eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
ZIENTZIA ETA TEKNOLOGIA FAKULTATEA
FACULTAD DE CIENCIA Y TECNOLOGÍA

Trabajo Fin de Grado
Grado en Ingeniería Electrónica

Desarrollo Software de un sistema de análisis y resolución de circuitos electrónicos

Autora:
Carmen Legarreta Gallo
Directores:
Javier Echanove Arias
Iñigo Arredondo López

Leioa, 21 de Junio de 2019

Abstract

En los casos de circuitos lineales y pequeños es fácil obtener su soluciona manualmente. Sin embargo, a medida que la complejidad aumenta es necesaria la complementación de métodos numéricos aplicados en computación. En este trabajo se presenta el desarrollo de un programa software para resolver circuitos algebraicos y dinámicos, y se explican los pasos seguidos para la implementan de los métodos numéricos como el de Newton-Raphson y Euler.

Índice general

Índice de cuadros

Índice de figuras

CAPÍTULO 1

Introducción

Con el transistor, inventado en 1945, fue posible mejorar los diseños de los amplificadores de señal, rectificadores, puertas lógicas, etc., y además su pequeño consumo y tamaño permitían una mayor integración. A medida que aumentaba la magnitud de los diseños, el análisis de su comportamiento se complicaba; con conocimientos teóricos era posible obtener una solución que describiera el comportamiento aproximado de un circuito electrónico, sin embargo, en los diseños de aplicaciones especificas era muy importante conocer cuales serían los resultados exactos, por lo tanto era necesario un método/herramienta para diseñar y resolver circuitos electrónicos. A partir de esta necesidad se creó una teoría de circuitos que permitió, más tarde, su implementación en computadoras.

Las computadoras ofrecen cálculos numéricos de gran precisión, por ello se comenzaron a desarrollar programas software para obtener soluciones exactas de circuitos electrónicos. En el año 1973 Laurence Nagel creo PSpice (Simulation Program with Integrated Circuits Enphasis), un programa software para la simulación y diseño de circuitos electrónicos analógicos. Es una herramienta que permite realizar todo tipo de análisis; calculo nodal, respuesta de las variables en función del tiempo, funciones de transferencia, etc. En la versión comercial actual, el usuario implementa de forma gráfica el circuito que desea, después escoge el tipo de análisis, y ejecuta el programa, entonces Pspice ofrece una gráfica con la información que el usuario ha pedido. Además, con el fin de poder recoger la mayor la mayor cantidad de datos posible, se puede interactuar con el gráfico que se obtiene. Al ejecutar el programa, se crea un archivo del tipo text en el que se describe el circuito dibujado, y el programa PSpice coge esa información para realizar el análisis que se desea. Sin duda, por todo lo que ofrece, Pspice es muy importante en la electrónica.

En este trabajo se ha creado un software con el objetivo de resolver circuitos, y así comprender las bases de Pspice, además de poner en práctica conceptos teóricos de algunas asignaturas impartidas a lo largo del grado en Ingeniera Electrónica, como por ejemplo, Circuitos Lineales y No-Lineales, Métodos Computacionales, Fundamentos de la Programación, y Señales y

Sistemas.

Antes de comenzar con el desarrollo software, es imprescindible tener una base teórica sobre los circuitos, y los métodos para solucionarlos, por lo tanto en el segundo capítulo se explica la manera de representar la topología de un circuito. También se aborda la clasificación de los elementos y circuitos, ya que dependiendo del tipo de circuito, se deben aplicar diferentes métodos para su solución.

Más adelante se discuten tres posibles lenguajes de programación para el desarrollo del software, entre los cuales están Python, Java, y Fortran, todas ellos impartidos en el grado. Dependiendo de las prestaciones de cada uno, se determina cuál es el que mejor se adapta al problema que se plantea en este trabajo, y al propio programador, para obtener una mejor eficiencia en el proceso de desarrollo. Con el lenguaje seleccionado, se elegirá una forma de representar los datos, y enlazando la teoría de circuitos con las formulas numéricas de Newton-Raphson y Euler, también se determinará el esquema general para analizar y resolver circuitos. A medida que se vaya implantando parte del esquema, se realizarán pruebas del código, y de haber un error se corregirán. Una vez terminado el código, se mostrarán los diagramas UML (Unified Modeling Language) y una documentación del diseño final.

Finalmente, en el capítulo de Resultados, se ponen a prueba todas aquellas funciones del software que sirven para analizar las respuestas de las variables del circuito. Para ello, se utilizan una gran variedad de circuitos, lineales, dinámicos, no-lineales, etc., muchos de ellos estudiados y puestos en práctica a lo largo del grado. Con la herramienta Pspice se comparan todos los resultados obtenidos. Teniendo ello como base, se determinará cuales son los puntos positivos y negativos del trabajo realizado.

CAPÍTULO 2

Teoría de Resolución de Circuitos

Para la creación de un programa informático que resuelva circuitos en un lenguaje de programación, es imprescindible comprender la Teoría de Circuitos (TC). Por lo tanto, en este capítulo se explicarán los procedimientos que se deben seguir para representar un circuito en forma matemática, y dependiendo de las características del mismo, desarrollar los métodos fundamentales para su análisis y resolución.

En la TC se describen las características de los elementos, y su clasificación; lineales/no-lineales, algebraicos/dinámicos, y dependiente/no-dependiente temporal. Después se determinan los pasos que se deben seguir para lograr la representación topológica del circuito, y dependiendo de las características de los elementos que lo componen, su clasificación, esto es; circuitos resistivos lineales, resistivos no-lineales, dinámicos lineales, y por último dinámicos no-lineales.

Finalmente, se detallarán los métodos que se usarán para la resolución de cada tipo de circuito, de entre los cuales se encuentra el método de Newton-Raphson, Laplace, y Euler.

2.1. Conceptos Básicos de la Teoría de Circuitos

En la TC, los elementos se caracterizan por tener puertos, y ecuaciones que representen el comportamiento físico del elemento real. Un puerto lo conforman un par de terminales, y la corriente que entra por una de ellas deberá salir por la otra. A las ecuaciones del elemento se les denomina ecuaciones constitutivas, y el número de estas será igual al la cantidad de puertos del elemento. Por otro lado, cada una de ellas dependerá de las corrientes y tensiones de todos sus puertos, esto es;

$$\begin{aligned} h_1(v_1, v_2, ..., v_n, i_1, i_2, ..., i_n, t) &= 0, \\ h_2(v_1, v_2, ..., v_n, i_1, i_2, ..., i_n, t) &= 0, \\ &\cdot \\ &\cdot \\ &\cdot \\ h_n(v_1, v_2, ..., v_n, i_1, i_2, ..., i_n, t) &= 0, \end{aligned} \tag{2.1}$$

donde n es igual al número de puertas.

En las variables de la expresión (2.1) no intervienen derivadas ni integrales, sin embargo existen ecuaciones de elementos con dependencia de v_j^α y/o i_j^α; en los casos de $\alpha > 0$ ($\alpha < 0$), el grado de la derivada (integral) sobre la variable v_j y/o i_j será igual a α. Esto último se aprecia en la ecuación del condensador lineal:

$$h_1(v_1^1, i_1) = i_1 - C\frac{dv_1}{dt} = 0 \tag{2.2}$$

donde C es igual a la capacitancia.

Los elementos se clasifican según las características de sus funciones; lineales/no-lineales, algebraicos/dinámicos, y dependientes del tiempo.

- Un elemento es lineal si al aplicar una excitación de corriente/tensión en la señal de entrada, la señal de salida cumple con el principio de superposición, por ejemplo; si al aplicar como señal de entrada las tensiones v_{i1} y v_{i2} se obtienen las señales de salida v_{o1} y v_{o2} respectivamente, el circuito será lineal si al aplicar $v_i = v_{i1} + v_{i2}$, en la salida se obtiene $v_o = v_{o1} + v_{o2}$. En caso contrario se dice que el elemento es no-lineal.
- Un elemento es dinámico si en alguna de sus ecuaciones aparece una variable de puerta con varios ordenes.
- Un elemento es invariante del tiempo si sus ecuaciones constitutivas no dependen explícitamente del tiempo.

Un circuito esta formado por elementos, cuyas terminales están interconectadas. A dichas conexiones se les denomina nodos, y a cada uno se le asocia un número, $i \in \{0, 1, 2, ..., n\}$ y una tensión, e_i. Una vez hecha la asignación numérica de los nodos, y con las ecuaciones bien definidas de los elementos que componen el circuito, es posible construir el digrafo; esquema gráfico constituido por nodos conectados mediante ramas dirigidas. Se creará una rama dirigida desde el $nodo_i$ al $nodo_j$, o viceversa, ambas opciones son validas, por cada puerta conectada a dichos nodos. A cada rama se le asociara un número, $j \in \{0, 1, 2, ..., m\}$, donde m es igual al número de puertas, y a cada una le corresponderá una corriente, i_j. Una vez construido el digrafo, y establecidas las variables del mismo $(e_0, e_1, ..., e_n, i_0, i_1, ..., i_m)$, es posible crear las ecuaciones de Kirchhoff (LK) independientes a partir de las leyes de corrientes (LCK) y de voltaje (LVK) de Kirchoff .

Un circuito está debidamente conectado si tomando el $nodo_i$ como punto de inicio es posible

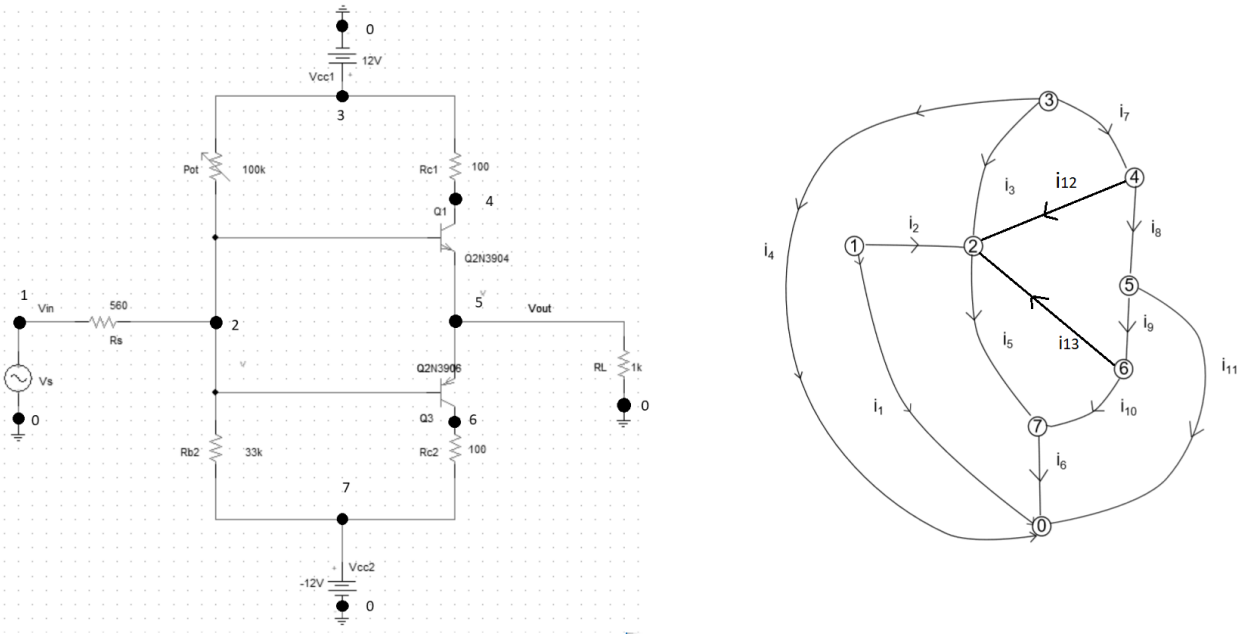


Figura 2.1: Ejemplo de un circuito y su digrafo.

llegar, mediante las ramas, al $nodo_j$. Por otro lado, se dice que un circuito está bien construido si no hay fuentes de tensión (corriente) independientes conectadas en paralelo (serie), ya que de realizar dichas conexiones no se cumplirían las leyes de Kirchoff.

LCK establece que si el circuito está debidamente conectado y construido, la suma de las corrientes entrantes y salientes de un nodo debe ser igual a cero. Por lo tanto, en el caso de un digrafo con $\mathbf{i} = (i_0, i_1, ..., i_m)^T$,

$$A * \mathbf{i} = \mathbf{0} \tag{2.3}$$

donde A es la matriz de incidencia y cuyos elementos son:

$$a_{ij} = \begin{cases} +1, \ si\ la\ rama\ j\ sale\ del\ nodo\ i \\ -1, \ si\ la\ rama\ j\ entra\ al\ nodo\ i \\ 0, \ si\ la\ rama\ j\ no\ toca\ el\ nodo\ i \end{cases} \tag{2.4}$$

Al analizar la matriz resultante, se observa que todas las columnas tienen únicamente un 1 y un -1, y por lo tanto una de las corrientes dependerá de las demás. En consecuencia, se escoge un nodo de referencia (normalmente se selecciona el $nodo_0$, nodo al que está conectado la tierra), y de ese modo su representación en la matriz A desaparece. A esta nueva A se le denomina matriz de incidencia reducida y su dimensión será igual a $(n-1) \times b$, siendo n el número de nodos, y b el de puertas.

LVK establece que si el circuito está debidamente conectado y construido, la diferencia de tensión, v_{ij}, en una rama que fluye del $nodo_i$ al $nodo_j$ debe ser igual a diferencia de tensión entre el par de nodos al que está conectada. Por lo tanto, al aplicar LVK sobre todas las ramas del digrafo y siendo el nodo de referencia el $nodo_0$, se cumple que,

$$\mathbf{v} = B * \mathbf{e}, \tag{2.5}$$

siendo $\mathbf{e} = (e_1, ..., e_n)^T$. Por otro lado los términos de la matriz deben cumplir con,

$$b_{ij} = \begin{cases} +1, \ si\ la\ rama\ j\ sale\ del\ nodo\ i \\ -1, \ si\ la\ rama\ j\ entra\ en\ el\ nodo\ i \\ 0, \ si\ la\ rama\ j\ no\ toca\ el\ nodo\ i \end{cases} \tag{2.6}$$

La dimensión de la matriz B es $b \times (n-1)$, y además $B = A^T$.

Las ecuaciones (2.3), y (2.5) establecen las relaciones entre las variables del digrafo. Sin embargo, para conocer sus valores son necesarias las ecuaciones de los elementos que lo componen. Al conjunto de ecuaciones, LK independientes más las de los elementos, se les designa el nombre de ecuaciones Tableau del circuito.

$$\begin{gathered} A * \mathbf{i} = \mathbf{0}, \\ \mathbf{v} - A^T * \mathbf{e} = \mathbf{0}, \\ \mathbf{h} = \mathbf{0} \end{gathered} \tag{2.7}$$

siendo $\mathbf{h} = (h_1, h_2, ..., h_b)^T$.

El método que se vaya aplicar sobre sus ecuaciones Tableau dependerá del tipo de circuito. Existen 4 clases principales de circuitos, resistivos lineales, resistivos no-lineales, dinámicos lineales, y dinámicos no-lineales, y su clasificación depende de los elementos que lo componen, esto es;

- Un circuito es resistivo lineal (RL) si está compuesto por elementos lineales (incluyendo fuentes independientes) cuyas ecuaciones únicamente contienen variables de orden cero, como por ejemplo resistencias, fuentes independientes, fuentes controladas, etc.

- Un circuito es resistivo no-lineal (RNL) si está compuesto al menos por un elemento no-lineal (excluyendo las fuentes independientes), y sus ecuaciones únicamente contienen variables de orden cero.

- Un circuito es dinámico lineal (DL) si está compuesto por elementos lineales (incluyendo elementos con excitaciones independientes), y sus ordenes son distintos, esto es; en las ecuaciones Tableau aparece al menos una variable con distintos ordenes.

- Un circuito es dinámico no-lineal (DNL) si está compuesto por elementos no-lineales (excluyendo los elementos con excitaciones independientes), y sus ordenes son distintos.

2.2. Solución de Circuito Resistivos Lineales

Si el circuito es del tipo RL el conjunto de ecuaciones constitutivas se puede representar de la siguiente forma;

$$\begin{pmatrix} M & N \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ \mathbf{i} \end{pmatrix} = \mathbf{u} \tag{2.8}$$

donde M y N son matrices, y $\mathbf{u}$ vectores con los valores de las fuentes independientes. Por lo tanto, la forma matricial de las ecuaciones Tableau es la que sigue:

$$\begin{pmatrix} 0 & 0 & A \\ -A^T & 1 & 0 \\ 0 & M & N \end{pmatrix} \begin{pmatrix} \mathbf{e} \\ \mathbf{v} \\ \mathbf{i} \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{u} \end{pmatrix} \tag{2.9}$$

Una vez que se consigue la expresión (2.9) de un circuito, es posible calcular los valores de las variables usando métodos como el de Kramer, Gauss, etc.

2.3. Solución de Circuitos Resistivos No-lineales

Cuando se trata de un circuito RNL es imposible obtener la expresión 2.9. Una alternativa para lograr los valores de las variables del circuito es aplicar métodos numéricos que manipulen las ecuaciones constitutivas, de tal forma que se logre una solución aproximada. Existen una gran variedad de métodos numéricos que consiguen dicho objetivo, como por ejemplo el método de bisección, falsa posición, Newton-Raphson, Secante, etc.

Las formulas de bisección y falsa posición se caracterizan por ser cerrados (necesitan de un intervalo que contenga la raíz), estables, y tener un coste computacional alto. Por otro lado, los métodos de Newton-Raphson y secante son abiertos (necesitan un punto inicial). Convergen rápidamente cuando el punto inicial está próximo a la solución, en caso contrario tienden a ser muy inestables; divergen o se alejan de la raíz.[12]

Cuando la linealidad depende de más de una variable, como normalmente ocurre en las ecuaciones de los circuitos, la búsqueda de un intervalo donde se halle la raíz se complica. Es por eso que se descarta el uso de los métodos de bisección y falsa posición.

El método de la secante, a diferencia de Newton-Raphson, usa procedimientos numéricos para obtener una aproximación de la derivada [12, 10]. Dado que los elementos de los circuitos tienen funciones bien definidas, es posible calcular sus derivadas manualmente. Por lo tanto, es favorable optar por Newton-Raphson ya que computacionalmente es mas eficiente.

2.3.1. Método de Newton-Raphson para Ecuaciones No-lineales

Partiendo de que la búsqueda de una raiz en casos de ecuaciones lineales es trivial, el método de Newton-Raphson tiene como base obtener una función lineal que se aproxime a la

función original, $f(x)$. Esto último se consigue mediante el teorema de aproximación de Taylor de grado uno. Por lo tanto, al aplicar Taylor sobre el punto x_i, punto obtenido en la i-enesima iteración, se obtiene:

$$F(x) = f(x_i) + f'(x_i)(x - x_i) \tag{2.10}$$

si para x_{i+1} se cumple que $F(x_{i+1}) = 0$,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \tag{2.11}$$

y el error es proporcional al cuadrado del error anterior,

$$E_{i+1} = \frac{f''(x')}{2f'(x_i)} E_i^2 \tag{2.12}$$

Sin embargo, la expresión 2.11 no siempre consigue que x_{i+1} converja a x', raíz de f(x). Analizándola, se observa que x_{i+1} diverge cuando $f'(x_i) = 0$; al ser la pendiente de $F(x)$ nula, $F(x)$ nunca se cruzará con el eje x. Por otro lado, si x_i se encuentra cerca de un máximo o mínimo, x_{i+1} saltará a una posición lejana a la raíz. Estos problemas decrecen al escoger un punto inicial cercano a la solución.

Puesto que no existe un criterio que asegure la convergencia, los programas que implementen la formula de Newton-Raphson deben verificar si las soluciones obtenidas se alejan de x_n, y parar en caso de que así ocurra, o imponer un número máximo de iteraciones.

El algoritmo de Newton-Raphson en pseudocódigo es el que sigue,

```
1
2  Funcion newtonRaphson(f, fd, x0)
3
4          # e (precision), y N (numero maximo de iteraciones)
5          # debe seleccionarlo el desarrollador
6
7          xi = x0;
8          f1 = f(xi);
9          i = 0;
10
11         Mientras (f1 > e o i < N) Hacer
12
13                 x = xi + f(xi)/fd(xi);
14                 f1 = f(x);
15                 xi = x;
16                 i = i + 1:
17
18         Fin Mientras
19
```

```
        Escribir x;

Fin Funcion
```

Para aplicar este método de Newton-Raphson en la TC, se deben sustituir las ecuaciones constitutivas de los elementos no lineales por sus ecuaciones linealizadas. La linealización se realizará sobre un punto $\mathbf{x^j} = (x_1^j, x_2^j, ..., x_n^j)$, donde n es igual al número de puertas del elemento, y j en estos casos determina la iteración. Después, se obtendrá la solución de las variables del circuito mediante (2.9), y de no cumplirse el criterio de precisión, a cada elemento no-lineal se le dará un nuevo punto $\mathbf{x^{j+1}}$, obtenido de la ultima solución, para volver a linealizar sus ecuaiones.

