[…]

```
 # Objetivo del laboratorio
 El objetivo de la presenta práctica es conocer el estándar de
simulación de circuitos [SPICE]
(http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE) y realizar
pequeñas simulaciones en corriente continua con el mismo. SPICE
es una forma elegante y sencilla de codificar circuitos
eléctricos de manera que puedan ser procesados por un ordenador.
Mediante un sencillo lenguaje podemos definir resistencias,
fuentes de alimentación, etc., las conexiones entre ellos y los
resultados que deseamos obtener.

 # El estándar SPICE
 **SPICE** es una abreviabiación de *Simulation Program with
Integrated Circtuit Emphasis*.
 Se trata básicamente de un método estándar para describir
circuitos usando texto plano en
 lugar de una representación gráfica (o *esquemática*). A esta
descripción en texto se
 la llama también **netlist** y básicamente se corresponde con la
*lista* de los componentes del circuito y cómo estos están
conectados entre sí, es decir, de los nodos de unión.
 Los ficheros netlist pueden tener extensiones `.cir`, `.net`,
`.ckt`, ó `.sp` y es muy común encontrárselos con cualquiera de
estas.

 Existen en el mercado muchas variantes (intérpretes) de Spice,
aunque el original fue descrito
 en la Universidad de Berkeley. En la lista de intérpretes de
Spice tenemos desde esfuerzos y proyectos comerciales hasta *open
source* y regidos por distintas comunidades de usuarios y
programadores.

> **Pregunta:** Enumera todos los intérprete de Spice que puedas
encontrar. Crea una tabla en Markdown con varias columnas (para
el nombre, fabricante, versión actual, licencia y alguna
característica sobresaliente). Aquí tienes un ejemplo del que
puedes partir y seguir completando:

| Intérprete | Licencia           | Fabricante              |
Características  |
| ---------- | --------           | ------------------      | ---
------------ |
| Ahkab      | GPL                | Giuseppe Venturini      |
Basado en Python |
| XSPICE     | GPL                | Georgia Tech            |
Agregado de modelos Digitales y Anàlogos para simulacion
                           |
```

CIDER[^1]	GLP	Un. Berkeley	Agregado de Simulaciòn de semiconductores y manipulaciòn de los paràmetros de los modelos fìsicos
SPICE OPUS	Free (BSD)	Un. Ljubljana	Los Modelos pueden ser cargados desde .dll[^2], nutmeg como lenguaje interpretado para sesiones interactivas
ngspice	Free (BSD)	Open Project	Al ser **open** source permite modificaciones de cualquier tipo
LTSpice	Freeware	Linear Techonolgy[^3]	Gran cantidad de formatos de entrada soportados, permite la exportaciòn de los netlist
HSpice	Propietaria	Synopsys	Posee modelos únicos MOS / BJT
PSpice	Propietaria	Cadence Design Systems	Modelo Inductor nolineal
TINA	Propietaria	DesignSoft	TINA-TI version limitada para pequeños dispositivos
QUCS	GPL	Michael Margraf	Portabilidad
ADIsimPE	Free (BSD)	Analog Devices	Basado en SIMetrix/SIMPLIS, Optimizado para circuitos Lineales
Dr. Spice	Free (BSD)	-	Alta Portabilidad, Poco Tamaño y Bajo Consumo de Recursos
Proteus	Propietaria	Labcenter Electronics	Rapidez de Simulacion y posibilidad de introducir el programa en el microcontrolador
ICAP/4	Propietaria	Intusoft	Basado en Berkeley SPICE 3 analog simulation y Georgia Tech XSPICE
PartSim	Propietaria	PartSim	Intuitivo y Facil de Usar
SIMetrix	Propietaria	SIMPLIS	Intuitivo y Facil de Usar
TopSpice 8	Propietaria	Penzar	Varias Extensiones para simplificacion de la simulaciòn
Multisim	Propietaria[^4]	National Instrument	Varias Extensiones para simplificacion de la simulaciòn, Soporte 24x7

| gEDA | GLP | Ales Hvezda | Captura Esquemàtica y Placas de Circuitos Impresos |
| Gnucap | GLP | Albert Davis | Modo Batch Y Modo Server |

[^1]: Anteriormente llamado CODECS.

[^2]: Significa que la modalidad de importaciòn de los modelos es no solo portable sino fàcilmente adoptable.

[^3]: Actualmente parte del grupo Analog Devices.

[^4]: Permite una licencia para estudiantes.

