3,74E-03",,
26,54,"1,87E-03","1,87E-03",,
```