Los elementos no-lineales más comunes son los diodos, y los transistores bipolares (BJT). En el caso del diodo su ecuación constitutiva es la que sigue:

$$i_1 = I_0 e^{(v_1/v_t)} \quad : v_t = 8{,}61 * 10^{-5} t \tag{2.13}$$

donde el valor de I_0 depende del tipo de diodo, y v_1 e i_1 son las variables de puerta. Al linealizar la ecuación (2.13) sobre un valor v_1^j se obtiene la siguiente ecuación:

$$-\frac{I_0}{v_t} e^{(v_1^j/v_t)} v_1^{j+1} + i_1^{j+1} = I_0(e^{(v_1^j/v_t)} - 1) - \frac{I_0 v_1^j}{v_t} e^{(v_1^j/vt)} \tag{2.14}$$

Al analizar (2.14) se observa que al linealizar la ecuación, el diodo se está sustituyendo por una fuente de corriente y una conductancia. Para los transistores bipolares se ha usado el modelo de Ebers-Moll. Por lo tanto en el caso de un NPN la ecuación constitutiva es la siguiente:

$$\begin{aligned} i_1 &= \alpha_F I_{Es}(e^{(v_2/v_t)} - 1) - I_{Cs}(e^{(v_1/v_t)} - 1) \\ i_2 &= -\alpha_R I_{Cs}(e^{(v_1/v_t)} - 1) + I_{Es}(e^{(v_2/v_t)} - 1) \end{aligned} \tag{2.15}$$

y para los PNP,

$$\begin{aligned} i_1 &= \alpha_F I_{Es}(e^{(-v_2/v_t)} - 1) - I_{Cs}(e^{(-v_1/v_t)} - 1) \\ i_2 &= -\alpha_R I_{Cs}(e^{(-v_1/v_t)} - 1) + I_{Es}(e^{(-v_2/v_t)} - 1) \end{aligned} \tag{2.16}$$

donde, en ambos casos $v_t = 8{,}61 * 10^{-5} * t$, y los valores de α_F, α_R, I_{Es}, y I_{Cs} dependen del tipo de transistor. Por otro lado, $v_1 \equiv v_{BC}$, y $v_2 \equiv v_{BE}$ son las tensiones del primer puerto, y el segundo, respectivamente. Al linealizar la ecuación (2.15) sobre el punto $\mathbf{v^j} = (v_1^j, v_2^j)$,

$$\begin{aligned} G_2 v_1^{j+1} - \alpha_F G_1 v_2^{j+1} - i_1^{j+1} &= \alpha_F I_1 - I_2, \\ -\alpha_R G_2 v_1^{j+1} + G1 v_2^{j+1} - i_2^{j+1} &= \alpha_R I_2 - I_1, \end{aligned} \tag{2.17}$$

donde,

$$\begin{aligned} G_1 &= \tfrac{I_{Es}}{v_t} e^{v_2^j/v_t}, \quad I_1 = I_{Es}\left[(e^{(v_2^j/v_t)} - 1) - \tfrac{v_2^j}{v_t} e^{(v_2^j/v_t)}\right], \\ G_2 &= \tfrac{I_{Cs}}{v_t} e^{v_1^j/v_t}, \quad I_2 = I_{Cs}\left[(e^{(v_1^j/v_t)} - 1) - \tfrac{v_1^j}{v_t} e^{(v_1^j/v_t)}\right]. \end{aligned} \tag{2.18}$$

En el caso del transistor PNP:

$$\begin{aligned} -G_2 v_1^{j+1} + \alpha_F G_1 v_2^{j+1} + i_1^{j+1} &= \alpha_F I_1 - I_2, \\ \alpha_R G_2 v_1^{j+1} - G1 v_2^{j+1} + i_2^{j+1} &= \alpha_R I_2 - I_1, \end{aligned} \tag{2.19}$$

donde,

$$\begin{aligned} G_1 &= \tfrac{I_{Es}}{v_t} e^{-v_2^j/v_t}, \quad I_1 = I_{Es}\left[(e^{(-v_2^j/v_t)} - 1) + \tfrac{v_2^j}{v_t} e^{(-v_2^j/v_t)}\right], \\ G_2 &= \tfrac{I_{Cs}}{v_t} e^{-v_1^j/v_t}, \quad I_2 = I_{Cs}\left[(e^{(-v_1^j/v_t)} - 1) + \tfrac{v_1^j}{v_t} e^{(-v_1^j/v_t)}\right]. \end{aligned} \tag{2.20}$$

Del mismo modo que ocurre con el diodo, al linealizar los transistores, su representación en el circuito se reemplaza por conductancias, y fuentes de corriente independientes.

2.3.2. Método de Newton-Raphson para Sistema de Ecuaciones No-lineales

Cuando se trata de un sistema de ecuaciones no lineales,

$$\begin{aligned} f_1(x_1, x_2, ..., x_n) &= 0, \\ f_2(x_1, x_2, ..., x_n) &= 0, \\ &\vdots \\ f_n(x_1, x_2, ..., x_n) &= 0, \end{aligned} \tag{2.21}$$

la solución, $\mathbf{x}' = (x_1', x_2', ..., x_n')$, hará que simultáneamente todas las ecuaciones que componen el sistema sea igual a cero, esto es; $\mathbf{f}(\mathbf{x}') = \mathbf{0}$.

El método de Newton-Raphson para la solución de una sola ecuación se puede aplicar a casos de múltiples ecuaciones. Sin embargo, al usar la aproximación de Taylor, hay que tener en cuenta que la función es vectorial; la derivada de una sola variable debe sustituirse por el Jacobiano ($J_{\mathbf{f}}(\mathbf{x})$).

$$\mathbf{F}(\mathbf{x}) = \mathbf{f}(\mathbf{x_i}) + J_{\mathbf{f}}(\mathbf{x}_i)(\mathbf{x} - \mathbf{x}_i) \tag{2.22}$$

y teniendo en cuenta que para $\mathbf{x}_{i+1}$ se cumple $\mathbf{F}(\mathbf{x}_{i+1}) = \mathbf{0}$,

$$\mathbf{0} = \mathbf{f}(\mathbf{x_i}) + J_{\mathbf{f}}(\mathbf{x}_i)(\mathbf{x}_{i+1} - \mathbf{x}_i). \tag{2.23}$$

Para lograr $\mathbf{x}_{i+1}$ primero hay que obtener $\Delta\mathbf{x} = \mathbf{x}_{i+1} - \mathbf{x}_i$ del siguiente sistema:

$$J_{\mathbf{f}}(\mathbf{x}_i)\Delta\mathbf{x} = -\mathbf{f}(\mathbf{x}_i). \tag{2.24}$$

De esta forma se consigue que $\mathbf{x}_{i+1}$ sea igual a la suma de los vectores $\Delta\mathbf{x}$ y $\mathbf{x}_i$:

$$\mathbf{x}_{i+1} = \Delta\mathbf{x} + \mathbf{x}_i \tag{2.25}$$

Del mismo modo que ocurre con las funciones escalares, al aplicar el método de Newton-Raphson para un sistema de ecuaciones no-lineales, la convergencia se ve afectada cuando el punto inicial no se encuentra cerca de la solución. Esto último supone un gran inconveniente, ya que determinar para cada $x_i : i \in n$ un valor cercano a x_i' es una tarea compleja.

El método de Newton amortiguado proporciona robustez al método original. Se basa en alterar el valor de $\mathbf{x}_{i+1}$ con la variable w_i [10], esto es:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + w_i\Delta\mathbf{x}_{i+1} \tag{2.26}$$

Un valor de w_i es valido, respecto cualquier valor de w, si al aplicarlo en la expresión 2.26 minimiza la norma de la función vectorial:

$$\|\mathbf{f}(\mathbf{x}_i + w_i\Delta\mathbf{x}_{i+1})\| \ll \|\mathbf{f}(\mathbf{x}_i + w\Delta\mathbf{x}_{i+1})\| : \forall w \in \Re \tag{2.27}$$

Para aplicar este método sobre un circuito, es necesario utilizar un método de aproximación numérica, como por ejemplo *Euler*[1], para el cálculo de $J_{\mathbf{f}}$.

2.4. Solución de Circuitos Dinámicos Lineales

En estos casos de un circuitos DL, la forma matricial de las ecuaciones constitutivas de los elementos cambia[2] :

$$\sum_{j\in B}\begin{pmatrix} M_j & N_j \end{pmatrix}\begin{pmatrix} \mathbf{v}^{(j)} \\ \mathbf{i}^{(j)} \end{pmatrix} + \begin{pmatrix} M & N \end{pmatrix}\begin{pmatrix} \mathbf{v} \\ \mathbf{i} \end{pmatrix} = \mathbf{u} \ : B = \{\beta_1, \beta_2, ...\}. \tag{2.28}$$

donde (j) indica el orden de las variables. Por lo tanto, es necesario lograr un sistema equivalente resistivo y lineal para conseguir unas ecuaciones Tableau con la forma de 2.8.

[1] El funcionamiento de Euler se explica en el apartado *Método de Euler*

[2] Se supondrá que los ordenes de los elementos siempre serán negativos, ya que, en caso contrario, es posible derivar todas sus variables de puerta para reducir el orden positivo a 0.

Existen varios métodos que logran la solución de sistemas de ecuaciones diferenciales. Uno de ellos es aplicar la transformada de Laplace, y se caracteriza por necesitar una notación simbólica. Su eficiencia depende de la capacidad de otros métodos para lograr la solución de ecuaciones algebraicas, que contengan expresiones simbólicas. Es de gran importancia la transformada de Laplace en la TC, ya que haciendo uso de la expresiones simbólicas, es posible ofrecer una gran variedad análisis; diagramas de polos, bode, etc.[Referencia]

Los métodos de Euler y Runge-Kutta son numéricos y se usan para la aproximación de derivadas. Se caracterizan por ser de un paso; el valor de la derivada debe ser calculado paso a paso. Runge-Kutta es una versión mejorada de Euler, pues reduce el error, y por ende es uno de los métodos mas usados en esta clase de problemas. Cabe mencionar que si se opta por uno de estos dos métodos, y el orden de las ecuaciones es mayor que uno, para poder implementarlos, es posible aplicar cambios de variables hasta lograr ecuaciones de orden uno[12, 10].

2.4.1. Método de Euler

Ante una ecuación de la siguiente forma:

$$g(x,t) = \frac{dx}{dt} \tag{2.29}$$

para obtener su solución es necesaria una aproximación de dx/dt.

El desarrollo de Taylor, de orden uno, ofrece una aproximación lineal de cualquier función haciendo uso de los valores x_i, x_{i+1} y dx/dt. Por lo tanto, despejando dx/dt es posible obtener una estimación de su valor:

$$\frac{dx}{dt} = \frac{x_{i+1} - x_i}{t_{i+1} - t_i} = \frac{x_{i+1} - x_i}{h} + \frac{O(h^2)}{h}, \tag{2.30}$$

donde h se denomina paso, y se mantiene constante para cada cálculo de dx/dt. Sustituyendo la expresión 2.30 en 2.29 se obtiene la formula de Euler progresiva:

$$x_{i+1} = x_i + hg(x_i, t_i) + O(h^2) \tag{2.31}$$

En cada paso se crea un error, conocido como error de truncamiento local (E_l), proporcional a h^2. Sin embargo, a medida que se aplica 2.31, E_l se transmite; en el paso n, x_{n+1} tendrá un error proporcional a n veces el error de truncamiento local. A este último error se le denomina error de truncamiento propagado (E_p). Por lo tanto, el error total (E_t^n) en el último paso es

$$E_t = E_l + E_p \propto h^2(1 + 1/h). \tag{2.32}$$

Si se escoge un valor de h pequeño E_t será proporcional a h, y en consecuencia reduciendo el tamaño del paso se reduce el error. Por otro lado, la convergencia de este método tiene una

gran dependencia del valor de h. Si el tamaño del paso no es lo suficiente pequeño las soluciones acaban divergiendo.

A la hora de aplicar el desarrollo de Taylor es posible evaluarlo en x_{i-1}, y siguiendo el mismo procedimiento que con x_{i+1} se consigue la siguiente expresión:

$$x_i = x_{i-1} + hg(x_i, t_i) + O(h^2). \tag{2.33}$$

Aplicando los cambios de variable $x_{i-1} = x_i$ y $x_i = x_{i+1}$, se obtiene la formula de Euler regresiva:

$$x_{i+1} = x_i + hg(x_{i+1}, t_{i+1}) + O(h^2). \tag{2.34}$$

Al aplicar 2.34 la convergencia no se ve tan afectada por el valor de h, y en consecuencia la fórmula regresiva se emplea con mas frecuencia. Sin embargo, si $g(x, t)$ fuera una función no-lineal, el método regresivo necesitaría de otros métodos como el de Newton-Raphson para obtener x_{i+1}.

El algoritmo de Euler progresivo y regresivo en pseudocódigo es el que sigue,

```
1
2  Funcion eulerProgresivo(f, t1, t2, pasos)
3
4          h = (t2-t1)/pasos;
5          i = 1;
6
7          Mientras (i < pasos) Hacer
8
9                  t1 = t1 + i*h;
10                 x2 = x1 + h*f(x1, t1);
11                 x1 = x2;
12                 i = i + 1;
13                 Escribir x2;
14         Fin Mientras
15
16 Fin Funcion
17
18
19 Funcion eulerRegresivo(f, fd, t1, t2, pasos)
20
21         h = (t2-t1)/pasos;
22         i = 1;
23
24         Mientras (i < pasos) Hacer
25
26                 t2 = t1 + i*h;
```

```
27                  x2 = x1 + h*f(x2, t2);
28                  g = x2 - x1 - h*f(x2, t2);
29
30                  Si g == no-lineal Entonces
31                          gd = 1 - h*fd(x2, t2);
32                          x2 = newtonRaphson(g, gd, x0);
33                  Fin Si
34
35                  x1 = x2;
36                  i = i + 1;
37                  Escribir x2;
38
39          Fin Mientras
40
41  Fin Funcion
```

Para resolver circuitos DL, a las ecuaciones de los elementos dinámicos se les aplicará (2.31), o (2.33), y entonces sera posible la obtención de las ecuaciones Tableu en forma matricial (2.9).

El condensador es un elemento dinámico muy común, y al aplicar Euler regresivo sobre (2.2) su ecuación constitutiva quedaría de la siguiente forma:

$$h(v_1^{t+1}, i_1^{t+1}) = v_1^{t+1} - \frac{h}{C} i_1^{t+1} - v_1^t = 0, \tag{2.35}$$

donde v_1^{t+1}, e i_1^{t+1} serán los valores de las variables de puerta del elemento en el tiempo $t+h$. La ecuación (2.35) es no-lineal, porque contiene excitaciones independientes, y el orden de todas sus variables igual a cero. Además, se puede observar que el condensador continuo en el tiempo, se ha sustituido por una resistencia y una fuente de tensión independiente conectadas en paralelo.

Los inductores son otro caso común de elementos dinámicos, y sus ecuaciones constitutivas cumplen con la siguiente estructura:

$$h(v_1, i_1^{(1)}) = v + L\frac{di}{dt} \tag{2.36}$$

y al aplicar Euler regresivo,

$$h(v_1^{t+1}, i_1^{t+1}) = i_1^{t+1} - \frac{h}{L} v_1^{t+1} - i_1^t. \tag{2.37}$$

Del mismo modo que ocurre con el condensador, la nueva ecuación constitutiva del inductor (2.37) es resistiva y no lineal, y se puede interpretar como una conductancia y una fuente de corriente independiente conectadas en paralelo.

2.5. Solución de Circuitos Dinámicos No-Lineales

Para la solución de circuitos de la clase DNL, es necesario combinar los métodos de Euler, y Newton-Raphson. Para un tiempo t se aplicará Euler sobre las ecuaciones de los elementos dinámicos, y después los elementos no-lineales darán sus ecuaciones linealizadas para aplicar Newton-Raphson, y conseguir una aproximación de la solución. Cuando el criterio de precisión se cumpla se volverá a realizar el mismo procedimiento para el siguiente paso en el tiempo, $t + h$.

CAPÍTULO 3

Implementación

Se puede plantear el diseño de un programa sin que el lenguaje con el que se va a realizar este concretado. Sin embargo, es preferible revisar las diferentes opciones de lenguajes de programación que existen, y todo lo que engloban. Por todo ello, en la sección Opciones de Diseño, se explicaran las diferentes generaciones, y se describirán las características y extensiones de los lenguajes impartidos en el grado de Física e Ingeniería Electrónica, esto es; Fortran, Java y Python.

En el apartado *Diseño Selecionado*, se discutirán las ventajas y desventajas de cada lenguaje de programación, y se seleccionará el que mejor se adapte al problema. Después, se abordará la forma del programa, y los pasos que se deben realizar para su elaboración.

Finalmente, se describirán los problemas que han ido surgiendo, y en consecuencia las soluciones que se han ideado. Una vez finalizado el proceso de creación, se ofrecerá una breve descripción del diseño final, su documentación, y diagramas UML.

3.1. Opciones de Diseño

Un lenguaje de programación es un lenguaje escrito con el que es posible escribir una agrupación de instrucciones. Una instrucción se reduce a una serie de ordenes que la maquina debe cumplir, con el fin de lograr el objetivo dado por la instrucción. Se denomina algoritmo al conjunto de instrucciones, distribuidas de forma clara y ordenada, que dan solución a un problema.

La base de la escritura en los lenguajes de programación lo conforman los elementos léxicos, la sintaxis, y la semántica, y varían según el lenguaje que se este utilizando. La sintaxis establece la forma en la que se deben combinar los elementos, mientras que la semántica establece el significado; se pueden dar casos en los que el código sea sintácticamente correcto pero no

semánticamente[7]. Sin embargo, para la elaboración de un programa no basta con la escritura del código; es necesario ejecutar las instrucciones, corregir los errores emergentes, testearlo para verificar que se cumple con todos los objetivos, y crear la documentación[11].

A la hora de barajar los posibles lenguajes de programación para este proyecto, únicamente se han tenido en cuenta los lenguajes de alto nivel que se han impartido en el grado de Ingeniera Electrónica y Física de la UPV/EHU, entre los cuales se encuentran Fortran, Python, y Java (en el Cuadro 3.1 se listan algunas de sus características). Aunque su implementación en *hardware* sea menos eficiente en comparación con los lenguajes ensambladores y de máquina [7], compensa el hecho de tener una base teórica/practica de ellos, y que sean más simples.

Uno de los paradigmas de los lenguajes de alto nivel es el orientado a objetos, se caracterizan por tener un nivel alto de abstracción, esto es: la forma de programar se acerca más al problema real. Para ello se crean tipos de dato, llamadas clases/objetos, definidos por una serie de atributos, métodos, etc. A la hora de programar, es necesario establecer los objetos que se quieren crear y después definir las interacciones que tendrán entre si. Otro paradigma es la programación por procesos, y se basan en tener funciones con una serie de instrucciones a ejecutar. Además, desde una función se permite la llamada de otras funciones.

Los lenguajes del tipo compilado, precisan de un proceso de compilación en el que se analiza el código fuente; primero examina los elementos, luego realiza un análisis sintáctico, y por último comprueba si semánticamente está bien construido (compatibilidad de datos, valores de entrada de los métodos bien definidos, ...). Una vez terminado el análisis se crea un código intermedio, y este se enlaza con otros códigos situados en la biblioteca para producir un código ejecutable. En los lenguajes interpretados las instrucciones se ejecutan una tras otra, si el código lo requiere.

Fortran, creado en 1957, es un lenguaje del tipo compilado y por procesos[11]. La página The Fortran Company [5] presenta una lista, bastante amplia, de librerías, muchas de ellas con enfoque matemático; LINPACK, y MINPACK para la resolución de ecuaciones lineales y no-lineales, respectivamente, ODEPACK para la solución de ecuaciones diferenciales ordinarias. La página indica que la documentación se realiza con la librería DOCTRAN.

Java es un lenguaje de programación orientado a objetos y del tipo compilado[6]. La pagina oficial de Oracle ofrece un gran servicio en relación a Java. Presenta una API (Application Program Interface) de la plataforma: lista de las clases, paquetes, etc, que ofrece la compañía, junto con una descripción de como interactuan. También proporciona la herramienta Javadoc, para la creacion de un API, y sus normas de uso. En cuanto a librerías de aplicaciones matemáticas, uno debe buscarlas externamente. JScience, entre otros, ofrece herramientas y librerías para aplicaciones científicas, y en ella podemos encontrar módulos de álgebra lineal, representación de funciones, etc.

Python, creado en 1991, es un lenguaje orientado a objetos y del tipo interpretado[14]. La pagina web de Python [3] es clara y completa. Ofrece multitud de herramientas para su compresión y uso: APIs, referencias de sus librerías, una wiki, etc. En su wiki se pueden encontrar librerías de ámbito científico, como SciPy, NumPy, Sympy, etc.

	FORTRAN	Java	Python
IDE	Code::Blocks	NetBeans	Spyder
Paradigma	orientado a objetos	orientado a objetos	por procesos
interprete/compilador	compilador	compilador	interprete
Librerías	LINPACK, MINPACK,...	JScience, JAMA,...	numpy, scipy,...
Gráficos	gnufortran	FreeChart	Matplotlib

Cuadro 3.1: Resumen de los lenguajes de programacion analizados.

3.2. Diseño seleccionado

3.2.1. Seleccion del Lenguaje de Programacion

Es muy importante seleccionar el lenguaje que mejor se adapte al problema y al programador, ya que de esa manera se facilita la rapidez a la que se programa, y la optimización de la solución. En el proceso de selección se tendrán en cuenta los paradigmas, las facilidades que proporciona, la usabilidad, y los tiempos de ejecución.

Desde el punto de vista de un lenguaje orientado a objetos, es fácil plantear la construcción de un software que simule el comportamiento de un circuito. Por ejemplo, los elementos serían objetos, y sus atributos representarían las características físicas del elemento al que representa. Esta clase de abstracción hace que su uso y entendimiento sea mas sencillo para otros programadores. En el caso de una programación por procesos/funcional, la estructura no sería tan trivial. Además, En los lenguajes orientados a objetos, las clases pueden heredar atributos y funciones de otras clases, y esto proporciona una gran rapidez a la hora de desarrollar y aumentar la magnitud del proyecto. Aunque en Fortran 90 se puedan realizar diseños que, de alguna manera, representen un lenguaje orientado a objetos, no cumple con las expectativas que puede cumplir C++, lenguaje totalmente orientado a objetos [8]. Fortran, pese a ser una herramienta científica muy potente, queda descartado por el momento.

Tal y como se ha manifestado en el capítulo 2, la transformada de Laplace es una herramienta muy importante, dado que ofrece un abanico de posibles análisis. Por lo tanto, de querer realizar operaciones en el espacio s, sería necesario un instrumento que permitiera usar y solucionar ecuaciones simbólicas. Python y Java ofrecen ese servicio con la librería Sympy, y la clase Polynomial, respectivamente. Con el método *solve_linear_system_LU()* de Sympy se obtiene la solución de un sistema de ecuaciones lineales con y sin expresiones simbólicas. Sin embargo, esa opción no es válida para la clase polynomial.

Como se puede observar en la Figura 3.1 el código de Java es sintácticamente más complejo. Esta característica hace que sus códigos sean limpios, tengan menores ambigüedades, y ofrecen mayores posibilidades. Sin embargo, dificulta su aprendizaje y uso. Python demuestra una sintaxis simple e intuitiva, y en consecuencia un mejor aprendizaje y rapida implementación implementación [9].

Una desventaja de Python reside en los tiempos de ejecución. Python es mas lento y esta diferencia proviene del tipo de lenguaje, compilador/interprete. La compilación es un proceso

```java
// Defines local variables.
Variable.Local<Rational> varX = new Variable.Local<Rational>("x");
Variable.Local<Rational> varY = new Variable.Local<Rational>("y");

// f(x, y) =  x² + x·y + 1;
Polynomial<Rational> x = Polynomial.valueOf(Rational.ONE, varX);
Polynomial<Rational> y = Polynomial.valueOf(Rational.ONE, varY);
Polynomial<Rational> fx_y = x.pow(2).plus(x.times(y)).plus(Rational.ONE);
System.out.println("f(x,y) = " + fx_y);

// Evaluates f(1,0)
System.out.println("f(1,0) = " + fx_y.evaluate(Rational.ONE, Rational.ZERO));

// Calculates df(x,y)/dx
System.out.println("df(x,y)/dx = " + fx_y.differentiate(varX));

> f(x,y) = [1/1]x^2 + [1/1]xy + [1/1]
> f(1,0) = 2/1
> df(x,y)/dx = [2/1]x + [1/1]y
```

(a) Java

```python
from sympy import*

#Defines local variables
x, y=symbols("x y")

#f(x, y)=x^2+x*y+1
f=Function("f")
f=x**2+x*y+1
print("f=", f)

#Evaluates f(1, 0)
print("f(1, 0)=", f.evalf(subs={x:1, y:0}))

#Calculates df(x, y)/dx
print("df(x, y)/dx=", f.diff(x))

f= x**2 + x*y + 1
f(1, 0)= 2.00000000000000
df(x, y)/dx= 2*x + y
```

(b) Python

Figura 3.1: Ejemplo del uso de expresiones simbolicas en Java y Python. El código de Java ha sido obtenido de JScience[1].

lento, pero solo hay que realizarlo una vez, y ejecutar el ejecutable no suele suponer un coste temporal alto. En cambio, cada proceso de interpretación supone un coste alto, y debe repetirse cada vez que se ejecute. En pequeños programas no es ningún problema, pero a medida que aumenta la envergadura del proyecto puede suponer un gran dilema.