Links de Interès
1. Ahkab: (https://ahkab.readthedocs.io/en/latest/)
2. XSPICE: (http://ngspice.sourceforge.net/xspice.html)
3. CIDER: (https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/cider/)
4. SPICE OPUS: (http://www.spiceopus.si/)
5. ngspice: (http://ngspice.sourceforge.net/)
6. LTSpice: (https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html)
7. HSpice: (https://www.synopsys.com/verification/ams-verification/hspice.html)
8. PSpice: (https://www.pspice.com/)
9. QUCS: (http://qucs.sourceforge.net/)
10. ADIsimPE: (https://www.analog.com/en/design-center/interactive-design-tools/adisimpe.html)
11. Dr. Spice: (http://dktkonline.tripod.com/dktkonline_files/Course/ECT2036/SIG2/DrSpice.htm)
12. Proteus: (https://www.labcenter.com)
13. ICAP/4: (http://www.intusoft.com/)
14. PartSim: (https://www.partsim.com/)
15. SIMetrix: (http://drumknott.simplistechnologies.com/)
16. TopSpice: (http://penzar.com/topspice/topspice.htm)
17. Multisim: (http://www.ni.com)
18. gEDA: (http://www.geda-project.org/)
19. Gnucap: (http://www.gnucap.org/dokuwiki/doku.php?id=gnucap:start)

Menciones sin trascendencia pero existentes
1. Spice-It!
2. MicroCad
3. MacSpice
4. Oregano
5. TclSpice

> **Pregunta:** ¿Qué comparación puedes efectuar entre C y Spice como estándares (lenguajes) y sus respectivas implementaciones en software? ¿Qué implementaciones reales (compiladores) del lenguaje C conoces?

Tenemos que partir de la base que C es un lenguaje de tipos de datos estáticos, débilmente tipificado, de medio nivel, ya que dispone de las estructuras típicas de los lenguajes de alto nivel, pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores (tan variados que existen) suelen ofrecer extensiones que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos. Ahora bien, SPICE es una suerte de Framework, un programa, una aplicacion, "algo" ya compilado, creado con una finalidad que en sus últimas dos versiones (SPICE2, SPICE3) fueron escritas en C, sin entrar en detalle con la versión 1 (SPICE) que fue escrita en fortran.
Partiendo de esta base ya tenemos una diferencia radical, uno es un lenguaje de programación general, es decir permite hacer programas, métodos, funciones, operaciones de casi cualquier índole mientras que SPICE es un programa ya escrito y funcional que permite realizar simulaciones de circuitos, es decir operar sobre un caso concreto o un tipo de desarrollo.
Otra de las grandes diferencias es que C permite crear compiladores mientras que SPICE permite crear simulaciones, es decir, uno es para crear cosas el otro es para simular circuitos (radical diferencia debemos aceptar).
Si tomamos en cuenta las definiciones formales de programación, otra diferencia que encontramos es que C es un lenguaje de propósito general mientras que SPICE es un lenguaje de propósito especifico.
Ahora bien, tomando como contexto los "lenguajes" estándares para las "cosas", aquí presenciamos una similitud, tanto SPICE como C se convirtieron en estándares en sus respectivas áreas, gracias a ciertas características comunes que tiene todo “lenguaje” o “forma” de escritura en sistemas, estas características a grandes rasgos (y por mencionar algunas) son:

* Excelente funcionabilidad
* Capacidad de abarcar el 99% de los casos que los usuarios fueron presentando
* Baja tolerancia a fallos
* Mecanismos claros de control de flujo y errores
* Estructuras de decisión
* Portabilidad
* Adaptabilidad
* Robustez
* Eficiente
* Conjunto reducido, claro y objetivo de palabras claves

También podemos hablar de algunas constantes, por ejemplo, tanto en C como en SPICE se lograron formas de escritura simplistas, es decir, se intenta que con unas pocas instrucciones sea suficiente para definir un comportamiento.
Por otro lado, hay que destacar que como mencionamos antes, C es un lenguaje que puede ser considerado relativamente a bajo nivel, es decir, permite interactuar casi directamente con cuestiones tales como punteros de memoria, instrucciones al microprocesador, etc. Esta característica es una gran diferencia respecto a SPICE (Recordemos la diferencia básica; SPICE es una aplicación ya compilada utilizada para un sector especifico de funcionabilidades mientras que C es un lenguaje para crear “cosas”)

Propiedades de C
* Núcleo de lenguaje simple
* Funciones matemáticas
* Gestión de archivos
* Lenguaje estructurado
* Lenguaje de preprocesador
* Acceso a memoria de bajo nivel punteros.