Si se considera el lanzamiento público de este proyecto software, es combeninete elegir un lenguaje que no este obsoleto, o que se este dejando de usar. El ranking de la Figura 3.2 usa 12 fuentes de datos: busquedas en Google, preguntas en Stack Overflow, GitHub, etc. Tal y como se puede observar, Python encabeza la lista seguido de C++ y Java.

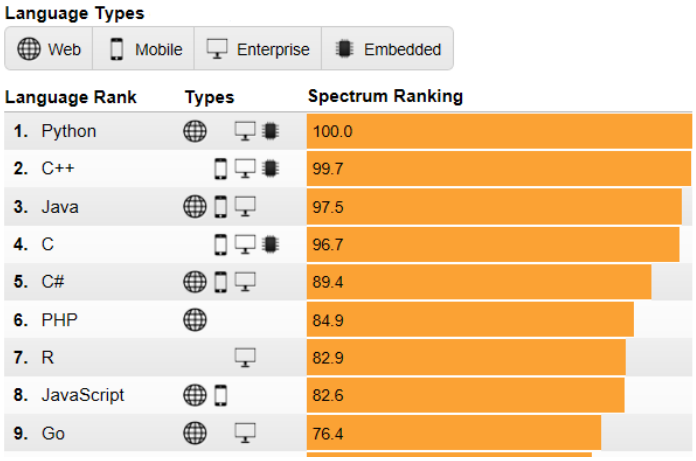


Figura 3.2: Ranking de los 9 lenguajes de programación mas usados. Imagen obtenida de IEEE SPECTRUM

Una vez comparados, principalmente, Java y Python, se ha optado por el uso de Python. Son muchas sus ventajas: es un lenguaje orientado a objetos, que ayuda a simular el problema físico, pero gracias a su multi-abstracción es capaz de comportarse como uno funcional, ideales para una computación más científica. Además, su usabilidad es muy superior a la de Java.

3.2.2. Diseño del Programa

Teniendo en cuenta la Teoría de Circuitos, y que el lenguaje seleccionado está orientado a objetos, las bases del programa serán las clases *Elements*, y *Circuit*. *Elements* representará los elementos del circuito, sus atributos caracterizarán el elemento y sus métodos principales se usarán para comunicar *Elements* con *Circuit*. Siguiendo la clasificación de los elementos explicado en el apartado *Conceptos Básicos de la Teoría de Circuitos*, se diferenciarán 3 clases, *NotLineal*, *Differential*, y *Resistive*, cada una tendrá unos atributos y métodos propios, pero heredaran las características de *Elements*. Esto último se puede observar en la Figura 3.3.

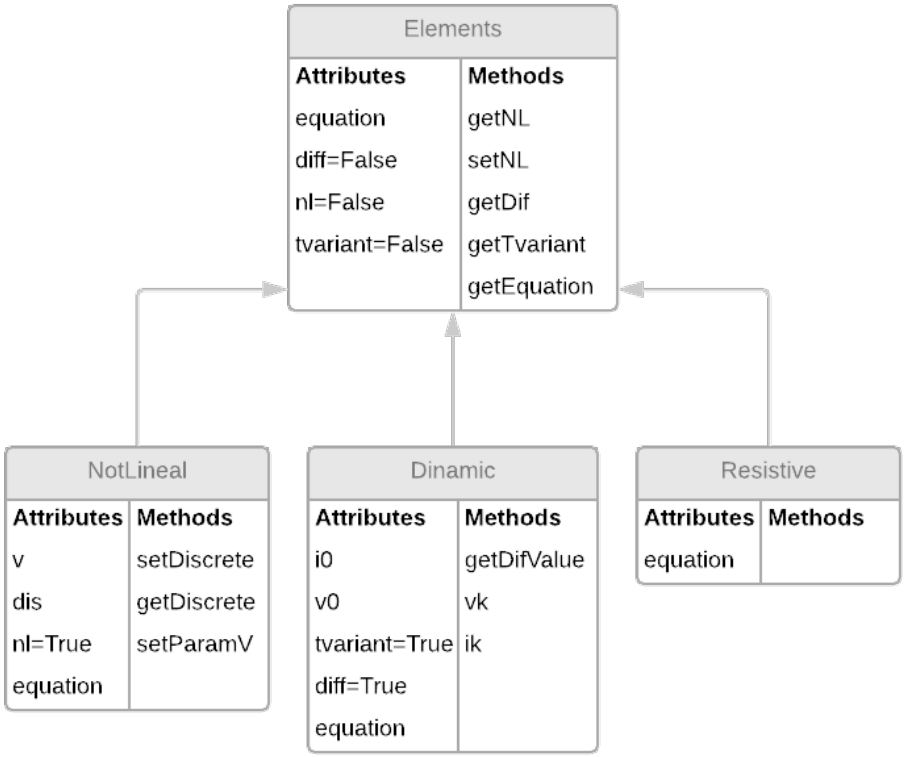


Figura 3.3: Diagrama en el que se muestran los atributos, y métodos principales de las clases *Elements*, *NotLineal*, *Differential*, y *Resistive*. La clase *Elements* y *Resistive* son equivalentes, sin embargo esta última se ha incluido para aportar claridad en la explicación del diseño.

Los métodos de *Elements* se usarán para extraer información de los atributos o para modificarlos. Los atributos *diff*, *nl*, y *tvariant* tendrán valores de *True*/*False* para determinar si el objeto es dinámico, no-lineal, y/o dependiente del tiempo, respectivamente. Por otro lado, *equation* representará las funciones constitutivas del elemento, y se usará una matriz para representarlo: la fila i representará la ecuación del puerto i, las columnas 1 y 2 serán listas con los términos asociados a los voltajes y corrientes de puerta, respectivamente, del elemento, y la columna 3 será otra lista con el valor independiente:

$$\begin{array}{l} a_{11}v_1 + a_{12}v_2 + \cdots + b_{11}i_1 + b_{12}i_2 + \cdots = u_1 \\ a_{21}v_1 + a_{22}v_2 + \cdots + b_{21}i_1 + b_{22}i_2 + \cdots = u_2 \\ \cdots \\ a_{n1}v_1 + a_{n2}v_2 + \cdots + b_{n1}i_1 + b_{n2}i_2 + \cdots = u_n \end{array} \equiv \left[\begin{array}{ccc} [a_{11}, a_{12}, \cdots] & [b_{11}, b_{12}, \cdots] & [u_1] \\ [a_{21}, a_{22}, \cdots] & [b_{21}, b_{22}, \cdots] & [u_2] \\ \cdots & \cdots & \cdots \\ [a_{n1}, a_{n2}, \cdots] & [b_{n1}, b_{n2}, \cdots] & [u_3] \end{array} \right], \quad (3.1)$$

donde v_{ij} e i_{ij} son las variables internas del elemento. Más tarde se verá que la ecuación 3.1 se podrá usar para representar tanto ecuaciones lineales, como no-lineales.

Un objeto de la clase *Elements* necesitará de *equation* para su instanciación (si no se desean los valores por defecto de los demás atributos, véase Figura 3.3, en el momento de instanciación se podrán redefinir).

Circuit representará un circuito, y sus métodos principales se usarán para permitir al usuario realizar diferentes tipos de análisis; *getTableau()* dará las ecuaciones Tableau en forma matricial, *solution()* proporcionará la solución de las variables del circuito, *theveninEquivalent()*, y *nortonEquivalent()* los equivalentes Thevenin y Norton, respectivamente, *timeAnalysis()* el cambio de las variables en función del tiempo, y por último *dcAnalysis()* la representación de las variables en función de una fuente de tensión independiente. Los atributos determinarán el tipo de circuito. La lista de los métodos y atributos principales se observan en la Figura 3.4.

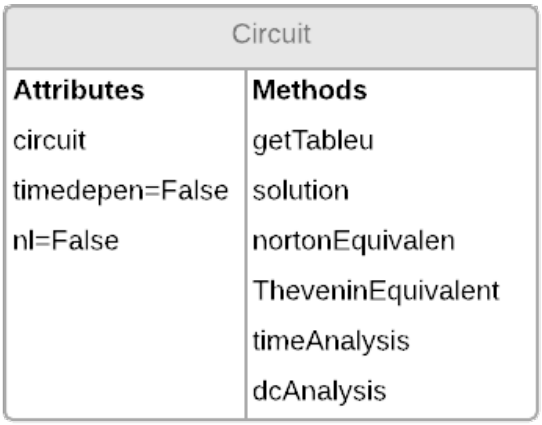


Figura 3.4: Atributos y métodos principales de la clase Circuit

Cuando se ejecute uno de los métodos principales, exceptuando $getTableau()$ se le pedirá al objeto de la clase $Circuit$ que obtenga una solución, y cuando eso ocurra, el flujo del programa será el que se muestra en la Figura 3.5.

El atributo *circuit* será una matriz: cada fila contendrá un elemento de la clase *Elements* y sus nodos, tal y como se puede apreciar en la expresión 3.2.

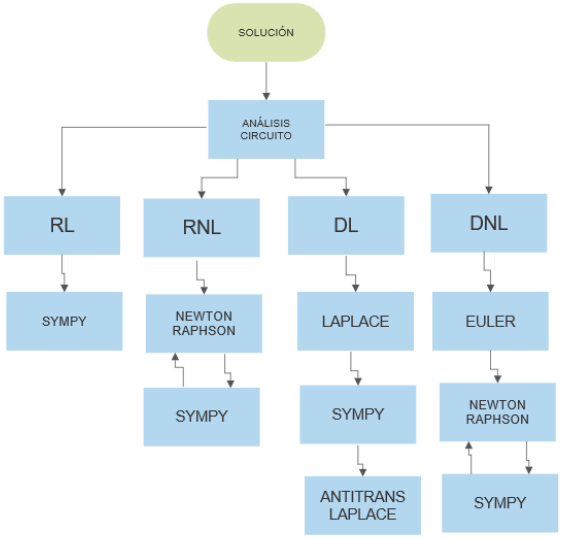


Figura 3.5: Representación gráfica del flujo interno del objeto *Circuit* cuando se le pide una solución

$$\begin{bmatrix} elemento_1 & [nodo_{11}^{+}, nodo_{11}^{-}] & \cdots \\ elemento_2 & [nodo_{21}^{+}, nodo_{21}^{-}] & \cdots \\ \cdots & \cdots & \cdots \\ elemento_n & [nodo_{n1}^{+}, nodo_{n1}^{-}] & \cdots \end{bmatrix} \tag{3.2}$$

donde $nodo_{ij}^{+,-}$ serán números enteros.

Un objeto de la clase *Circuit* necesitará de *circuit* para su instanciación, después,internamente, el objeto aplicará a cada uno de los elementos de *circuit* los métodos *getNL*, *getDif*, y *getTvariant* para determinar los valores reales de sus atributos *timedepen*, y *nl*, ya que de ese modo determinará el tipo de circuito que es.

Para el diseño interno de la clase *Circuit*, el problema se ha divido en tres partes dependiendo del tipo de circuito que se trata. Primero, se ha planteado la forma de resolver circuitos resistivos lineales, después dinámicos lineales, y por último no-lineales.

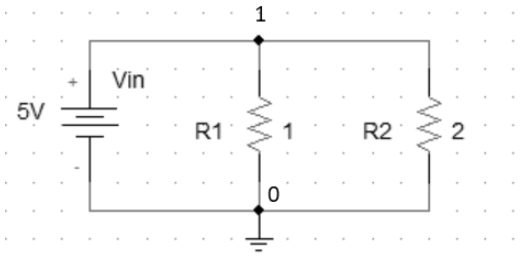
Finalmente, se creará una clase llamada *SubCircuit*[1] que representará un objeto compuesto por un conjunto de elementos de la clase *Elements*. Será necesario conseguir una ecuación equivalente del circuito, y así será posible la utilización de objetos complejos, como por ejemplo un OpAmp, en los circuitos.

Solucion de Circuitos Resistivos Lineales

Se construirá una función, privada de la clase *Circuit*, para la construcción de las matrices, A, B, y M N y $\boldsymbol{u}$ de la expresión (2.9).

[1]Los subcircuitos se comportan igual que los elementos, por lo tanto heredaran de la clase *Elements*.

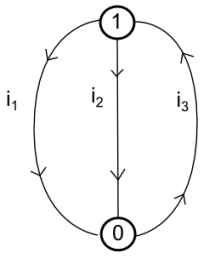
Para calcular A, es necesario conocer las direcciones de las ramas del digrafo. No se proporciona el digrafo, y por lo tanto es necesario crear un criterio para determinar las direcciones a partir de la representación, en forma matricial, del circuito , esto es: dado el $elemento_i$ y el puerto j de cualquier circuito, la rama transitara desde el $nodo_{ij}^{+}$ al $nodo_{ij}^{-}$. Ejemplo de esto último está representado en la Figura 3.6. Siguiendo el criterio expuesto, y cumpliendo con (2.3) se obtendría la matriz A. Para obtener B unicamente habría que calcular la transpuesta de A.



(a)

```
#Vin:  objeto de la clase VoltageSource
# con un valor de 5V.
Vin=VoltageSource(5)

#R1: objeto de la clase Resistence
#con un valor de 1 ohm.
R1=Resistance(1)

#R2: objeto de la clase Resistance
#con un valor de 2 ohm.
R2=Resistance(2)

#circuito: representacion del circuito
#en forma textual.
circuito=[[Vin, [1, 0]],
          [R1, [1, 0]],
          [R2, [0, 1]]]

#c: objeto de la clase Circuit.
c=Circuit(circuito)
```

(b)



(c)

Figura 3.6: La imagen (a) representa el circuito, en su forma gráfica, que se desea construir en Python. (b) muestra cómo sería la estructura externa para realizar dicha representación en forma textual y con objetos. Por ultimo (c) es una demostración del digrafo del circuito usando el criterio planteado.

Dentro del objeto de la clase *Circuit* se aplicará a cada elemento de *circuit* el método *getTableau*() para obtener las ecuaciones constitutivas de todos los elementos, y así poder calculas matrices M y N, y el vector $\boldsymbol{u}$. Con todo ello, se elaborará la matriz Tableau y se calculará la solución del sistema lineal mediante la función *solve_linear_system_LU()* de SymPy.

Solucion de Circuitos Dinamicos y Lineales

Para comenzar con la solución de circuitos dinámicos, existen dos posibilidades. Por un lado hacer que las ecuaciones de los elementos estén definidos en el espacio temporal, y luego aplicarles la transformada de Laplace mediante la función *laplace_transform()* de SymPy, o por otro lado definir las ecuaciones constitutivas en el espacio s. Esta última opción es la más favorable dado que las ecuaciones de los elementos con dependencia explicita de t, no son complejas de transformar manualmente, y de esa manera se estaría ahorrando el paso de aplicar *laplace_transform()*. Por lo tanto, las ecuaciones de los elementos como las fuentes independientes de voltaje/corriente sinusoidales, condensadores, y bobinas, estarán definidos en el programa de la siguiente forma:

$$\begin{array}{c} [[1],[0],[A*(2\pi f)/(s^2+(2\pi f)^2)]], \quad \textit{Fuente de tension sinusoidal} \\ [[0],[1],[A*(2\pi f)/(s^2+(2\pi f)^2)]], \quad \textit{Fuente de corriente sinusoidal} \\ [[-Cs],[1],[-C*v(0)]], \quad \textit{condensador} \\ [[1],[-Ls],[-L*i(0)]], \quad \textit{bobina} \end{array} \tag{3.3}$$

donde A y f son la amplitud y frecuencia, respectivamente, de las fuentes. C y L la capacitancia e inductancia del condensador y el inductor, respectivamente. Estos valores serán los inputs de instanciacion de sus respectivos objetos. El valor predefinido de $i(0)$ como el de $v(0)$ será igual a cero, sin embargo el usuario a la hora de instanciar los elementos dinámicos tendrá la opción de redefinirlos.

Una vez definidas las ecuaciones de los elementos, el procedimiento a seguir sería el mismo que en el de los casos RL, ya que *solve_linear_system_LU()* soporta la soluciona de sistemas lineales con expresiones simbólicas. Sin embargo, habría que incluir un paso adicional: aplicar la transformada inversa de Laplace a la solución obtenida, para lograrla en función del tiempo. Cuando el usuario llame a la método *timeAnalysis*, los el valor de la solución se evaluará en función del tiempo mediante *evalf()* de Sympy.

Solucion de Circuitos No-lineales

Se mantendrá la estructura del atributo *equation* en los casos de elementos no lineales. Sin embargo, los términos no-lineales de las variables internas, expresadas mediante símbolos, se ubicarán dentro de $[u_i]$ de la matriz (2.1). Para el caso del diodo, teniendo en cuenta la expresión (2.13) su *equation* tendrá la siguiente estructura,

$$\left[[0],[1],\left[I_0 * e^{(v_0/v_t)}\right]\right] \quad : v_t = 8{,}61 * 10^{-5} * t \tag{3.4}$$

el valor de I_0 depende del tipo de diodo, y v_0 es la tensión de la puerta. Para determinar *equation* del transistor NPN se ha cogido la ecuación (2.15),

$$\begin{array}{c} \left[[0,0],[1,0],\left[\alpha_F I_{Es}(e^{(v_1/v_t)}-1)-I_{Cs}(e^{(v_0/v_t)}-1)\right]\right] \\ \left[[0,0],[0,1],\left[-\alpha_R I_{Cs}(exp^{(v_0/v_t)}-1)+I_{Es}(e^{(v_1/v_t)}-1)\right]\right]. \end{array} \tag{3.5}$$

Y para el caso del transistor PNP se ha cogido la ecuación (2.16)

$$\begin{array}{c} \left[[0,0],[1,0],\left[\alpha_F I_{Es}(e^{(-v_1/v_t)}-1)-I_{Cs}(e^{(-v_0/v_t)}-1)\right]\right] \\ \left[[0,0],[0,1],\left[-\alpha_R I_{Cs}(e^{(-v_0/v_t)}-1)+I_{Es}(e^{(-v_1/v_t)}-1)\right]\right], \end{array} \tag{3.6}$$

donde, en ambos casos $v_t = 8{,}61 * 10^{-5} * t$, y los valores de α_F, α_R, I_{Es}, y I_{Cs} dependen del tipo de transistor[2]. Por otro lado, $v_0 \equiv v_{BC}$, y $v_1 \equiv v_{BE}$ son las tensiones del primer puerto, y el segundo, respectivamente.

[2]Sus valores se han obtenido de PSpice.

Cuando el circuito es no-lineal las ecuaciones KL de corrientes y voltajes se calcularán de la misma forma que en los anteriores casos. Sin embargo, para calcular el vector **u** es necesario tener en cuenta los elementos no-lineales, esto es; las variables internas del elemento que aparecen en $[u_i]$, deben ser sustituidas por las correspondientes variables del circuito. Terminado el proceso de sustitución, las ecuaciones se podrán expresar con la forma de (2.9).

Se añadirá a los objetos de la clase Circuit una función que aplique el método de Newton-Raphson para la solución de un sistema de ecuaciones. Para ello, se escogerá como punto inicial un vector con valores cercanos a cero. No se conocen las derivadas de **f**, y por ende a la hora de calcular $J_{\mathbf{f}}(\mathbf{i}, \mathbf{e}, \mathbf{v})$, es necesario aplicar el método de Euler progresivo. Para lograr $\mathbf{i}_{i+1}, \mathbf{e}_{i+1}, \mathbf{v}_{i+1}$ se volverá a usar *solve_linear_system_LU()*. El cálculo de $J_{\mathbf{f}}(\mathbf{i}, \mathbf{e}, \mathbf{v})$ se repetirá hasta que la norma de **f** sea pequeña, si no se consigue ese objetivo después de un numero determinado de iteraciones, se provocará un error.

Para realizar un análisis temporal de circuitos no-lineales, no se pueden usar ecuaciones en el espacio s. Por lo tanto, a los elementos dependientes de t se les definirá dos ecuaciones, una estará representada en el espacio del tiempo, y la otra en el s. Además, a las ecuaciones, representadas en t, con alguna derivada se les aplicará el método de Euler progresivo, véase expresión 3.7. En los casos de circuitos no-lineales, para calcular la Tableau los elementos ofrecerán al circuito sus ecuaciones dependientes del tiempo. Al realizar el análisis temporal, y conseguir una solución para un tiempo t_i, a los elementos con derivadas en sus ecuaciones se les proporcionará la solución de sus variables internas, para que se realice Euler debidamente. Por otro lado, el objeto de la clase *Circuit* guardará la solución obtenida en t_i, para usarlo como punto inicial en el método de Newton-Raphson para t_{i+1}.

$$\begin{array}{ll} [[1], [-h/C], [v_i]], & condensador \\ [[-h/L], [1], [i_i]], & bobina \end{array} \tag{3.7}$$

donde h es el tamaño del paso (impuesto por el usuario y constante a lo largo de toda la ejecucion de la funcion), y v_i/i_i valores del paso anterior.

3.3. Problemas Encontrados y su Solución

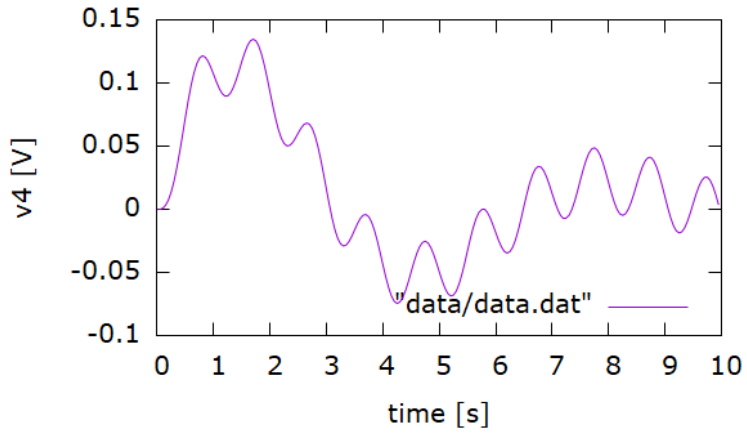
Los elementos V/I controlados por una rama se intentaron implementar como si de elementos de una puerta se tratara, esto hacia que las funciones para la creación de las ecuaciones Tableau fueran mas complejas. Finalmente, se optó por implementarlos como elementos bipuerta normales. Se aportó fiabilidad al diseño después de realizar numerosas pruebas con resultados correctos.

Después de realizar los pasos explicados en el apartado *Solución de Circuitos Dinámicos y Lineales*, se comenzó a testear la funcionalidad de *timeAnalysis()*. Para ello se analizó la tensión del condensador de un circuito RLC. En la primera simulación se ejecutó el código que se muestra en la Figura 3.7, donde en las primeras cuatro lineas se instancian los objetos, C, L, R, y Vin. Después se define *circuito* para representar un circuito RLC en forma matricial, siguiendo la estructura (3.2). Una vez instanciado x con *circuito* se llama a su método público *timeAnalysis()* para obtener la caída de tensión en C, en un intervalo $[0, 10]$. Los resultados se

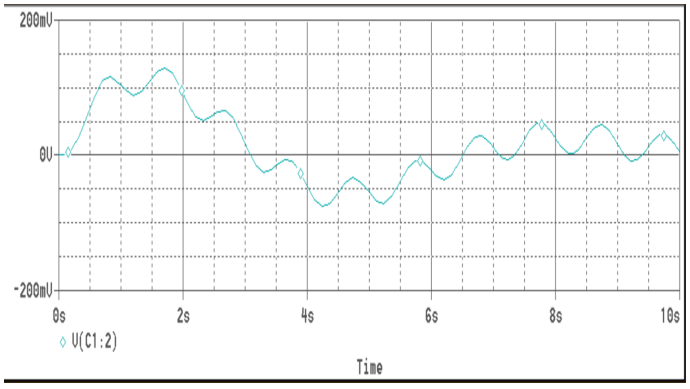
compararon con los de Pspice (vease Figura 3.8), y en ambos casos las respuestas concordaban. El tiempo de ejecución necesitado fue de 18 segundos aproximadamente.

```
C=Capacitor(1)
L=Inductor(1)
R=Resistance(0.5)
Vin=ACVoltageSource(1, 1)

circuito=[[Vin, [1, 0]],
          [R, [1, 2]],
          [L, [2, 3]],
          [C, [3, 0]]]


x=Circuit(circuito)
x.timeAnalysis([C, [3, 0]], 0, "voltage", 0, 10, 200)
```

Figura 3.7: Representación del código para para analizar un circuito RLC .



(a) Solucion obtenida del código de la Figura 3.7



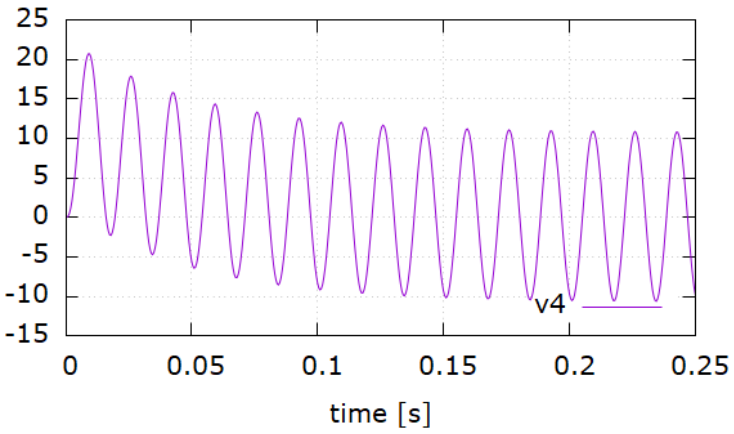
(b) Solucion obtenida de Pspice.

Figura 3.8: Simulación de la caída de tensión del condensador de un circuito RLC en función del tiempo. Los valores de los elementos son $C = 1f$, $L = 1H$, $R = 0{,}5\Omega$, y $A = 1V$ y $f = 1Hz$

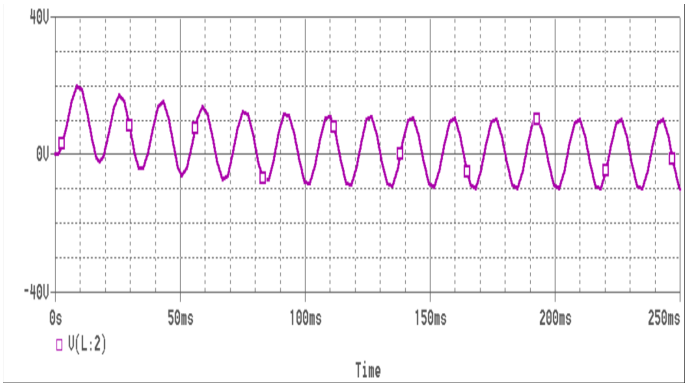
Sin embargo, la función *inverse_ laplace_ transform()* dejaba de funcionar debidamente para ciertas combinaciones de los valores de los objetos, como por ejemplo, $L = 100mH$,

$C = 500\mu f$, $R = 100\Omega$, $A = 220V$, y $f = 60Hz$. Por lo tanto, se decidió cambiar el esquema del código; las ecuaciones de los elementos con una dependencia explicita del tiempo dejarán de estar representados en el espacio de s, y se aplicará Euler regresivo a las ecuaciones diferenciales, como se observa en 3.7. De esta forma se prescindirá del métodos externos, *inverse_laplace_transform()*, del que se desconocen sus limitaciones.

Al implementar la nueva estructura, los resultados obtenidos de diferentes circuitos dinámicos concordaban con los de Pspice. Además, se observó una mejora en cuanto a los tiempos de ejecución. Por ejemplo, al realizar el mismo análisis que se plantea en la Figura 3.7, el tiempo necesitado fue de 9.87 segundos. En la Figura 3.9 se observa el resultado que anteriormente, utilizando *inverse_laplace_transform()*, no se podía conseguir, y su tiempo de ejecución fue de unos 10s aproximadamente.



(a) Solucion obtenida al implementar el método de Euler



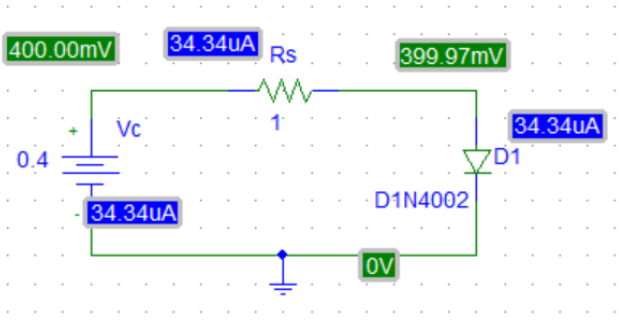
(b) Solucion obtenida de Pspice.

Figura 3.9: Simulación de la caída de tensión de un condensador del circuito RLC, en funcion del tiempo. Los valores de los elementos son $L = 100mH$, $C = 500\mu f$, $R = 100\Omega$, $A = 220V$, y $f = 60Hz$.

Al terminar de construir la parte del código que se encargaba de la solución de un sistema de ecuaciones no-lineales, se comprobó su funcionamiento con el código que se muestra en la Figura 3.10 (b). El resultado concordaba aproximadamente[3] con el de Pspice. El número de

[3]Cuando se usan elementos como diodos, y transistores, es probable que los resultados no concuerden del

iteraciones fueron 34 y el tiempo de ejecucion 7.48s.



(a) Solucion obtenida mediante el programa Pspice

```
Vc=VoltageSource(0.4)
Rs=Resistance(1)
D1=D1N4002()
circuito=[[Vc, [1, 0]],
          [Rs, [1, 2]],
          [D1, [2, 0]]]

x=Circuit(circuito)
x.solution()

e1: 0.400000000000000, e2: 0.372199833770999,
v1: 0.400000000000000, v2: 0.0278001662290014,
v3: 0.372199833770999, i1: -0.0278001662290014,
i2: 0.0278001662290014, i3: 0.0278001662290014
```

(b) Solucion obtenida utilizando el código contruido

Figura 3.10: Se muestra la representación y solución del mismo circuito de dos formas distintas; (a) se ha obtenido utilizando la herramienta PSpice, y (b) mediante el programa construido.

Con el fin de reducir el número de iteraciones, se implementó el metodo de Newton amortiguado. Para determinar un valor eficaz de w, véase ecuación 2.27, se tubo en cuenta su significado, el cual se puede observar en la Figura 3.11. Dependiendo de con que valor de $w_{1,2,3}$ se consiga el objetivo de minimizar $\|\mathbf{f}\|$, se optaría por diferentes intervalos, esto es:

- Si con w_1 se consigue un valor menor, el intervalo será $[0{,}1; 0{,}9]$.
- Si con w_3 se consigue un valor menor, el intervalo será $[1{,}1; 10]$.
- Si con w_2 se consigue un valor menor, no se determinara ningún intervalo, y por ende se dará por valido el valor de $(\mathbf{i}, \mathbf{e}, \mathbf{v})$ obtenido.

En el primer y segundo caso se dividiría el intervalo a la mitad y se seleccionaría el subintervalo que tuviera un valor de w que mejorara el resultado de $\|\mathbf{f}\|$. El final del proceso de división, y por ende de la busqueda de w, terminaría después de un número de iteraciones, como por ejemplo 10. Al implementar esto último, y ejecutar, el código de la Figura3.10, se observaron algunas mejorías: el número de iteraciones se redujo a 4 y el tiempo de ejecución a 3.23s.

Sin embargo, al estudiar otros casos, se contemplo que la convergencia hacia la precisión deseada (10^{-7}) no estaba siempre garantizada. Además, al aplicar la función de *timeAnalysis()* sobre un circuito no-lineal, el tiempo necesario era muy grande. Por ejemplo, en el caso que se presenta en la Figura 3.12, el tiempo de ejecución fue de 221.44s, y al cambiar la amplitud de la fuente de tensión sinusoidal de 0.4V a 1V el programa no llegaba a una solución. Es por eso, que se decidió realizar un cambio en la forma de atacar el problema: se aplicaría el método de Newton-Raphson a las ecuaciones de los elementos no lineales. De esa manera se evitaría el calculo de la matriz $J_{\mathbf{f}}$.

Para implementar el método de Newton-Raphson desde un enfoque mas individual, los elementos no-lineales tendrán que tener linealizadas sus ecuaciones sobre un punto[4] antes de

todo, ya que los parámetros de las ecuaciones no seran exactamente las mismas.

[4]En el momento en el que un elemento no-lineal se instancia, tendrá por defecto un punto inicial, cuyo valor dependerá del tipo de elemento.

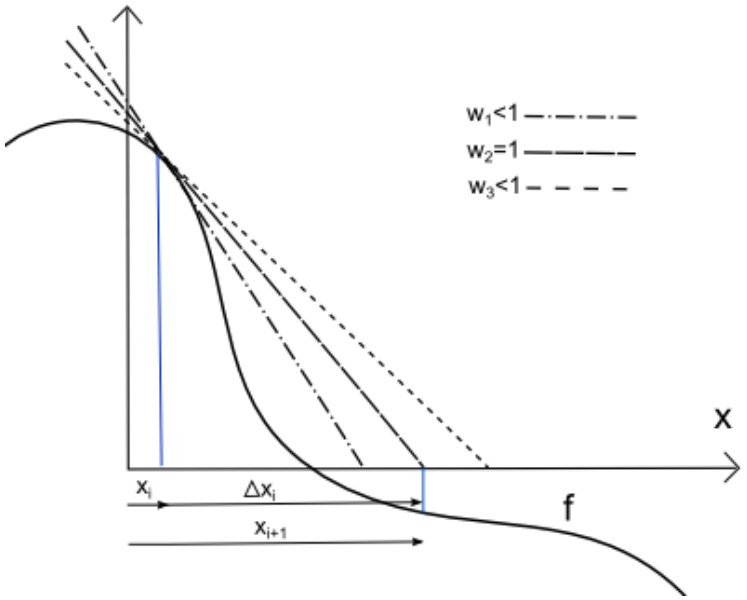
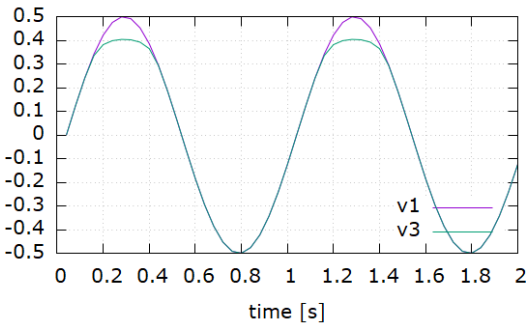


Figura 3.11: Representación gráfica del significado del valor de w, en el caso de una función con dependencia de una variable.

```
AV=Element.ACVoltageSource(0.5, 1)
Rs=Element.Resistance(1)
D1=Element.D1N4002()
circuito=[[AV, [1, 0]],
          [Rs, [1, 2]],
          [D1, [2, 0]]]

x=Circuit(circuito)
x.timeAnalysis([[AV, [1, 0]], [D1, [2, 0]]],
               [0, 0], ["voltage", "voltage"],
               0, 2, 50)
```

(a) Código ejecutado.



(b) Resultado de la ejecución.

Figura 3.12: En la figura (a) se muestra la representación de un circuito compuesto por una fuente de tensión, una resistencia, y un diodo. Mas adelante se realiza la llamada a la función *timeAnalysis()*, para mostrar el gráfico (b).

que el objeto de la clase *Circuit* cree las matrices M, N, y el vector $\mathbf{u}$. Después, aplicando la función *linalg.solve()* de Numpy[5], se obtendrá la solución del sistema, y cuando se cumpla el criterio de precisión se dará por valida la solución obtenida[6]. En caso contrario, a los elementos no-lineales se les dará la solución de las corrientes y/o voltajes de sus puertas, para linealizar, nuevamente, sus ecuaciones sobre dichos valores. El proceso se repetirá hasta que el criterio de precisión se cumpla.

A la hora de programar lo planteado, se optó por realizar la linealización a mano, de este

[5]Se comenzó a usar *linalg.solve()* de Numpy, en vez de *solve_linear_system_LU()*, cuando se decidió no resolver sistemas lineales con expresiones simbolicas

[6]Cuidado, dentro de **f** estarán definidas las ecuaciones reales de los elementos lineales, y no-lineales. Si **f** estuviera definido con las ecuaciones lienalizadas, el criterio de precisión no tendría validez.

modo solo se tendrían que sustituir los valores iniciales, o los dados por el objeto de la clase *Circuit*. Para obtener la expresión de *equation* para el caso de elementos linealizados se cogieron las ecuaciónes linealizadas y se implementaron acorde a la estructura definida en (3.1). En el caso del diodo se cogió la ecuación (2.14), y por lo tanto *equation* se definió de la siguiente forma:

$$\left[\left[-\frac{I_0}{v_t}e^{v_0/v_t}\right],[1],\left[I_0(e^{v_0/v_t}-1)-\frac{I_0v_0}{v_t}e^{v0/vt}\right]\right] \quad : v_t = 8{,}61 * 10^{-}5t. \tag{3.8}$$

v_0 es el valor sobre el que se linealiza la ecuación (2.13). Por otro lado, para el transitor NPN se cogió la ecuación (2.17),

$$\begin{array}{c}[[G_2,-\alpha_F G_1],[-1,0],[\alpha_F I_1 - I_2]]\\ [[-\alpha_R G_2, G1],[0,-1],[\alpha_R I_2 - I_1]],\end{array} \tag{3.9}$$

En el caso del transistor PNP se cogió la ecuación (2.17), y por lo tanto:

$$\begin{array}{c}[[-G_2,\alpha_F G_1],[1,0],[\alpha_F I_1 - I_2]]\\ [[\alpha_R G_2, -G1],[0,1],[\alpha_R I_2 - I_1]],\end{array} \tag{3.10}$$

Una vez que se definieron dentro de los elementos no-lineales sus ecuaciones linealizadas, se prosiguió con el diseño de la función privada *NewtonRaphson1()* de la clase *Circuit*, para la implementación del algoritmo de Newton-Raphson. Una vez terminado, se probo su eficiencia con los circuitos en los que el método de Newton amortiguado había fallado, y no surgieron problemas. Por otro lado, los tiempos de ejecución mejoraron, por ejemplo al implementar el circuito que se muestra en la Figura 3.12(a) el programa solo necesito 8.04s.

Teniendo en cuenta la forma en la que se debe escribir un circuito en este programa, cuando se trata de un circuito grande, su construcción puede suponer una tarea tediosa para el usuario. Con el fin de minimizar este problema se planteo crear una clase llamada *SubCircuit*. Un objeto del tipo *SubCircuit* se comportará como uno de la clase *Element*, pero en su interior tendrá definido un circuito interno. Dentro de este objeto se tendrá que indicar cuales serán sus entradas y/o salidas. Después, los métodos para obtener la ecuación que represente el objeto *SubCircuit* variarán dependiendo del circuito interno que tenga definido.

Si el circuito es resistivo y lineal se querrá obtener una ecuación con la siguiente forma:

$$\begin{bmatrix} [1,0,\cdots,0] & [b_{11},b_{12},\cdots,b_{1n}] & [u_1] \\ [0,1,\cdots,0] & [b_{21},b_{22},\cdots,b_{2n}] & [u_2] \\ \cdots & \cdots & \cdots \\ [0,0,\cdots,1] & [b_{n1},b_{n2},\cdots,b_{nn}] & [u_n] \end{bmatrix}, \tag{3.11}$$

Una opción para lograr la ecuación (3.11), es aplicar el criterio de Thevenin. Suponiendo que se quiera obtener el valor de u_j, se debe poner una resistencia en el puerto j, y en los demás puertos fuentes de corriente independientes con valor nulo. Después se calculará la tensión v_j^1 de la resistencia; su valor será igual al de u_j. De querer obtener el valor del término b_{jk}, bastaría con

elevar el valor de la fuente de corriente independiente del puerto k a $1A$, y calcular nuevamente la tensión de la resistencia, v_j^2. Por lo tanto, el valor de b_{jk} será el siguiente:

$$b_{jk} = u_j - v_j^2 = v_j^1 - v_j^2 \tag{3.12}$$

Sin embargo, este método no siempre se puede aplicar, como por ejemplo cuando hay alguna fuente de corriente independiente colocada en serie a la puerta. Como alternativa al criterio de Thevenin en las entradas se colocarán resistencias y se obtendrán sus soluciones de voltaje y corriente, n+1 veces, donde n es el numero de entradas. Cada vez que se resuelva el circuito los valores de las resistencias se cambiarán (el valor es indiferente, sin embargo no debe repetir la misma combinación). Una vez hecho esto, para cada valor de $i \in n$ se resolverá el siguiente sistema:

$$\begin{pmatrix} i_1^{(0)} - i_1^{(1)} & i_2^{(0)} - i_2^{(1)} & \cdots \\ i_1^{(0)} - i_1^{(2)} & i_2^{(0)} - i_2^{(2)} & \cdots \\ \cdots & \cdots & \cdots \\ i_1^{(0)} - i_1^{(n)} & i_2^{(0)} - i_2^{(n)} & \cdots \end{pmatrix} \begin{pmatrix} b_{i1} \\ b_{i2} \\ : \\ b_{in} \end{pmatrix} = \begin{pmatrix} v_i^{(0)} - v_i^{(1)} \\ v_i^{(0)} - v_i^{(2)} \\ : \\ v_i^{(0)} - v_i^{(n)} \end{pmatrix}, \tag{3.13}$$

donde la j e i de $v_i^{(j)}$, indican la vez en la que se ha resuelto, y la puerta, respectivamente. Una vez obtenido b_i, se podrá conseguir u_i:

$$u_i = v^{(j)} + b_{i1} * i_1^{(j)} + b_{i2} * i_2^{(j)} + \cdots, \quad \forall j \in (n+1) \tag{3.14}$$

Si el circuito tiene elementos dinámicos como condensadores o inductores, se aplicara el procedimiento de Euler sobre sus ecuaciones, y de esa manera se podrá lograr la expresión (3.11), usando los métodos descritos. Finalmente, si el circuito interno del objeto de la clase *SubCircuit* es no-lineal se colocarán en las entradas fuentes independientes de voltaje[7] para lograr la siguiente ecuación:

$$\begin{bmatrix} [a_{11}, a_{12}, \cdots, a_{1n}] & [1, 0, \cdots, 0] & [u_1] \\ [a_{21}, a_{22}, \cdots, a_{2n}] & [0, 1, \cdots, 0] & [u_2] \\ \cdots & \cdots & \cdots \\ [a_{n1}, a_{n2}, \cdots, a_{nn}] & [0, 0, \cdots, 1] & [u_n] \end{bmatrix}, \tag{3.15}$$

A las fuentes de voltaje de las entradas, se les impondrá unos valores iniciales. Con esos valores se obtendrá la solución del circuito definido dentro del objeto de la clase *SubCircuit*, y estando linealizado en las entradas se aplicaran diferentes voltajes para obtener (3.11).

Cuando un objeto de la clase *Circuit* tiene definido en su *circuit* un objeto del tipo *SubCircuit* no-lineal, al aplicar Newton-Raphson y obtener una solución, al *SubCircuit* se le proporcionará la solución de las tensiones de sus puertas, para linealizar su circuito interno sobre esos valores. Sin embargo en estos casos no es valido usar un criterio de precisión (debe

[7]En estos casos se considerara que en las entradas no hay fuentes independientes de voltaje(corriente) colocadas en paralelo(serie).

cumplirse $\|\mathbf{f}\| < \epsilon$ para que la solución se de por correcta), ya que no se posee la ecuación real"del objeto *SubCircuit*. Por lo tanto, en esta clase de casos se establecerá un criterio de convergencia, esto es: la solución se dará por valida si $\|\mathbf{f}_{i+1} - \mathbf{f}_i\| < \epsilon$.

Al crear la clase *SubCircuit*, y las clases que heredan de ella (de forma análoga ocurre con la clase *Elements*, y las clases *Resistance*, *Inductor*,...), se realizaron algunos ejemplos simples, y después de comprobar que funcionaban correctamente se crearon clases que implementaban circuitos mas complejos como el de un OpAmp. Se decidió comenzar con la implementación del OpAmp que ofrece Pspice, dado que es un modelo reducido (véase el apéndice A): presenta 5 diodos, 2 transistores, 2 condensadores, y 24 elementos lineales, incluyendo las fuentes independientes. Al implementarlo, el programa lanzaba errores como el de $SingularMatrix$, y después de analizar dicho modelo paso por paso, no se llego a comprender la raiz del fallo. Por lo tanto, se cambio al modelo que se presenta en [13]. Esté está compuesto por 8 transistores, 2 diodos, un condensador, y 6 elementos lineales, incluyendo las fuentes independientes, y al implementarlo la solución divergía: las fuentes Vcc^+ y Vcc^- al tener valores de $10V \sim 15V$ y $-10V \sim -15V$ respectivamente, en las primeras iteraciones las soluciones de puerta de los transistores eran valores cercanos a $|10V| \sim |15V|$, y debido al carácter exponencial de sus ecuaciones, se obtenían una ecuaciones linealizadas con valores muy altos, que mas tarde provocarían la divergencia del sistema entero. Para resolver este problema se plantearon dos opciones:

- Cuando los valores de linealización de los elementos no lineales, como el diodo o el transistor, provoquen unos valores altos de sus exponetes, las ecuaciones se sustituirán por sus equivalentes logarítmicos. Por ejemplo en el caso del diodo:

$$\begin{gathered}\left[\left[-\frac{I_0}{v_t}e^{v_0/v_t}\right],[1],\left[I_0(e^{v_0/v_t}-1)-\frac{I_0 v_0}{v_t}e^{v0/vt}\right]\right] \\ \Downarrow \\ \left[[1],\left[-\frac{v_t}{i_0+I_0}\right],\left[v_t\ln\left(\frac{i_0}{I_0}+1\right)-\frac{v_t}{i_0+I_0}i_0\right]\right]\end{gathered} \tag{3.16}$$

  donde, i_0 es la corriente de la puerta.