Compiladores de C
A decir verdad, conocer conozco varios compiladores, tanto en mi ingeniería anterior como en mi licenciatura tarde o temprano terminábamos viendo uno u otro al menos en mención. La variedad de compiladores radica en la portabilidad y compatibilidad de C, amén de la multiplicidad de beneficios que tiene como lenguaje estructurado y que mencionamos anteriormente. Las diferencias entre los compiladores son variopintas, es decir, mientras algunos suman ciertas capas de abstracción o permiten por ejemplo "optimizaciones automáticas", detección de errores pre ejecución (LINT[^5]), automatización de compilación y deploy (make[^6]), detección de "memory leak" (valgrind[^7]), debbuger o monitorización de la ejecución "paso a paso" (gdb, dbx [^8]), etc.

Nombre	Autor
Aztec C	Manx Software Systems
Clang	LLVM Project
CoderStudio	Manos
GCC C	GNU Project
IAR C/C\+\+ Compilers	IAR Systems

Intel C\+\+ Compiler \(icc\)	Intel
\(Borland\) Turbo C	Embarcadero
VBCC	Volker Barthelmann
Visual C\+\+ Express	Microsoft
Oracle C compiler	Oracle
Watcom C/C\+\+, Open Watcom C/C\+\+	Watcom
Wind River \(Diab\) Compiler	Wind River Systems
XL C	IBM

[^5]: Herramienta de programación utilizada para detectar código sospechoso, confuso o incompatible https://es.wikipedia.org/wiki/Lint, https://invisible-island.net/personal/lint-tools.html

[^6]: Herramienta de gestión de dependencias https://es.wikipedia.org/wiki/Make, https://www.tutorialspoint.com/unix_commands/make.htm

[^7]: Herramientas para depuración de problemas de memoria y rendimiento de programas https://es.wikipedia.org/wiki/Valgrind, https://valgrind.org/docs/manual/quick-start.html

[^8]: Debbuger https://es.wikipedia.org/wiki/GNU_Debugger

Elementos de un netlist

Como acabamos de comentar, un netlist se corresponde con la codificación de los elementos electrónicos de un circuito y las uniones entre los mismos. Veamos con más concreción qué partes y secciones lo componen.

Comentarios

La primera línea de un netlist se corresponderá siempre con un comentario. A partir de esta línea se pueden introducir más comentarios pero tienen que ir siempre precedidos de un `*`. Ejemplo:

```spice
Mi primer circuito
* Otro comentario
* más comentarios
*
```

```
## Dispositivos básicos de un circuito
Los elementos de un netlist son los mismos que encontramos en cualquier circuito eléctrico sencillo,
tales como resistencias, **condensadores**, **bobinas**, **interruptores**, **hilos** y **fuentes** de alimentación.
Para distinguir uno de otro, se reserva una letra característica: `V` para fuentes de alimentación, `R` para resistencias, `C` para condensadores y `L` para bobinas. También es posible usar estas letras en su versión en minúscula (`r`, `v`, `c`, `l`, etc.).
Después de esta letra característica se puede sufijar cualquier texto para diferenciar un elemento de otro (números, letras, palabras, etc.). Ejemplo:

```
* Una resistencia
R1
*  Otra resistencia
R2
* Fuente de alimentación
V
* Un condensador
Cprincipal
```

Conexiones
A continuación de indicar el elemento eléctrico, tenemos que informar a Spice cuáles
son los puntos de unión tanto a un lado como al otro del elemento.
Así es como Spice sabe qué está conectado a qué: porque comparten un **punto**
(o **nodo**, aunque este término se reserva sobretodo a uniones de más de dos elementos)
que hemos señalizado correctamente. Para nombrar nodos, lo mejor es emplear una
numeración secuencial: 0...n. **La enumeración de los puntos de unión es completamente
a nuestro criterio**.