- Dado que el gran problema proviene de las fuentes independientes con valores altos, se realizará una y otra vez la división de sus valores, hasta que sea posible encontrar una solución. Cuando se encuentre, los elementos no lineales guardarán las soluciones de corriente/voltaje de sus puertas, para que su polarización esté más cerca de la real. Las fuentes independientes volverán a sus valores normales, y si nuevamente no es posible obtener la solución, se volvería a repetir el procedimiento anterior. Poco a poco, se conseguiría llegar a la solución correcta.

Con la primera opción, la solución no llegaba a converger. Sin embargo, al implementar el segundo caso se consiguió que el circuito interno del objeto de la clase *SubCircuit* convergiera hacia una solución, pero al aplicar Newton-Raphson sobre el circuito en el que estaba implementado el OpAamp, la solución oscilaba.

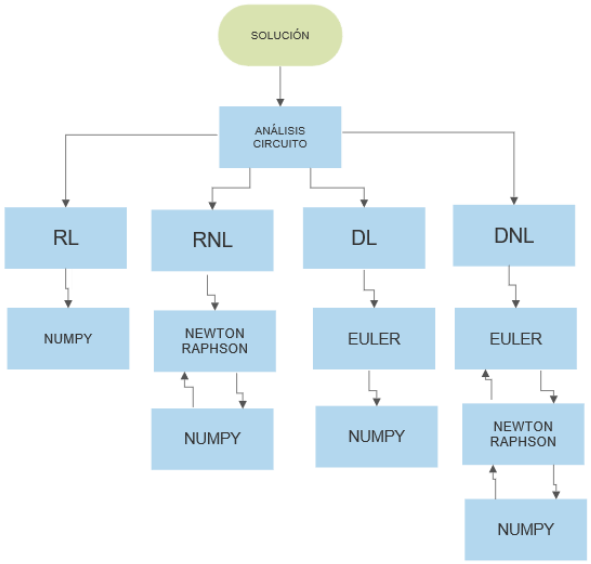


Figura 3.13: Flujo del programa del diseño final.

3.4. Diseño final

En el diseño final se han incluido una gran variedad de clases que heredan de *Elements*. Las clases que representan los elementos resistivos, son *Resistance*, $VoltageSource$, $CurrentSource$ entre otros. Las clases que representan elementos no lineales como $D1N4002$, o $Q2N2222$, tienen variables de puerta no lineales, y estas están expresadas de forma simbólica en *equation*. Los elementos no-lineales no son los únicos que tienen una notación simbólica en sus ecuaciones, dado que se aplica Euler regresivo directamente sobre las ecuaciones de los elementos dinámicos, *equation* tendrá una dependencia del símbolo h, que será igual al tamaño del paso. Por otro lado, los elementos con una dependencia explicita del tiempo, necesitarán del símbolo t para luego poder evaluarlas debidamente.

Cuando a la clase *Circuit* se le pide la solución de su atributo *circuit*, el flujo del programa será el que se muestra en la Figura 3.13, algunos métodos de *Circuit* solo pueden ir por algunas ramas que se muestran en la Figura 3.13, como se verá más adelante. La función *solution*() de la clase *Circuit* es solo valida cuando *circuit* está compuesta por elementos resistivos, y de no especificar ningún argumento de entrada, devolverá todas los valores de las variables del circuito. En cambio, si en los argumentos de entrada se determina la posición del elemento y el puerto del que se quiere obtener la información, junto con el tipo de variable que se desea, *voltage*, *current* o *node*, *solution*() lo proporcionará.

Cuando *solution*() se ejecuta, verifica si los elementos son lineales, o no. En el caso de que sean lineales, la representación matricial de Tableau se obtiene sin problemas, y la solución de las variables se consigue con *numpy.linalg.solve*(). En caso contrario, logra la expresión de las ecuaciones Tableau $\mathbf{f}$, necesario para el criterio de precisión que se aplicará más tarde, definidas en notación simbólica. Entonces, comenzará la aplicación del método de Newton-Raphson (2.11); las ecuaciones de los elementos no-lineales se linealizarán para conseguir una

solución que se aproxime a la real. El proceso se repetirá hasta que se cumpla el criterio de precisión, en el que se exige un valor menor que 10^{-10}. Por otro lado, se ha impuesto un número máximo de iteraciones, 50 para ser exactos. Si por el contrario, la solución provoca una valor de $\|\mathbf{f}|$ superior a 10^{20} el programa comenzará a dividir los valores de las fuentes independientes de *circuit* con el fin de que los elementos no-lineales se acerquen a su polarización real. Si después de 40 divisiones no se consigue una solución, el programa lanzara un *RuntimeError*.

solution() es la base de otras funciones como *dcAnalysis*(), y *theveninEquivalent*() esto es; en *dcAnalysis*() se obliga a una fuente de tensión, seleccionada por el usuario, a variar sus valores linealmente dentro de un rango dado por el usuario, y para cada valor de la fuente se aplicará *solution*(). *theveninEquivalent*() solo se aplicará para casos de circuitos resistivos lineales, y consistirá en insertar una fuente independiente de corriente entre un par de terminales, indicados por el usuario, y obtener la diferencia de tensión entre los dos terminales mediante *solution*(), para dos valores distintos de corriente. Des ese modo, se logrará la información suficiente para el cálculo del equivalente Thevenin. En el caso de *nortonEquivalent*(), se llama a *theveninEquivalent*(), y de sus datos se calcula el equivalente Norton.

la función *timeAnalysis*() exige meter como argumentos de entrada las posición del elemento y el puerto del que se quiere obtener el valor de voltaje o corriente, además de t_0, punto en el que se quiere comenzar el análisis temporal, t_N, punto final, y el número de pasos N. El tamaño del paso h se define de la siguiente forma:

$$h = \frac{t_1 - t_0}{N}, \tag{3.17}$$

y todos los puntos temporales serán equidistantes, esto es: para cualquier $i \in N - 1$, t_{i+1}, y t_i estarán separados por una distancia de h.

Cuando comience la ejecución de *timeAnalysis*() se buscarán los elementos dinámicos para sustituir el símbolo h de sus ecuaciones, por el valor obtenido de h usando los argumentos de entrada (3.4). Después, para un tiempo dado t_j, se evaluarán los elementos con dependencia explicita de t, y se llamará a la función *solution*() para obtener la solución en t_j. A los condensadores se les dará la tensión de sus puertas v_j, y a los inductores las corriente de sus puertas i_j para poder aplicar Euler. El proceso se repetirá hasta que $t_j = t_N$.

Una vez terminado, y analizado la funcionalidad del código, se ha realizo su documentación. A lo largo de todo el proceso se fue realizando una para el desarrollador, sin embargo era necesario crear una guía para usuarios; documento con la información necesaria para saber usar el programa. Dado que se trata de Python se decidido seguir sus estándares, esto es; después de definir una función se debe escribir un breve resumen de su contenido y funcionamiento. No basta con realizar dicha descripción, también es necesario detallar los inputs, outputs, los posibles errores, alguna nota de ser necesario, etc. En relación al estilo y la forma, se siguió la guía de documentación de numpydoc[4], un pequeño ejemplo de ello se puede observar en el siguiente código,

```
1 def theveninEquivalent(nodo1, nodo2):
2     """
```

```
    It calculates the Thevenin equivalent of the circuit.

    Parameters
    ----------
    node1: int
    It is equivalent to the positive node, node+

    node2: int
    It is equivalent to the negative node, node-

    """

    m = [0, 0]
    for j in range(2):
    cs = Element.CurrentSource(j)
    c = []
    ...
```

Por otro lado, cuando se trata de la documentación de una clase, en la sección Paramentes se definen las variables para instanciar el objeto. Para la descripción de los atributos, y métodos internos se utilizan las secciones *Attributes* y *Methods*, respectivamente. Existen otras secciones[8] como por ejemplo, *Raises*, *Notes*, *Examples*, etc. Una vez escrito toda la documentación del proyecto, se uso la herramienta Sphinx para obtener su representación en forma de HTML y LaTeX..

Para conocer mejor los atributos, y las funciones, de los objetos, se ha construido un diagrama UML usando la herramienta pyreverse, véase apéndice C.

[8]Para conocer la forma de escribirlas correctamente, véase la pagina oficial de numpydoc.

CAPÍTULO 4

Resultados

El procedimiento seguido fue crear funciones para un objetivo especifico, luego realizar pruebas, y si se contemplaba algún error, solucionarlo. Después, se escogía otro objetivo que cumplir, y se volvía a realizar el mismo procedimiento. Sin embargo, este proceso provoco un código poco esclarecedor, y que algunas funciones dejarán de funcionar debidamente. Cuando se cumplieron gran parte de los objetivos, se procedió a la limpieza del código, arreglar los pequeños problemas de las funciones, y fijar la estructura del proyecto entero. Solo queda probar la funcionalidad de *solution()*, *theveninEquivalent()*, *nortonEquivalent()*, *dcAnalysis()*, y *timeAnalysis()* para diferentes circuitos, y sus resultados se compararán con los de Pspice. Los circuitos usados para dicha finalidad, se han obtenido de algunos ejercicios/practicas de las asignaturas Circuitos Lineales y No Lineales, y Electronica Analogica. Por otro lado, los archivos que se han creado para realizar el análisis, más la representación gráfica de los circuitos que se resuelven se muestran en el apéndice B.

Para el circuito de la Figura 4.1, se pretende calcular su solución, y el equivalente Thevenin y Norton del nodo que esta marcado en rojo. Para ello se ha creado equivalente.py, y al ejecutarlo se observa la siguiente respuesta,

```
>>>{e1: 5.0, e2: 10.0, e3: 10.0, v1: 5.0, v2: 5.0, v3: 10.0, v4: -10.0, v5: 0.0, i1: 0.25, i2:
     0.25, i3: 1.0, i4: 1.25, i5: 0.25}

vth=10.0 V, rth=-5.0 ohm
inor=-2.0 A, rth=-5.0 ohm
```

A lo que las soluciones de las variables del cirucuito respecta, ambos casos coinciden. Sin embargo, los valores del equivalente Thevenin y Norton no se pueden comparar, ya que Pspice carece de dicha herramienta.

Despues, para observar la funcionalidad de *dcAnalysis()* se construyó en diode.py, un circuito compuesto por tres elementos, un diodo, una fuente de tensión independiente, y una

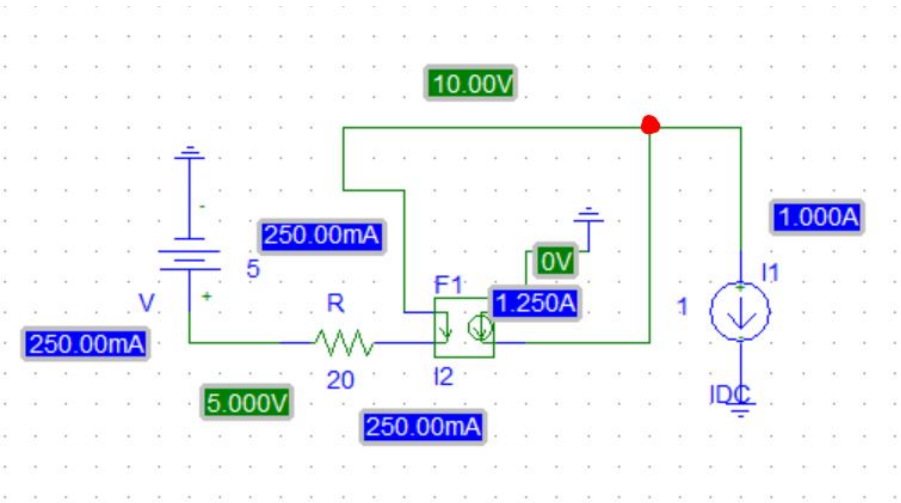


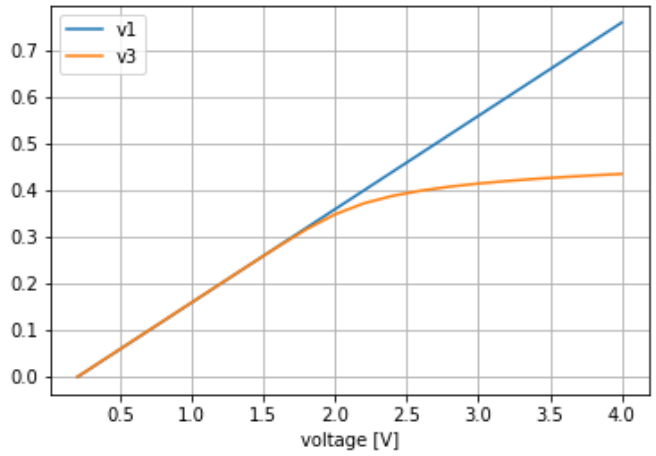
Figura 4.1: Se trata de un circuito simple, y lo componen una fuente de tension, una resistencia, una fuente de corriente independiente y otra controlada por una corriente.

resistencia, para así poder analizar el comportamiento del diodo a medida que la tensión aumentaba. El resultado se compara con el de Pspice en la Figura 4.2.

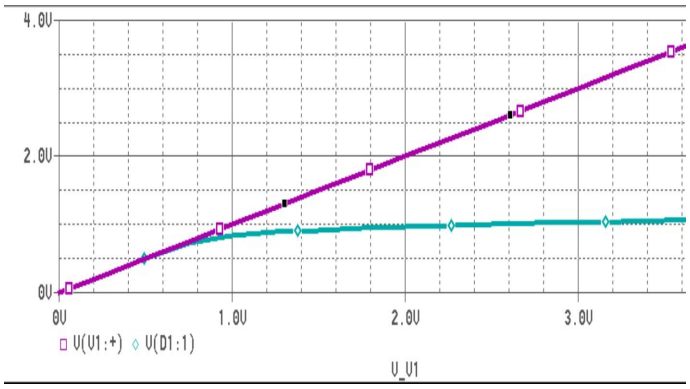
En ambos casos se observa la misma respuesta, y claramente se puede interpretar el funcionamiento del diodo analizando la gráfica; cuando la fuente de tensión se mantiene aproximadamente por debajo de los 0.4V, el diodo se mantiene como un circuito abierto, y cuando el valor de la tensión aumenta, el diodo deja de actuar como un circuito abierto y se crea una considerable corriente.

La herramienta mas importante y por lo tanto a las que mas pruebas se le someterá es *timeAnalysis()*, además se comprobarán los tiempos de ejecución para cada caso. Primero se usará un circuito no lineal y dinámico simple, al que comúnmente se le denomina rectificador de media onda con filtro de condensador, y al ejecutar el archivo rectificador.py con el código que se muestra en el apéndice B, se ha logrado la gráfica de la Figura 4.3. El tiempo de ejecución fue 142.48s. Aunque se trate de un circuito simple, se han necesitado una gran cantidad de pasos, 1200, en un intervalo de 0.25s para lograr un resultado correcto. Esto se debe a que a mayor cantidad de pasos, la solución de los elementos no-lineales se mantiene lo suficientemente cerca de la siguiente solución, por lo que a la hora de linealizarlos convergen con rapidez. En caso contrario, existe el peligro de que diverja. Pspice consiguió sin problemas el resultado, y claramente se ve que la Figura 4.3(a) y (b) muestran el mismo comportamiento.

Con los conocimientos sobre el funcionamiento de los elementos, y haciendo uso del gráfico 4.3, es posible interpretar la forma en la que actúan los componentes, esto es; a medida que transcurre el tiempo, el condensador se carga hasta llegar a $t \approx 0{,}4s$, primer máximo de la fuente de tensión sinusoidal. Después, al ser la carga del condensador superior a v_1, el diodo entra en modo inverso (no conduce), y comienza la descarga del condensador. La rapidez de la descarga estará determinada por el valor de la capacitancia. En este caso, con $C = 70 * 10^{-7} f$, se consigue una descarga rápida, y cuando $t > 0{,}16s$ la tensión de la fuente independiente superará la del condensador, y en consecuencia el diodo volverá a su modo activo, y el condensador se volverá a cargar.

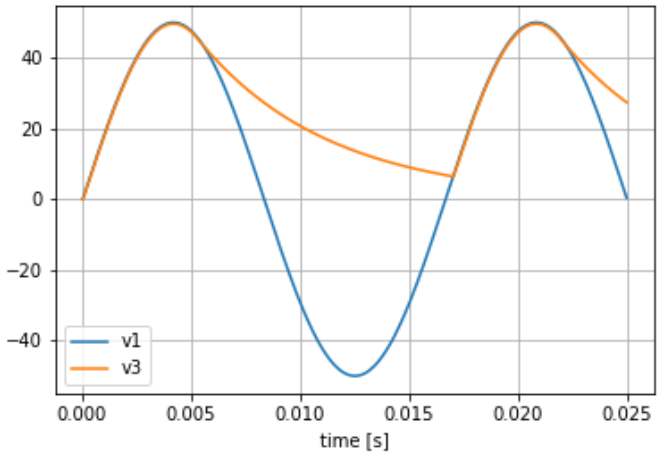


(a) Solución obtenida al ejecutar diodo.py.

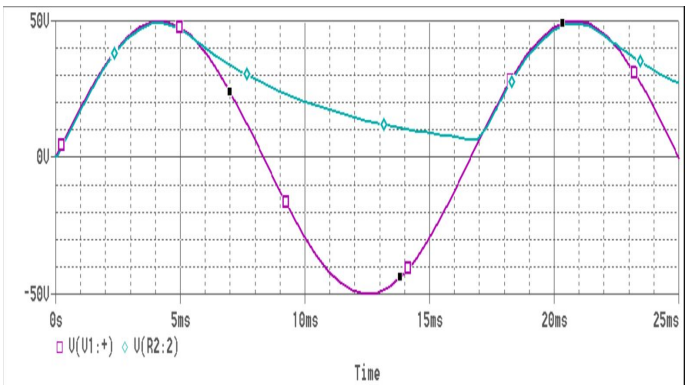


(b) Solución obtenida mediante el programa Pspice.

Figura 4.2: En ambos casos se simula un circuito compuesto por una fuente de tensión, una resistencia, y un diodo puestos en serie. Se muestra la tensión a través del diodo ((a) naranja, (b) rojo) a medida que el valor de la fuente de tensión aumenta linealmente ((a) azul, (b) verde).

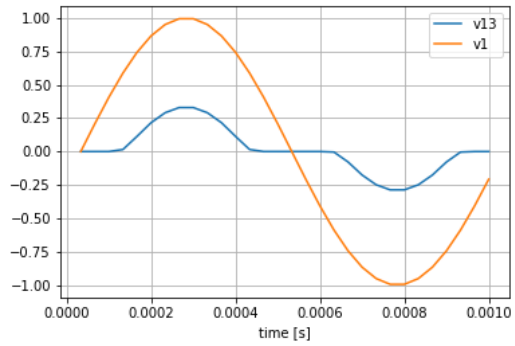


(a) Soluciónn obtenida al ejecutar rectificador.py. contruido

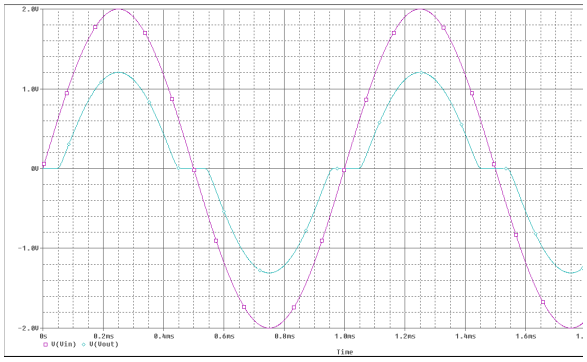


(b) Solución obtenida mediante el programa Pspice.

Figura 4.3: En ambos casos se simula un circuito compuesto por una fuente de tensión sinusoidal, una resistencia, un diodo, y un condensador. Tanto en (a) como en (b) se muestra la tensión a través de la resistencia ((a) naranja, (b) rojo), y la fuente de tensión ((a) azul, (b) verde) a medida que transcurre el tiempo.



(a) Solucion obtenida utilizando el código contruido



(b) Solucion obtenida mediante el programa Pspice

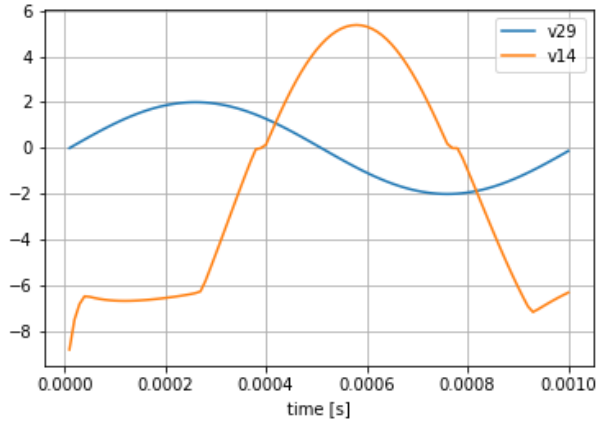
Figura 4.4: Se simula un circuito compuesto por una 3 fuentes de tensión, 4 resistencias, y 2 transistores, uno del tipo PNP, y el otro NPN. Tanto en (a) como en (b) se muestran la tensiones a través de la resistencia R1 ((a) azul, (b) rojo), y la fuente AV ((a) naranja, (b) verde) en función del tiempo.

Con el siguiente fichero, etapab.py, se simula una etapa de tipo B (segunda practica de la asignatura de Electronica Analogica). Aunque este caso parezca algo mas complejo que el anterior, dado que tiene dos elementos no lineales, no presenta problemas en relación a la selección de pasos, divergencia etc. Ademas el tiempo de ejecución ha sido relativamente corto, 50.41s. Al compararlo con los resultados de Pspice se observa el mismo resultado. Cuando el valor absoluto de la fuente sinusoidal, v_1, es menor que $0{,}4V$ aproximadamente, los transistores no conducen. Sin embargo, cuando $v_1 > 0{,}4V$ TR1 comienza a conducir mientras que TR2 sigue en modo cortocircuito. En caso contrario, $v_1 < -0{,}4V$, TR2 conduce, y TR1 se mantiene en cortocircuito.

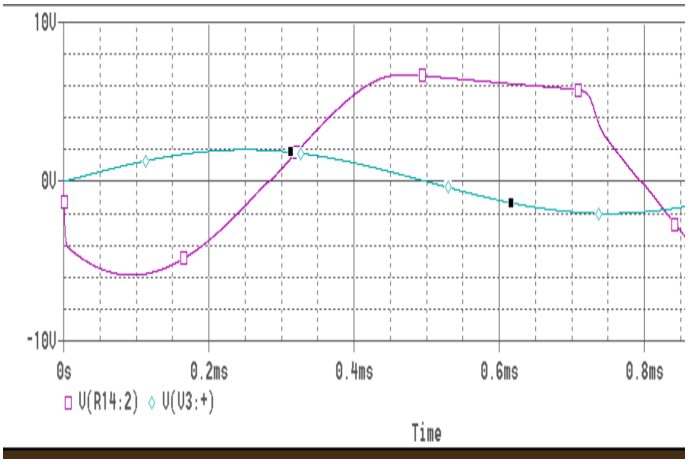
En etapa3.py se ha implementado algo mas completo, esto es: un amplificador por etapas (ejercicio de examen de la asignatura Electrónica Analógica), compuesto por 6 elementos no-lineales, y 14 lineales. El programa ha sido capaz de conseguir una solución, sin embargo no coincide con la de Pspice, como se puede observar la Figura 4.5. En ambos caso se puede observar el mismo desfase entre la señal de salida y la de entrada. Sin embargo, cuando $t \approx 0{,}5s$ (b) presenta una saturación, mientra que en el caso (a) no.

Por último, en opamp1.py se ha implementado la estructura de un OpAmp real y se ha configurado en modo inversor, es decir; la entrada positiva del OpAmp se ha conectado a tierra y en la entrada negativa hay una retroalimentación. Teniendo en cuenta que la frecuencia de la señal de entrada es de $1kH$, como punto inicial se ha escogido $t1 = 0s$, como punto final $tn = 0{,}01s$, y 100 el número de pasos a realizar. Para obtener el resultado de la Figura 4.6 (a) se han necesitado $3856{,}78s$.

Para dar fiabilidad al resultado e intentar conseguir más información, en opamp2.py en vez de usar la estructura de un OpAmp real, se ha insertado su diseño simplificado[13]. A diferencia de opamp1.py se ha optado por un tiempo final mayor, $tn = 0{,}03s$. Al realizar el análisis se ha observado la misma deriva, véase Figura 4.6 (b), y el tiempo de ejecución ha sido de $718{,}96s$. Parece que el OpAmp se encuentra en un estado transitorio, y con el fin de visualizar el su estado estable, para la nueva ejecución de opamp2.py se ha insertado en los valores iniciales de



(a) Solucion obtenida utilizando el código contruido

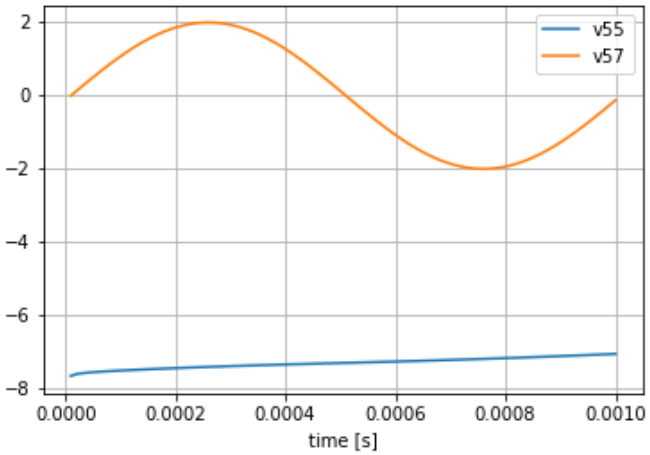


(b) Solucion obtenida utilizando el código contruido

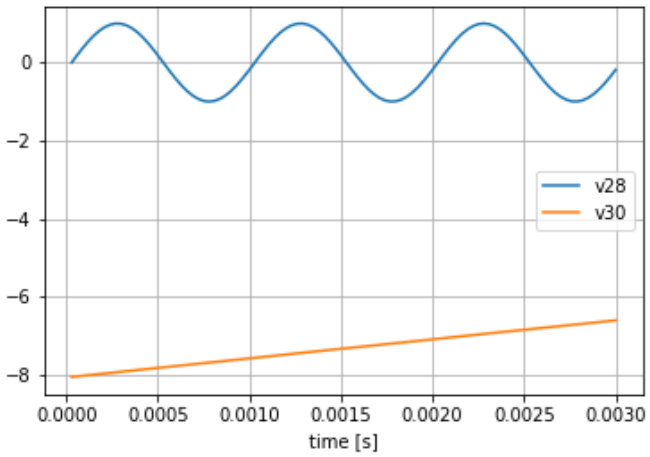
Figura 4.5: En estas gráficas se simula la respuesta de un circuito de con tres etapas amplificadoras en función del tiempo. En el caso (a) y (b), el valor de la fuente de tensión de entrada v_s se muestra con color azul, y la tensión de la resistencia RE (a), y (b), en color naranja y morado, respectivamente.

los condensadores, la última tensión obtenidas de la ejecución anterior (de ese modo se consigue un ahorro de pasos), y se ha seleccionado como punto final $tn = 0{,}018s$.

El mismo circuito se ha simulado con Pspice, y el resultado que muestra Pspice desde el principio es igual al que se observa en la Figura 4.7 (a) a partir de $t = 0{,}18s$. Esto es una clara evidencia de que Pspice ha hecho uso de los fasores para la obtención de la solución.

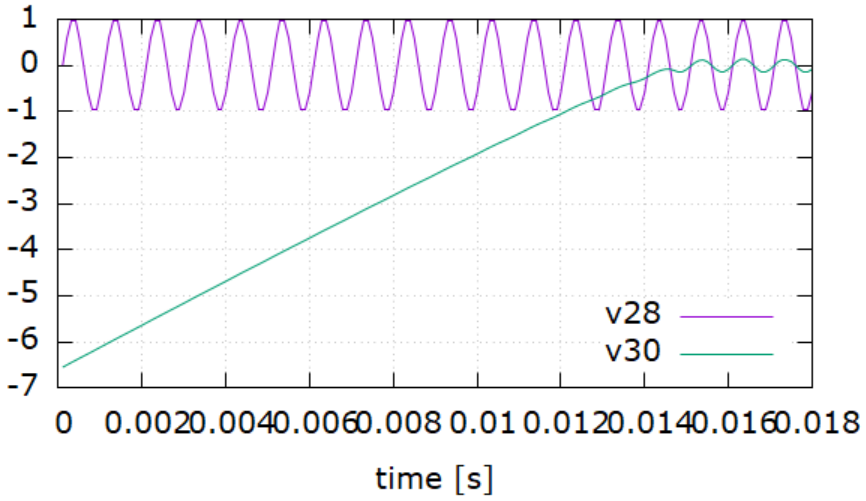


(a) Solucion obtenida utilizando opamp1.py contruido

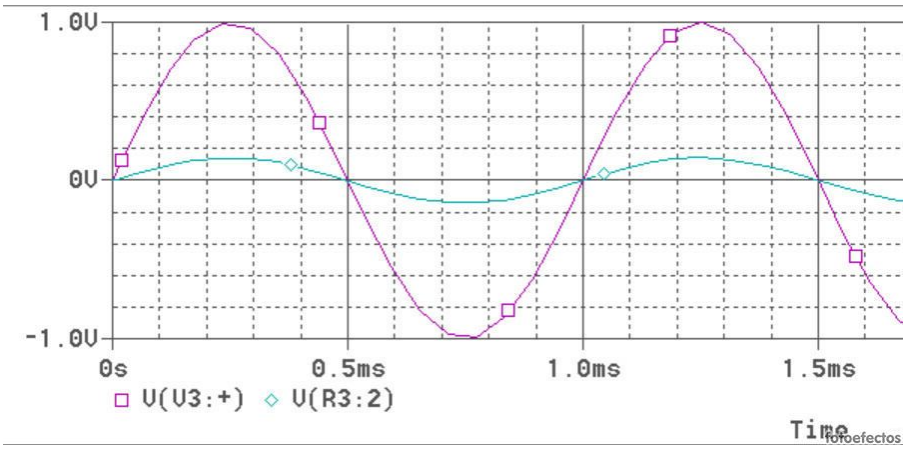


(b) Solucion obtenida utilizando opamp2.py

Figura 4.6: En estas gráficas se simula la respuesta de un amplificador inversor. En ambos se observa la misma deriva para la señal de salida ((a) azul, (b) naranja).



(a) Solucion obtenida utilizando opamp2.py



(b) Solucion obtenida de Pspice

Figura 4.7: En estas gráficas se simula la respuesta de un amplificador inversor. En el caso (a) la señal de salida (verde) es una superposición del estado estacionario con el transitorio. En el caso (b) unicamente se muestra el estado estacionario.

CAPÍTULO 5

Conclusión

5.1. Resumen

En este trabajo se ha descrito la forma de representar circuitos en forma matricial, y por ende poder resolverlos de forma mecánica aplicando métodos matemáticos. Para los circuitos dinámicos es necesario aplicar el método de Euler forward o backward, y con los resultados se ha observado un buen funcionamiento del backward. Por otro lado, para la resolución de circuitos no-lineales se han explicado dos métodos; el de Newton-Raphson para ecuaciones no lineales, y el de Newton-Raphson para sistemas de ecuaciones no lineales. El segundo solo es funcional en los casos de circuitos no lineales simples, como por ejemplo, los que solo tiene un elemento no lineal. Ademas, calcular $J\mathbf{f}$ provoca un incremento en los tiempos de ejecución. En cambio, el método de Newton-Raphson aplicado a las ecuaciones de los elemento no lineales, ha resultado ser una opción muy eficiente, prueba de ello se encuentra en la mayoría de los resultados obtenidos en el apartado anterior.

Aunque la clase *SubCircuit* no haya dado los resultados esperados, su desarrollo ha aportado herramientas muy útiles, como por ejemplo la obtención de las ecuaciones de un circuito multipuerta, de tres formas distintas.

En el proceso de implementación se han probado diferentes circuitos, y en los casos de circuitos no-lineales y grandes, no es suficiente con aplicar Newton-Raphson, por lo tanto se ha creado un método en el que se reducen los valores de las fuentes independientes, para que en el proceso de linealización de las ecuaciones no lineales no se produzcan valores altos. Después los valores de las fuentes independientes se aumentan hasta que se obtiene el resultado del circuito inicial. Esto ha dado unos resultados satisfactorios, y unos tiempos de ejecución altos, perse.

Al realizar el análisis de los diferentes lenguajes de programación, se escogió Python por ser sintácticamente mas simple, tener una amplia gamma de librerías etc. Sympy, ofrece realizar

cálculos simbólicos de todo tipo; integrales, calculo de sistemas, etc., una herramienta, a prior, útil para implementar transformadas de laplace. Sin embargo al introducirlo en el código, para dicho objetivo, se ha observado un mal funcionamiento. Sin embargo, ha sido de utilidad para definir en símbolos en las ecuaciones de los elementos; h, en los casos de elementos dinámicos, t para los dependientes temporales, y para representar los términos no-lineales en los casos de elementos no-lineales. Como cabria esperar, Pspice es mas rápido que el código construido, ya que ha estado en constante desarrollo desde su lanzamiento.

5.2. Trabajo futuro

El programa desarrollado es capaz de resolver circuitos lineales, no-lineales, y dinamicos, ademas de ofrecer diferentes tipos de analisis, segun las caracteristicas del mismo. Sin embargo, esta limitado por el tamaño del circuito que se desea analizar y los tiempos de ejecucion. Con el desarrollo actual, es una buena herramienta para su uso en los ejercicios y en algunas practicas que se dan en las asignaturas de este grado, como por ejemplo, Electronica, Electronica Analogica, Circuito Lineales y No-Lineales, etc. Para su uso en aplicaciones más amplias y complejas, debería mejorar en varios aspectos.

Uno de los problemas principales son los tiempos de ejecucion; un usuario del programa no debe esperar mucho tiempo para obtener una solucion. El metodo de Newton Raphson se caracteriza por ser rapido cuando los puntos iniciales se encuentran cerca de la raiz, y por ello seria interesante implementar metodos que de una aproximacion del punto inicial. Con la reduccion de los valores de las fuentes independientes se consigue que el sistema no diverga, sin embargo es un procedimiento de busqueda lento. Sustituyendolo por diferentes metodos de busqueda, se podria mejorar su funcionamiento.

Por otro lado, seria interesante incluir otras herramientas de analisis, como por ejemplo funciones que realicen diagramas de Bode, análisis de Fourier, etc. Para realizar Bode se tendría que analizar la relacion entre las variables de entrada y de salida con el cambio de las frecuencias (el procedimiento seria parecido al que se realiza en timeAnalisis()). Respecto al análisis de Fourier, hay varios algoritmos desarrollados que lo implementan.

De todos modos, todo esto no es de gran utilidad si el usuario no puede usarlo de forma fácil y sencillas. Por eso, un entorno grafico para realizar circuitos, seria de gran utilidad. Con el fin de evitar que el usuario tenga que instalar Python, seria recomendable crear un servidor web que incluyera este programa.

APÉNDICE A

Amplficador Operacional Simplificado

En este apartado se muestra el circuito de un Opamp 741 en forma textual, obtenido de Pspice.

```
* Model for uA741 Op Amp (from EVAL library in PSpice)
*          connections: non-inverting input
*          | inverting input
*          | | positive power supply
*          | | | negative power supply
*          | | | | output
*          | | | | |
*
.subckt uA741    1 2 3 4 5
*
c1 11 12 8.661E-12
c2 6 7 30.00E-12
dc 5 53 dy
de 54 5 dy
dlp 90 91 dx
dln 92 90 dx
dp 4 3 dx
egnd 99 0 poly(2),(3,0),(4,0) 0 .5 .5
fb 7 99 poly(5) vb vc ve vlp vln 0 10.61E6 -1E3 1E3 10E6 -10E6
ga 6 0 11 12 188.5E-6
gcm 0 6 10 99 5.961E-9
iee 10 4 dc 15.16E-6
hlim 90 0 vlim 1K
q1 11 2 13 qx
q2 12 1 14 qx
r2 6 9 100.0E3
rc1 3 11 5.305E3
rc2 3 12 5.305E3
re1 13 10 1.836E3
re2 14 10 1.836E3
ree 10 99 13.19E6
ro1 8 5 50
ro2 7 99 100
rp 3 4 18.16E3
vb 9 0 dc 0
vc 3 53 dc 1
ve 54 4 dc 1
vlim 7 8 dc 0
vlp 91 0 dc 40
vln 0 92 dc 40
.model dx D(Is=800.0E-18 Rs=1)
.model dy D(Is=800.00E-18 Rs=1m Cjo=10p)
.model qx NPN(Is=800.0E-18 Bf=93.75)
.ends
```

APÉNDICE B

Pruebas

En este apéndice se muestran los códigos, usados para realizar las pruebas del apartado *Resultados*, más la representación gráfica del circuito[1]. equivalente.py

```
import elementos as Element
import subcircuito as SCir
from Circuito import*

V = Element.VoltageSource(5)
R = Element.Resistance(20)
I1 = Element.CurrentSource(1)
I2 = Element.CCCS(5)
c = [[V, [1, 0]], [R, [2, 1]],
[I1, [3, 0]], [I2, [0, 3], [3, 2]]]

circuito = Circuit(c)
print(circuito.solution())
print(circuito.theveninEquivalent(0, 3))
print(circuito.nortonEquivalent(0, 3))
```

```
>>>{e1: 5.0, e2: 10.0, e3: 10.0, v1: 5.0, v2: 5.0, v3: 10.0, v4: -10.0, v5: 0.0, i1: 0.25, i2:
    0.25, i3: 1.0, i4: 1.25, i5: 0.25}

vth=10.0 V, rth=-5.0 ohm
inor=-2.0 A, rth=-5.0 ohm
```

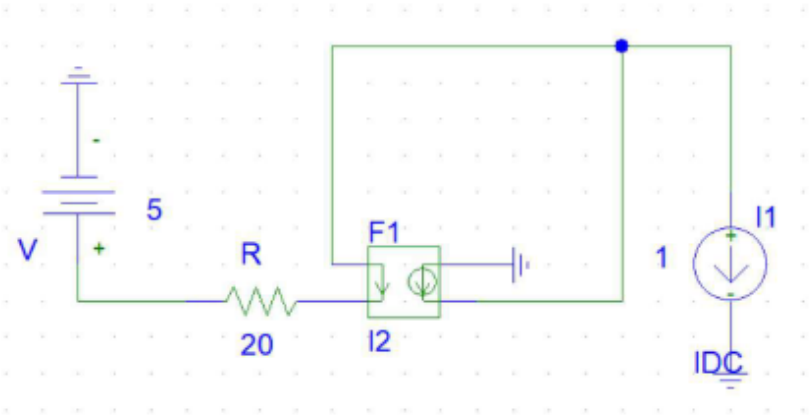
diodo.py

[1]Para la realizar la representación gráfica se ha usado Pspice.

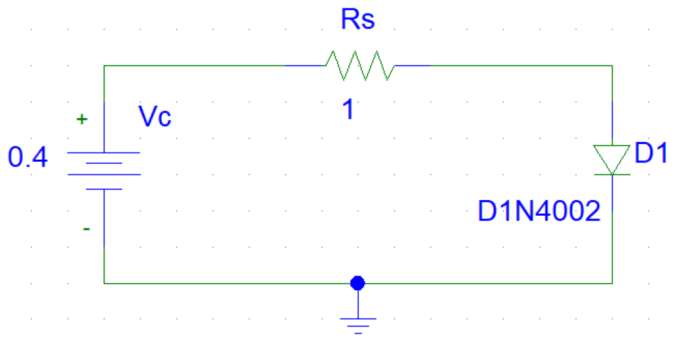
```
import elementos as Element
import subcircuito as SCir
from Circuito import*

V = Element.VoltageSource(0)
Rs = Element.Resistance(1)
D1 = Element.D1N4002()
c = [[V, [1, 0]],
[Rs, [1, 2]],
[D1, [2, 0]]]

circuito = Circuit(c)
circuito.dcAnalysis([0, 2], [0, 0], ["voltage", "voltage"],[V, [1,
    0]], 0, 4, 20)
```

rectificador.py

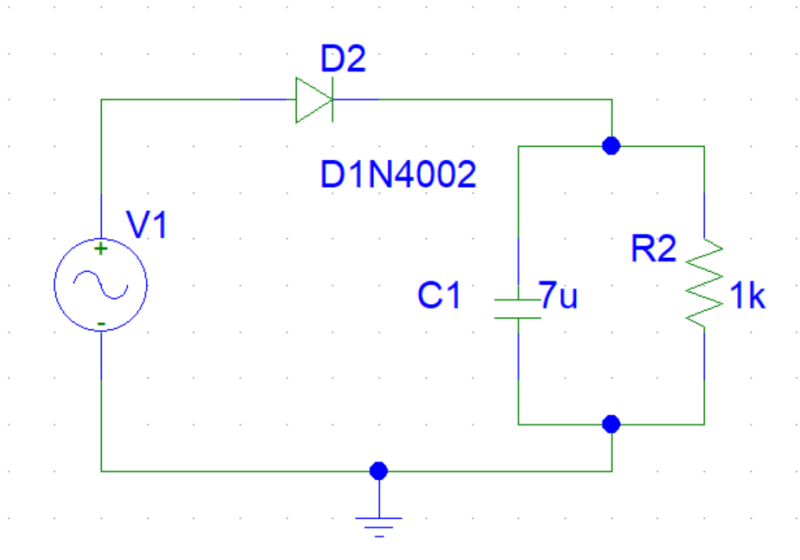
```
import elementos as Element
import subcircuito as SCir
import time*
from Circuito import*

AV = Element.ACVoltageSource(50, 60)
R = Element.Resistance(1e3)
C = Element.Capacitor(60e-7)
D = Element.D1N4002()

c = [[AV,[1, 0]], [D, [1, 2]],
[C, [2, 0]], [R, [2, 0]]]

t1=time()
circuito = Circuit(c)
circuito.timeAnalysis([0, 3], [0, 0], ["voltage", "voltage"],[V, [1,
    0]], 0, 0.025, 1200)
t2=time()
print(t1-t2)
```



(a) Representación gráfica del la variable c del archivo equivalente.py



(b) Representación gráfica del la variable c del archivo diodo.py

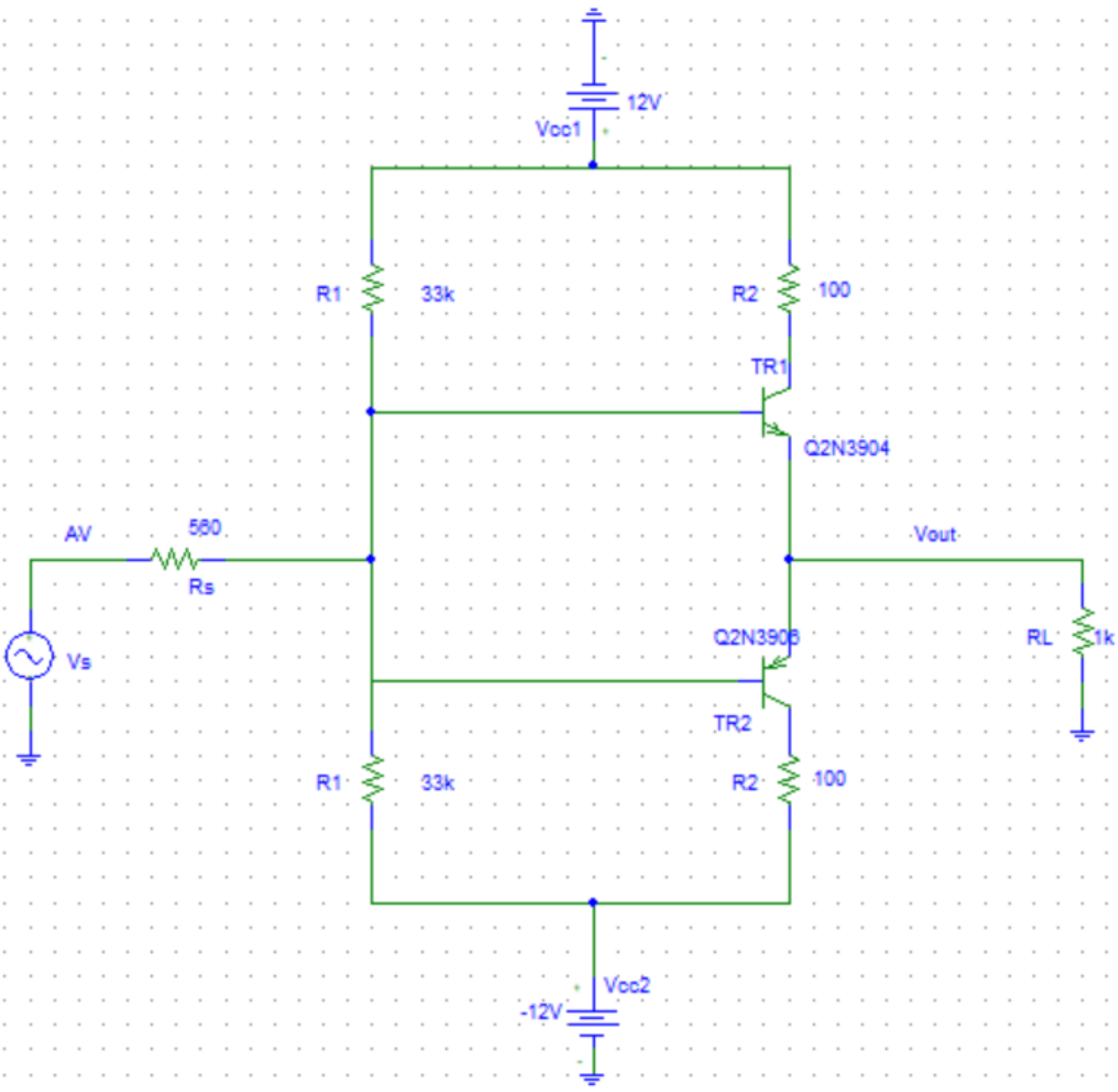


(c) Representación gráfica del la variable c del archivo rectificador.py

Figura B.1: Representaciones gráficas de los circuitos usados para realizar las simulaciones del apartado *Resultados*.

etapab.py

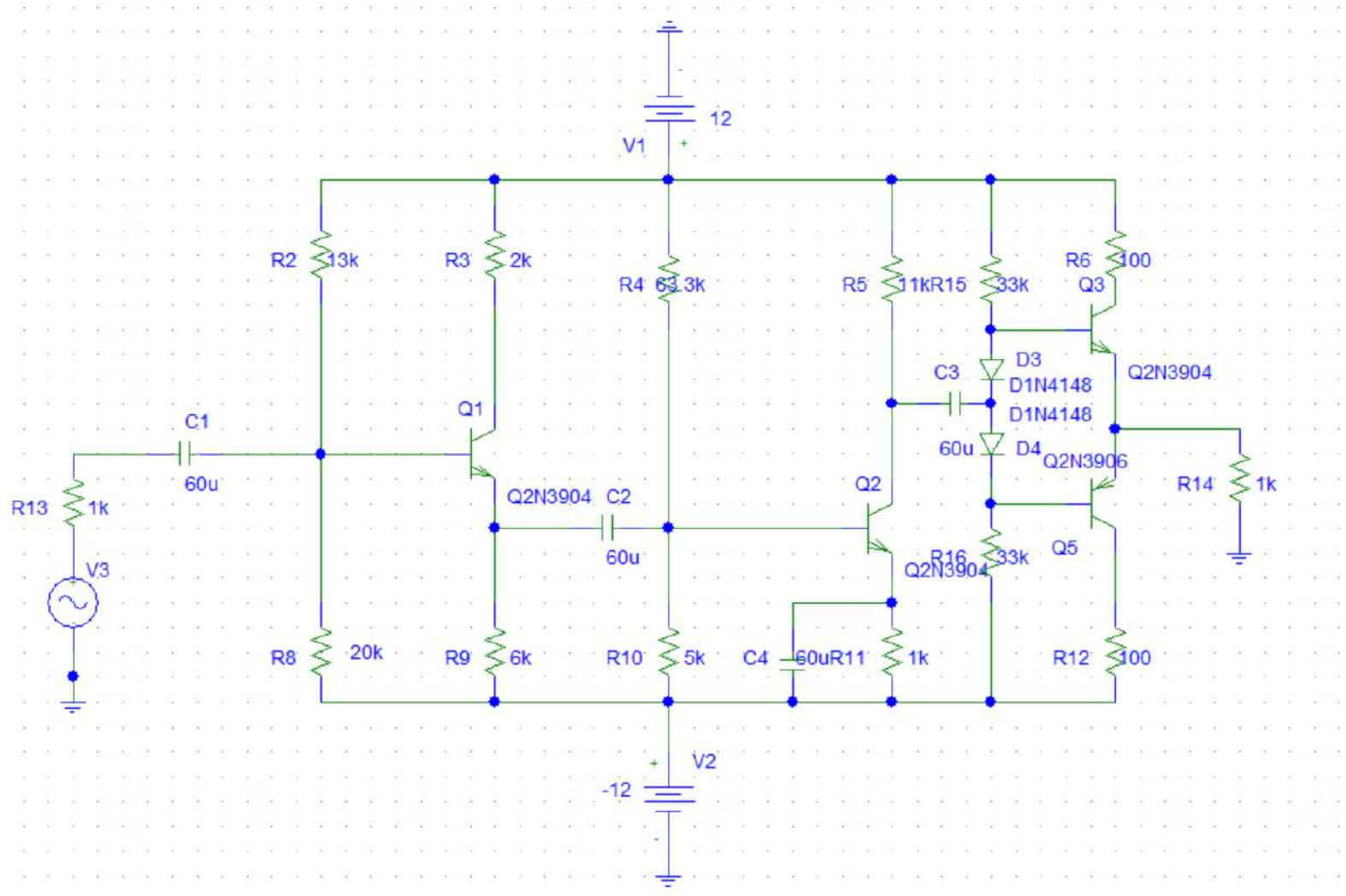
```
import elementos as Element
import subcircuito as SCir
import time*
from Circuito import*

AV = Element.ACVoltageSource(1, 1e3)
Vcc = Element.VoltageSource(12)
Rs = Element.Resistance(560)
R1 = Element.Resistance(33e3)
R2 = Element.Resistance(100)
Rl = Element.Resistance(1e3)
TR1 = Element.Q2N3904()
TR2 = Element.Q2N3906()

circuito = [[AV,[1, 0]], [Rs, [1, 2]],
                         [R1, [2, 3]], [R1, [2, 4]],
                         [Vcc, [3, 0]], [Vcc, [0, 4]],
                         [TR1, [2, 6],[2, 5]],
                         [TR2, [2, 7],[2, 5]],
                         [R2, [6, 3]],[R2, [7, 4]],
                         [Rl, [5, 0]]]

t1=time()
x=Circuit(circuito)
x.timeAnalysis([10, 0], [0, 0], ["voltage", "voltage"], 0, 0.001,
   30)
print(time()-t1)
```


12V
Vcc1
R1
33k
R2
100
TR1
Q2N3904
AV
560
Rs
Vout
Vs
Q2N3906
RL
1k
TR2
R1
33k
R2
100
Vcc2
-12V

etapa3.py

```
import elementos as Element
import subcircuito as SCir
import time*
from Circuito import*

R1 = Element.Resistance(1e3)
R2 = Element.Resistance(13e3)
R3 = Element.Resistance(20e3)
R4 = Element.Resistance(2e3)
R5 = Element.Resistance(5e3)
R6 = Element.Resistance(63.2e3)
R7 = Element.Resistance(5e3)
R8 = Element.Resistance(11e3)
R9 = Element.Resistance(1e3)
RA = Element.Resistance(33e3)
RB = Element.Resistance(33e3)
RC = Element.Resistance(100)
RD = Element.Resistance(100)
RE = Element.Resistance(1e3)
c1 = Element.Capacitor(1e-7)
c2 = Element.Capacitor(1e-7)
c3 = Element.Capacitor(1e-7)
c4 = Element.Capacitor(1e-7)
Q1 = Element.Q2N3904()
Q2 = Element.Q2N3904()
Q3 = Element.Q2N3904()
Q4 = Element.Q2N3906()
D1 = Element.D1N4002()
D2 = Element.D1N4002()
vs = Element.ACVoltageSource(1, 10e3)
vc1 = Element.VoltageSource(12)
vc2 = Element.VoltageSource(-12)

circuito = [[R1, [2,3]], [R2, [4,1]], [R3, [4,16]], [R4, [5,1]], [R5
    , [6,16]],
[R6, [7,1]], [R7, [7,16]], [R8, [8,1]], [R9, [9,16]], [RA, [11,1]],
[RB, [12,16]], [RC, [13,1]], [RD, [15,16]], [RE, [14,0]], [c1,
    [3,4]],
[c2, [6,7]], [c3, [9,16]], [c4, [8,10]], [D1, [11, 10]],
[D2, [10,12]], [Q1, [4, 5], [4,6]], [Q2, [7, 8], [7,9]],
[Q3, [11, 13], [11,14]], [Q4, [10, 15], [10,14]], [vs, [2, 0]],
[vc1, [1, 0]], [vc2, [16, 0]]]


```

```
43 stime = time()
44 c = Circuit(circuito)
45 c.timeAnalysis([24, 13], [0, 0], ["voltage", "voltage"],
46 0, 0.00001, 100)
47 etime = time()-stime
48 print(etime)
```


12
V1
R2 13k
R3 2k
R4 63.3k
R5 11k
R15 33k
R6 100
Q3
D3
C3
D1N4148
Q2N3904
D1N4148
C1
Q1
60u
D4
Q2N3906
Q2
R13 1k
60u
Q2N3904
C2
R14 1k
R16 33k
Q5
V3
60u
Q2N3904
R8 20k
R9 6k
R10 5k
C4 60u
R11 1k
R12 100
V2
-12

opamp1.py

```
import elementos as Element
import subcircuito as SCir
import time*
from Circuito import*

Q1 = Element.Q2N3904(v=[-0.1, -0.1])
Q2 = Element.Q2N3904(v=[-0.1, -0.1])
Q3 = Element.Q2N3904(v=[-0.1, -0.1])
Q7 = Element.Q2N3904(v=[-0.1, -0.1])
Q8 = Element.Q2N3904(v=[-0.1, -0.1])
Q13 = Element.Q2N3904(v=[-0.1, -0.1])
Q12 = Element.Q2N3904(v=[-0.1, -0.1])
Q14 = Element.Q2N3904(v=[-0.1, -0.1])
Q17 = Element.Q2N3904(v=[-0.1, -0.1])
Q15 = Element.Q2N3904(v=[-0.1, -0.1])
Q16 = Element.Q2N3904(v=[-0.1, -0.1])
Q18 = Element.Q2N3904(v=[-0.1, -0.1])
Q20 = Element.Q2N3904(v=[-0.1, -0.1])

Q6 = Element.Q2N3906(v=[-0.1, -0.1])
Q5 = Element.Q2N3906(v=[-0.1, -0.1])
Q4 = Element.Q2N3906(v=[-0.1, -0.1])
Q9 = Element.Q2N3906(v=[-0.1, -0.1])
Q10 = Element.Q2N3906(v=[-0.1, -0.1])
Q11 = Element.Q2N3906(v=[-0.1, -0.1])
Q19 = Element.Q2N3906(v=[-0.1, -0.1])

R1 = Element.Resistance(1e3)
R2 = Element.Resistance(50e3)
R3 = R1
R4 = Element.Resistance(5e3)
R5 = R2
R6 = Element.Resistance(50)
R7 = Element.Resistance(7.5e3)
R8 = Element.Resistance(4.5e3)
R9 = Element.Resistance(25)
R10 = R6

c = Element.Capacitor(60e-12)


r1 = Element.Resistance(2e3)
rf = Element.Resistance(1e3)
ro = Element.Resistance(500)
```

```
45
46 Vs=Element.ACVoltageSource(1, 1e3)
47
48 FT1=Element.VoltageSource(10) # Vcc+
49 FT2=Element.VoltageSource(-10) # Vcc-
50 FT3=Element.VoltageSource(0) # Vin+
51 FT6=Element.VoltageSource(0)
52 FT7=Element.VoltageSource(0)
53 FT8=Element.VoltageSource(0)
54
55 FT=[[FT1, [1, 0]], [FT2, [26, 0]], [FT3, [3, 0]], [FT4, [4, 0]],
56 [FT5, [5, 0]]]
57
58 circuito=[[FT1, [1, 0]], [FT2, [26, 0]], [r1, [3, 27]],
59 [Q1, [2, 5], [2, 6]], [Q2, [3, 5], [3, 7]],
60 [Q3, [9, 1], [9, 11]], [Q4, [28, 5], [28, 1]],
61 [Q5, [8, 10], [8, 7]],
62 [Q6, [8, 9], [8, 6]], [Q7, [11, 9], [11, 12]],
63 [Q8, [11, 10], [11, 14]], [Q9, [28, 15], [28, 1]],
64 [Q10, [30, 18], [30, 1]],
65 [Q11, [30, 19], [30, 1]], [Q12, [29, 17], [29, 26]],
66 [Q13, [29, 15], [29, 16]], [Q14, [20, 19], [20, 21]],
67 [Q15, [22, 21], [22, 23]],
68 [Q16, [10, 21], [10, 22]], [Q17, [23, 10], [23, 26]],
69 [Q18, [19, 1], [19, 24]], [Q19, [21, 26], [21, 25]],
70 [Q20, [24, 19], [24, 4]],
71 [R1, [12, 26]], [R2, [13, 26]], [R3, [14, 26]],
72 [R4, [16, 26]], [R5, [22, 26]], [R6, [23, 26]],
73 [R7, [20, 21]],
74 [R8, [20, 19]], [R9, [24, 4]], [R10, [4, 25]],
75 [c, [19, 10]], [ro, [4, 0]], [rf, [3, 4]], [vs, [27, 0]],
76 [FT3, [2, 0]], [FT6, [5, 28]], [FT7, [17, 29]], [FT8, [18, 30]]]
```

opamp2.py

```
1 import elementos as Element
2 import subcircuito as SCir
3 import time*
4 from Circuito import*
5
6 Q1=Element.Q2N3906(v=[-0.7, -0.7])
7 Q2=Element.Q2N3906(v=[-0.7, -0.7])
8 Q8=Element.Q2N3906(v=[-0.7, -0.7])
9
10 Q3=Element.Q2N3904(v=[-0.7, -0.7])
11 Q4=Element.Q2N3904(v=[-0.7, -0.7])
```

```
Q5=Element.Q2N3904(v=[-0.7, -0.7])
Q6=Element.Q2N3904(v=[-0.7, -0.7])
Q7=Element.Q2N3904(v=[-0.7, -0.7])

D1=Element.D1N4002(2.68e-9)
D2=Element.D1N4002(2.68e-9)

C=Element.Capacitor(60e-12, -1.4746408142723768)
R=Element.Resistance(50e3)

Ia=Element.CurrentSource(7.53e-14)
Ic=Element.CurrentSource(1.62e-15)

FT1=Element.VoltageSource(0)
FT3=Element.VoltageSource(10)
FT4=Element.VoltageSource(-10)
FT=Element.VoltageSource(0)

r1=Element.Resistance(2e3)
rf=Element.Resistance(1e3)
ro=Element.Resistance(500)

Vs=Element.ACVoltageSource(1, 1e3)

circuito=[[Q1, [2, 8], [2, 7]], [Q2, [1, 6], [1, 7]],
[Q3, [13, 8], [13, 4]],[Q4, [13, 6], [13, 4]],
[Ia, [3, 7]], [C, [6, 10]], [Q5, [6, 3], [6, 9]],
[R, [9, 4]], [Q6, [9, 10], [9, 4]], [D1, [11, 10]],
[D2, [12, 11]],[Ic, [3, 12]], [Q7, [12, 3], [12, 5]],
[Q8, [10, 4], [10, 5]],[FT1, [1, 0]],
[FT3, [3, 0]], [FT4, [4, 0]], [FT, [8, 13]], [r1, [14, 2]],
[Vs, [14, 0]], [rf, [2, 5]],
[ro, [5, 0]]]
```

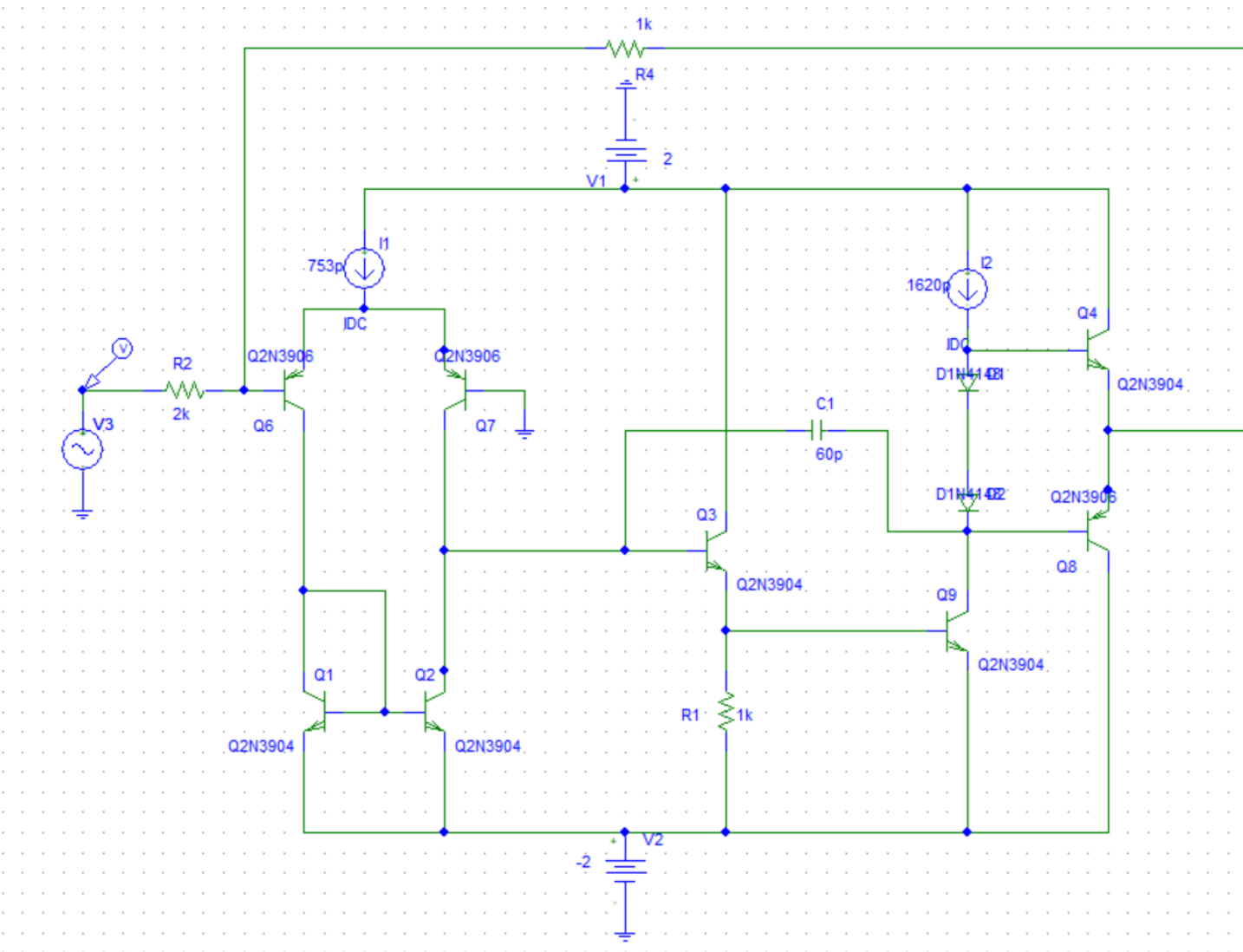
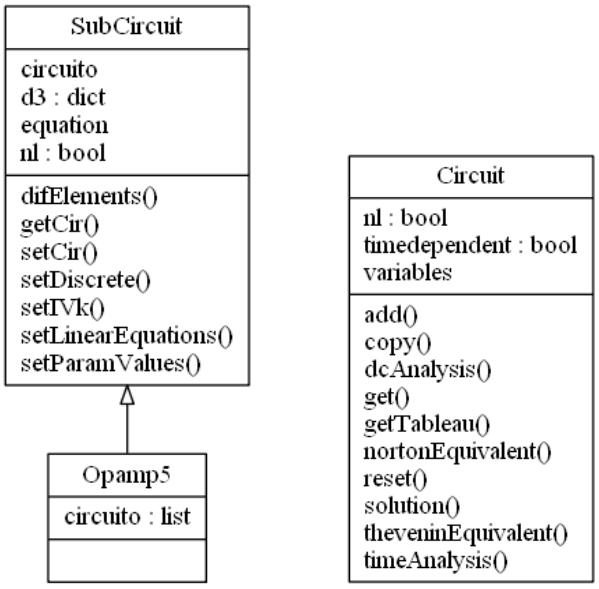


Figura B.2: Representación gráfica del circuito representado en opamp2.py

APÉNDICE C

Diagramas UML

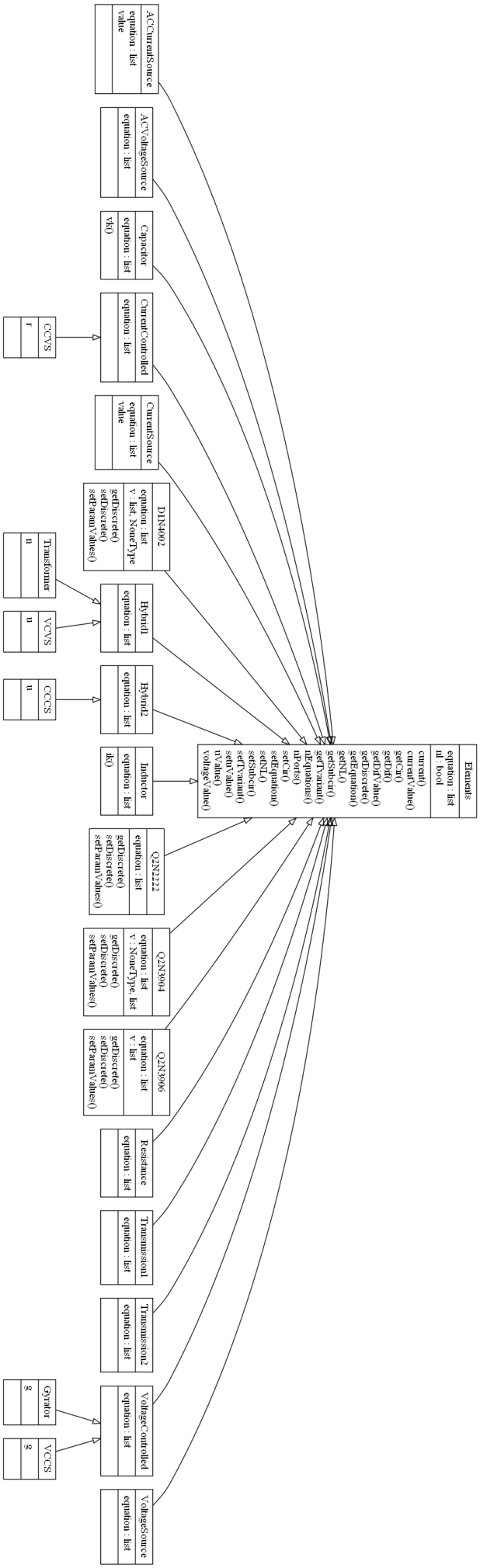
En este apéndice se presentan los diagramas UML de los módulos principales de este proyecto.



(a) Diagrama UML de la clase SubCircuit.

(b) Diagrama UML de la clase Circuit

Figura C.1: Representación de los diagramas UML obtenidos mediante Pyreverse.


ACCurrentSource
equation : list
value
ACVoltageSource
equation : list
Capacitor
equation : list
Vk()
CurrentControlled
equation : list
CCVS
r
CurrentSource
equation : list
value
D1N4002
equation : list
v : list, NoneType
getDiscrete()
setDiscrete()
setParamValues()
Transformer
n
Hybrid1
equation : list
VCVS
u
Hybrid2
equation : list
CCCS
u
Inductor
equation : list
Ik()
Elements
equation : list
isL : bool
current()
currentValue()
getCir()
getDifi()
getDifiValue()
getDiscrete()
getEquation()
getNLO
getSubcir()
setCVariant()
setEquations()
nPorts()
setCir()
setEquation()
setNLO
setSubcir()
setCVariant()
setVValue()
voltageValue()
Q2N2222
equation : list
getDiscrete()
setDiscrete()
setParamValues()
Q2N3904
equation : list
v : NoneType, list
getDiscrete()
setDiscrete()
setParamValues()
Q2N3906
equation : list
v : list
getDiscrete()
setDiscrete()
setParamValues()
Resistance
equation : list
Transmission1
equation : list
Transmission2
equation : list
Gyrator
g
VCCS
g
VoltageControlled
equation : list
VoltageSource
equation : list

APÉNDICE D

Documentación

Software Development for Circuit Analysis and Solving Documentation

Release

Carmen Legarreta

Jun 21, 2019

CONTENTS

The aim of this project is provide a free python program to solve circuits.

CHAPTER

ONE

CIRCUIT

The class Circuit has been made to represent, analysed, and solve circuits. This class provides some solving modes.

1.1 Documentation

This script contains the Circuit class. It allows the user to save the parameters of the circuit, such as; elements and each nodes. Once the circuit is defined, the user can get tableau equations, voltage solution, current solution. . . .

class circuito.**Circuit**(*circuit*)

Bases: object

A class used to represent a circuit.

circuit: list It is a list, with elements and lists, which represents a circuit. circuit parameter must have the following structure; [[r1, [0, 1]], [r2, [1, 2]], . . . , [V, [n, 0]]] [r1, [0, 1]]–>r1 resistive element connected to 0 and 1 nodes. The current will pass from 0 node to 1. [r2, [1, 2]]–>r1 resistive element connected to 1 and 2 nodes. The current will pass from 1 node to 2. . . .

variables: list It is a list with circuit variables in a symbolic form.

nl: boolean It determinates if the circuit is linear, or non-lineal, that it: if nl is True the circuit will be non-lineal, and if it False, lineal.

add(n): It adds a element/group of elements, defined by the input n,in the circuit attribute.

reset(circuit): It erases the defined circuit attribute and introduces another one.

get(): when this method is called, the user will get the objectÂ´s circuit attribute value.

getTableua(): when this method is called, the user will get the matrix representation of the tableau equations.

solution(position, port, variable, initcondition): It returns the solution of the circuit.

dcAnalysis(position, ports, variable, dcelement, t0, t1, steps): It makes a graph which shows the change of a element variable, as a VoltageSource type element changes its value

timeAnalysis(position, ports, variable, t0, t1, steps): It makes a graph which shows the change of a element variable, as the time change.

theveninEquivalent(node1, node2): It returns the thevenin equivalent of the circuit.

NortonEquivalent(node1, node2): It return the Norton equivalent of the circuit.

InputError: If the Circuit object initiate argument structure is not what supposed to be. The structure it must follow is described in the Parameters section. If the Circuit initiate argument does not meet the khirchoffs law criteria.

Circuit 0 node represents ground.

add(*n*)
This method adds elements, with each respective node notation. The adding element must have the same structure as the initiate arguments; [[element1,[node1, node2], [node3, node4],...], [element2, [node1Â´, node2Â´], [node3Â´, node4Â´],...],...]

n: list

copy()
It return a copy of this object.

dcAnalysis(*position*, *ports*, *variables*, *dcelement*, *v0*, *vi*, *steps*)
This function allows the user to see a circuit variable change as the VoltageSource type element, defined with the input variavle dcelement value change from v0, to vi.

position: list This parameter must contain integers which the position of the wanted elements to be analysed.

ports: list Specify from which ports will bet get the variable information, respectively.

variables: list the variable to be studied. There are four options; voltage, current, node1, node2. These variables are related to the Element type object.

dcelement: list It must be a VoltageSource Type element with each nodes

v0: float Voltage at which the analysis starts.

vi: float Voltage at which the analysis finishes.

steps: int number of steps from v0 to vi.

It is possible to use this function in circuits with capacitors and inductor. The step size will be 1e-5. However, elements with t dependency are not allowed.

InputError If the element parameter is not in the circuit

get()
It returns it's circuit attribute.

getTableau()
This method returns a tuple with two list. The first one holds tableu for currents, voltages, and tableau for elements. The second one has the values of the excited sources.

t, u: tuple t: list u: list

nortonEquivalent(*nodo1*, *nodo2*)
It calculates the Norton equivalent of the circuit.

nodo1: int It is equivalent to the positive node, node+

nodo2: int It is equivalent to the negative node, node-

reset(*circuit=[]*)
When reset method is called, the defined circuit attribute will be replaced by the input parameter. If there is no input parameter, then the circuit attribute will be defined by a default variable.

circuit: list It represents a circuit that will replaced the already defined circuit attribute.

solution (*position=None*, *port=None*, *variable=None*, *initcondition=None*)
It returns a dictionary which keys are the circuit variables: i_1, i_2,. . . , e_1, e_2,. . . . The values of the keys are the solutions of the circuit.

position: list This parameter must contain integers which the position , from 0 to the number of elements-1, of the wanted elements to be analysed.

port: list Specify from which ports will bet get the variable information, respectively.

variable: list The variable to be studied. There are four options; voltage, current, node1, node2. These variables are related to the Element type object.

initcondition: dictionary A dictionary which keys are the circuit variables, and each values are the initial conditions.

theveninEquivalent (*nodo1*, *nodo2*)
It calculates the Thevenin equivalent of the circuit.

nodo1: int It is equivalent to the positive node, node+

nodo2: int It is equivalent to the negative node, node-

timeAnalysis (*position*, *ports*, *variable*, *t0*, *t1*, *steps*)
This function allows the user to see a circuit variable change as the time changes.

position: list This parameter must contain integers which the position of the wanted elements to be analysed.

ports: list Specify from which ports will bet get the variable information, respectively.

variable: list the variable to be studied. There are four options; voltage, current, node1, node2. These variables are related to the Element type object.

t0: float Time at which the analysis starts.

t1: float Time at which the analysis finishes.

steps: int number of steps from t0 to t1.

InputError If the element parameter is not in the circuit

CHAPTER

TWO

ELEMENTS

The Elements class represents differents elements, all of them have terminals, equations to represent the physical behaviour, etc.

2.1 Documentation

class elementos.**ACCurrentSource**(*value*, *frequency*, *phase=0*)
Bases: *elementos.Elements*

A class used to represent a AC voltage source.

value: float The amplitude of the AC current source..

frequency: float The frequency of the AC current source.

phase: float The phase of the AC current source

equation: list The representation of the voltage source constitutive equation.

class elementos.**ACVoltageSource**(*value*, *frequency*, *phase=0*)
Bases: *elementos.Elements*

A class used to represent a AC voltage source.

value: float The amplitude of the AC voltage source..

frequency: float The frequency of the AC voltage source.

phase: float The phase of the AC voltage source

equation: list The representation of the voltage source constitutive equation.

class elementos.**CCCS**(*u*)
Bases: *elementos.Hybrid2*

A class used to represent a current controlled current source.

u: float The relation between the first and second port current when the first portÂ´s voltage is 0 V.

class elementos.**CCVS**(*r*)
Bases: *elementos.CurrentControlled*

A class used to represent a current controlled voltage source element.

r: float The impedance value of the voltage source controlled by a current.

class elementos.**Capacitor**(*value*, *v0=0.0*)
Bases: *elementos.Elements*

A class used to represent a capacitor.

value: float The capacitance value of the capacitor.

initcondition: float Determinates the voltage value across the capacitor when t=0.

equation: list The representation of the capacitor constitutive equation.

vk(*value*)

class elementos.**CurrentControlled**(*r11*, *r12*, *r21*, *r22*)
Bases: *elementos.Elements*

A class used to represent a current controlled element.

r11: float The impedance value of the first port when the second port current is 0 A, open port.

r12: float The impedance value of the first port when the first port current is 0 A.

r21: float The impedance value of the second port when the second port current is 0 A.

r22: float The impedance value of the second port when the first port current is 0 A.

equation: list The representation of the current controlled constitutive equation.

class elementos.**CurrentSource**(*value*)
Bases: *elementos.Elements*

A class used to represent a current source.

value: float The impedance value of the resistance.

equation: list The representation of the current source constitutive equation.

class elementos.**D1N4002**(*i0=1.411e-08, v=[0.4], i=[0.001]*)
Bases: *elementos.Elements*

A class used to represent a diode.

equation: list The representation of a diode constitutive equation.

v: list It is a list with the initial values.

getDiscrete()
This method return True when the equation has been linealized.

setDiscrete(*dis*)
The input value determinates wheter the element equation is going to be linealized or not.

dis: boolean True to make the equation linear. False, otherwise.

setParamValues(*v=None*, *i=None*)
To linearization a set of values are needed. So, this method takes each input arguments, and it uses to make the linealization and change the equation.

v: list v is a list with the voltage values of the elements port.

i: list i is a list with the current values of the elements port

class elementos.**Elements**(*equation=[]*, *differential=False*, *difValue=None*, *nl=False*, *tvariant=False*, *subcir=False*)

Bases: object

A class used to represent a set of electric elements.

equation: list The representation of the constitutive equations of the element.

equation: list The representation of the constitutive equations of the element.

current(terminal): It returns the way in which goes the current in the terminal, input value.

nEquations(): It returns the number of constitutive equations of the element.

nPorts(): It returns the number of ports of the element.

VoltageValue(equation, port): It takes a equation, specified by the input parameter, of the constitutive equations and it will return the value that is within the specified port voltage.

CurrentValue(equation, port): It takes a equation, specified by the input parameter, of the constitutive equations and it will return the value that is within the specified port current.

UValue(equation): It takes a equation, specified by the input parameter, of the constitutive equations and it will return the u value.

setEquation(equation): It changes the element equation with the input argument.

getEquation(): It return the element equation.

setuValue(equation, value): The u value of the equation is changed with the value input.

getDif(): It determinates if the equation has any differentiation.

getDifValue(): It specifies which variable, v or i, has been differentiate.

getNL(): It specifies whether the element is non-linear, or not.

setNL(value): It allows the user to specify the element non-linearity.

getTvariant(): It determinates wheter the element has a time dependency.

setTvariant(value=True): This method allows the user to determinate whether the elements is timer variant or not.

getSubcir(value): It allows the user to determinate whether the element is a sub-circuit or not.

getDiscret(): It determinates whether the element equation has beenlinearized.

getCir(): It returns the solution of the inner circuit of a SubCircuit type object.

setCir(cir): It gets the solution of the inner circuit of a SubCircuit type object

current(*terminal*)

It returns the way in which goes the current in the terminal, input value.

terminal: int The number of the terminal. It will take 0 or 1 value.

int 1 if the terminal number is 0, -1 otherwise.

currentValue(*equation*, *port*)

It takes a equation, specified by the input parameter, of the constitutive equations and it will return the value that is within the specified port current.

equation: int It is a number that specifies the equation that the user wants to take from the set of constitutive equations.

port: int It is a number that specifies the port from which the user wats to get current information.

float The parameter value multiplying with the current variable, specified by equation and port input parameters.

getCir()
It returns the solution of the inner circuit of a SubCircuit type object.

getDif()
It determinates if the equation has any differentiation.

boolean True if the equation has any differentiation. False, otherwise

getDifValue()
It specifies which variable, v or i, has been differentiate.

getDiscrete()
It determinates whether the element equation has been linearized.

getEquation()
It return the element equation.

getNL()
It specifies whether the element is non-linear, or not.

getSubcir()
This method determinates wheter the element is a SubCircuit type objetc

getTvariant()
It determinates whether the element has a time dependency.

nEquations()
It returns the number of constitutive equations of the element.

int number of constitutive equations of the element.

nPorts()
It returns the number of ports of the element.

int number of ports of the element.

setCir(*cir*)
It gets the solution of the inner circuit of a SubCircuit type object

cir: dictionary the keys are equal to the circuit variables, and the dictionary values are the solution of the circuit.

setEquation(*equation*)
It changes the element equation with the input argument.

equation: list A representation of the constitutive equation with lists.

setNL(*value*)
It allows the user to specify the element non-linearity.

value: boolean True to became non-linear. False, to linear.

setSubcir(*value=True*)
It allows the user to determinate whether the element is a sub-circuit or not.

value: boolean True to sub-circuits. False, otherwise

setTvariant(*value=True*)
This method allows the user to determinate wheter the elements is timer variant or not.

value: boolean True to time variant element. False, otherwise.

setuValue(*equation*, *value*)
The u value of the equation is changed with the value input.

equation: int It is a number that specifies the equation that the user want to take to change each u value

value: float It is a float number, and it will replace the u value of the specified equation

uValue(*equation*)
It takes a equation, specified by the input parameter, of the constitutive equations and it will return the u value.

equation: int It is a number that specifies the equation that the user wants to take from the set of constitutive equations.

float The u value of the equation.

voltageValue(*equation*, *port*)
It takes a equation, specified by the input parameter, of the constitutive equations and it will return the value that is within the specified port voltage.

equation: int It is a number that specifies the equation that the user wants to take from the set of constitutive equations.

port: int It is a number that specifies the port from which the user wats to get voltage information.

float The parameter value multiplying with the voltage variable, specified by equation and port input parameters.

class elementos.**Gyrator**(*g*)
Bases: *elementos.VoltageControlled*

A class used to represent a gyrator.

g: float The admittance value of the gyrator

class elementos.**Hybrid1**(*h11*, *h12*, *h21*, *h22*)
Bases: *elementos.Elements*

A class used to represent Hybrid circuit.

h11: float The impedance value of the first port when the second port voltage is 0 V.

h12: float The relation between the first and second port voltages when the first portÂ´s current is 0 A.

h21: float The relation between the first and second por currents when the second portÂ´s voltage is 0 V.

h22: float The admittance value of the second port when the first port current is 0 A.

equation: list The representation of the Hybrid1 constitutive equation.

class elementos.**Hybrid2**(*h11*, *h12*, *h21*, *h22*)
Bases: *elementos.Elements*

A class used to represent a inverse Hybrid circuit.

h11: float The admittance value of the first port when the second port current is 0 A.

h12: float The relation between the first and second port current when the first portÂ´s voltage is 0 V.

h21: float The relation between the second and first port voltage when the second port current is 0 A.

h22: float The admittance value of the second port when the second port current is 0 A.

equation: list The representation of the inverse Hybrid constitutive equation.

class elementos.**Inductor**(*value*, *i0=0*)
Bases: *elementos.Elements*

A class used to represent a inductor.

value: float The inductance value of the inductor.

initcondition: float Determinates the current value across the inductor when t=0.

equation: list The representation of the capacitor constitutive equation.

ik(*value*)

class elementos.**Q2N2222**(*ics=3.307157571149511e-15, ies=1.1403e-15, ar=0.8602961925565159, af=0.9943, v=[-0.7, -0.7], i=[0.001, 0.001]*)
Bases: *elementos.Elements*

A class used to represent a NPN transistor.

equation: list The representation of the NPN transistor constitutive equation.

getDiscrete()
This method return True when the equation has been linealized.

setDiscrete(*dis*)
The input value determinates wheter the element equation is going to be linealized or not.

dis: boolean True to make the equation linear. False, otherwise.

setParamValues(*v=None*, *i=None*)
To linearization a set of values are needed. So, this method takes each input arguments, and it uses to make the linealization and change the equation.

v: list v is a list with the voltage values of the elements port.

i: list i is a list with the current values of the elements port

class elementos.**Q2N3904**(*ics=5.135668043523316e-15, ies=6.734423e-15, ar=0.42749, af=0.9976, v=[-0.7, -0.7], i=[0.001, 0.001]*)
Bases: *elementos.Elements*

A class used to represent a NPN transistor.

equation: list The representation of the NPN transistor constitutive equation.

getDiscrete()
This method return True when the equation has been linealized.

setDiscrete(*dis*)
The input value determinates wheter the element equation is going to be linealized or not.

dis: boolean True to make the equation linear. False, otherwise.

setParamValues(*v=None*, *i=None*)
To linearization a set of values are needed. So, this method takes each input arguments, and it uses to make the linealization and change the equation.

v: list v is a list with the voltage values of the elements port.

i: list i is a list with the current values of the elements port

class elementos.**Q2N3906**(*ies=1.1537553902975205e-15, ics=1.3908311155608809e-15, ar=0.8278688524590164, af=0.994, v=[-0.7, -0.7], i=[-0.0001, -0.0001]*)
Bases: *elementos.Elements*

A class used to represent a NPN transistor.

equation: list The representation of the NPN transistor constitutive equation.

getDiscrete()
This method return True when the equation has been linealized.

setDiscrete(*dis*)
The input value determinates wheter the element equation is going to be linealized or not.

dis: boolean True to make the equation linear. False, otherwise.

setParamValues(*v=None*, *i=None*)
To linearization a set of values are needed. So, this method takes each input arguments, and it uses to make the linealization and change the equation.

v: list v is a list with the voltage values of the elements port.

i: list i is a list with the current values of the elements port

class elementos.**Resistance**(*value*)
Bases: *elementos.Elements*

A class used to represent a resistance.

value: float The value of the resistance.

equation: list The representation of the resistance constitutive equation.

class elementos.**Transformer**(*n*)
Bases: *elementos.Hybrid1*

A class used to represent a transformer.

n: float The numbers of turns in a winding.

class elementos.**Transmission1**(*t11*, *t12*, *t21*, *t22*)
Bases: *elementos.Elements*

A class used to represent a two port network.

t11: float The relation between the first and the second port voltages values when the second port current value is 0 A.

t12: float The impedance of the first port when the second port current flows in the opposite direction and its portÂ´s voltage is 0 V.

t21: float The admitance of the first port when the second port current is 0 A.

t22: float The relation between the firs and second port currents when the second portÂ´s currents flows in the opposite way and the itÂ´s voltage value is 0 V.

equation: list The representation of the transmission constitutive equation.

class elementos.**Transmission2**(*t11*, *t12*, *t21*, *t22*)
Bases: *elementos.Elements*

A class used to represent a two port network with inverse transmission parameters.

t11: float The relation between the second and first port voltages when the first port current value is 0 A.

t12: float The second port impedance value when the first port voltage is 0 V.

t21: float The second port admittance value when the first port current is 0 A.

t22: float The relation between the second and first port currents when first port voltage is 0 V.

equation: list The representation of the inverse transmission constitutive equation.

class elementos.**VCCS**(*g*)
Bases: *elementos.VoltageControlled*

A class used to represent a voltage controlled current source element.

g: float The admittance value of the current source controlled by a voltage.

class elementos.**VCVS**(*u*)
Bases: *elementos.Hybrid1*

A class used to represent a voltage controlled voltafe souce.

u: float The relation between the first and second port voltages when the first portÂ´s current is 0 A.

class elementos.**VoltageControlled**(*g11*, *g12*, *g21*, *g22*)
Bases: *elementos.Elements*

A class used to represent a voltage controlled element.

g11: float The admittance of the first port when the second portÂ´s voltage is 0 V.

g12: float The admittance of the first port when the first portÂ´s voltage is 0 V.

g21: float The admittance of the second port when the second portÂ´s voltage is 0 V.

g22: float The admittance of the second port when the first portÂ´s voltage is 0 V.

equation: list The representation of the voltage controlled constitutive equation.

class elementos.**VoltageSource**(*value*)
Bases: *elementos.Elements*

A class used to represent a voltage source.

value: float The excitation value of the source.

equation: list The representation of the voltage source constitutive equation.

elementos.**pi = 3.141592653589793**
==========Elements==========

CHAPTER

THREE

SUBCIRCUITS

The SubCircuit class is used to represent and implement differents integrated circuit, such as the Operational Amplifier.

3.1 Documentation

Created on Mon May 6 10:30:44 2019

@author: Carmen

class subcircuito.**SubCircuit**(*circuito*, *v*, *FT*, *nl=False*, *differential=False*, *cir=None*)

Bases: *elementos.Elements*

A class used to represent a set of subcircuits.

circuito: list It represents the inner circuit of the subcircuit

v: list A list with initial values of the subcircuit.

FT: list It determinates which are the input/output of the inner circuit.

nl: boolean True if the inner circuit is non-linear. False, otherwise.

differentia: boolean True if the inner circuit has any capacitor/inductor.

cir: dictionary The keys of the dictionary are the variables of the inner circuit, and the values are the solution of the circuit.

nl: boolean True if the inner circuit is non-linear. False, otherwise.

circuito: list It represents the inner circuit of the subcircuit

setDiscrete(): The input value determinates wheter the element equation is going to be linealized or not.

getCir(): It returns the solution of the inner circuit of a SubCircuit type object.

setCir(cir): It gets the solution of the inner circuit of a SubCircuit type object.

setParamValues(): This methods get values for linearization, and gets the equivalent equation.

setLinearEquations(): It recalculates the equations.

difElements(): It reuturns a list of elements with differential values on its equations.

setIVk(): It is a method for the application of the Euler method in capacitors, and inductors.

difElements()
It reuturns a list of elements with differential values on its equations.

getCir()
It returns the solution of the inner circuit of a SubCircuit type object.

setCir(*cir*)
It gets the solution of the inner circuit of a SubCircuit type object.

cir: dictionary The keys of the dictionary are the variables of the inner circuit and the values are the solutions of the circuit.

setDiscrete(*dis*)
The input value determinates wheter the element equation is going to be linealized or not.

setIVk(*V=None*, *I=None*)
It is a method for the application of the Euler method in capacitors, and inductors.

V: list V contains the voltage solutions of the ports.

I: list I contains the current solutions of the ports.

setLinearEquations()
This mehtod is only used for linear subcircuits. It is used to when a inner circuit component has a capacitor/inductor type element, that is: when those elements equations change, the equation must be recalculated.

setParamValues(*v=None*, *i=None*)
This methods get values for linearization, and gets the equivalent equation.

v: list v is a list with the voltage values of the elements port.

i: list i is a list with the current values of the elements port.

PYTHON MODULE INDEX

Bibliografía

[1] JScience official web page

`http://jscience.org/`

[2] The Fortran Company

`https://www.fortran.com/`

[3] Python official web page

`https://www.python.org/`

[4] numpydoc docstring guide

`https://numpydoc.readthedocs.io/en/latest/format.html`

[5] The fortran company.

`https://www.fortran.com/`.

[6] J. M. Garrido. *Object-Oriented Programming: From Problem Solving to Java.* Charles River Media, 2003.

[7] P. Grogono. *The Evolution of Programming Languages.* , 2002.

[8] S. G. Shasharina J. R. Cary. Comparison of c++ and fortran 90 for object-oriented scientific programming. *ELSEIVER*, 1996.

[9] E. C. Xavier J. Wainer. A controlled experiment on python vs c for an introductory programming course: students outcomes. *ResearchGate*, 2017.

[10] F. Varas N. Calvo. Curso básico de métodos numéricos. , 2009.

[11] W. Spector N. S. Clerman. *Moderm Fortran: Style and usage.* Cambridge University press, 2012.

[12] R. P. Canale S. C. Chapra. *Métodos numéricos para ingenieros.* McGraw-Hill, 2006.

[13] O. Vazquez S. Eberlein. Amplificador operacional real. , 2012.

[14] J. M. Zelle. *Python Programming: An Introdunction to Computer Science*. Franklin Beedle, 2017.

[15] J. M. Tarela. *Python Programming: An Introdunction to Computer Science*. Euskal Herriko Unibertsitatea, 2014.