```
* Una resistencia
* entre cables 0 y 1
R1 0 1
```

Sólo es necesario seguir un criterio: en el caso de una
fuente de alimentación, el nodo que pondremos primero será
aquel que está más cerca del *borne* positivo. Ejemplo:

```spice
```
```

```
 * Para una fuente indicamos primeramente conexión a nodo
positivo.
 v 2 3 type=vdc vdc=1
 ```

En el *caso de LTspice* no es necesario indicar los parámetros
`type=vdc` y `vdc=X`, sino que si no se especifica nada, se
supone que el último valor es el del voltaje a corriente
continua:

```spice
* Especificación de una fuente de alimentación de 10 V en
corrient continua en el caso de LTspice
v 0 1 10
```

Aquí tienes un ejemplo gráfico de los componentes comentados
justo arriba (resistencia y voltaje):


(https://raw.githubusercontent.com/pammacdotnet/spicelab/master/r
esistencia%20y%20pila%20con%20nodos.svg?sanitize=true)

 ## Unidades en SPICE

 Las unidades de las magnitudes características del circuito son
siempre [unidades
 del Sistema Internacional]
(https://en.wikipedia.org/wiki/SI_electromagnetism_units) y no es
necesario indicarlo explícitamente en el netlist.

 La forma de especificar múltiplos de estas cantidades es
añadiendo una letra.
 Básicamente las que nos interesan y las que suelen aparecer
mayoritariamente son `k` para "kilo-," `m` para "mili?" y `u`
para "micro?".

 > **Pregunta:** Crea una tabla en Markdown con todos los
prefijos de múltiplos que puedas, su abreviatura y su
equivalencia numérica.

Prefijo	Descripciòn	Equivalencia Numérica
T	tera	10E12
G	giga	10E9
Meg	mega	10E6
K	kilo	10E3
M	milli	10E-3
U	micro	10E-6
N	nano	10E-9
P	pico	10E12

| F | femto | 10E-15 |

Para PSpice tenemos un agregado: clock cycle

En el caso de las fuentes de alimentación hemos de especificar si se trata de corriente contínua (`vdc`) o alterna (`ac`).

```
* Una resistencia de 5 Ohmios
R2 1 0 5
* Una pila de 10 Voltios (continua)
V1 1 0 type=vdc vdc=10
* Una resistencia de 5 kΩ
RX 2 4 5k
```

> **Pregunta**: ¿qué unidades del Sistema Internacional relacionadas con la asignatura -y los circuitos en general- conoces? Responde aquí mismo en una celda de Markdown con una tabla.

Sufijo	Descripciòn
V	Voltios (Volts - V - Circuitos Eléctricos Un. 4)

| A | Amperes (Amps - A - Circuitos Eléctricos Un. 5)

| Hz | Hercio (Hertz - Hz - EJ: Efecto Fotoeléctrico)

| Ohm | ohm (Ohmios - W - EJ: Circuitos Eléctricos Un. 5)

Valores iniciales

Aparecen justo al final de la definición del componente (`ic`). Suelen aplicarse principalmente con condensadores.

```
* Una condensador inicialmente no cargado
c 1 0 1u ic=0
```

Fin del circuito

El fin de la descripción de un netlist se especifica mediante el comando `.end`.

```spice
* Mi primer circuito
```

```
V 1 0 vdc=10 type=vdc
R 1 0 5
* Fin del circuito
.end
```

Comandos SPICE para circuitos en corriente continua

Además de la descripción del circuito, hemos de indicar al intérprete de Spice qué
tipo de análisis queremos realizar en sobre el mismo y cómo queremos presentar
la salida de la simulación. Los comandos en Spice empiezan por un `.` y suelen
escribirse justo al final del circuito, pero antes del comando `.end`.

```
 Mi primer circuito
* Aquí van los componentes
R 1 0 6k
...
* Comandos
.op
...
* Fin del circuito
.end
```

> **Pregunta**: Hasta lo que has visto del lenguaje Spice, ¿dentro de qué tipo o conjunto de lenguajes encajaría? ¿Funcionales? ¿Específicos de dominio? ¿Procedurales? ¿Estructurados? ¿Orientado a Objetos ¿Funcionales? Justifica tu respuesta.

A mi consideración creo que todo lenguaje, si bien tiene una categoría primordial y característica siempre son parte de un conjunto de tipos, es decir, tomando como ejemplo JAVA o C#, ambos son leguajes estructurados y orientados a objetos, e incluso pueden tomar una orientación a eventos, pueden ser imperativos y funcionales, por lo tanto, refiriéndonos específicamente a SPICE mi opinión es que:
* Es estructurado porque: responde a una secuencia bien definida de pasos, obviando quizás la definición formal de Secuencia - Condicional - Bucle, pero participando estrictamente a su primer mención, es decir Secuencia, ya que si definimos un circuito y ponemos por ejemplo la instrucción de .op antes de definir sus componentes la simulación necesariamente fallara alegando faltantes en la declaración SECUENCIAL del circuito.
* Es especifico de dominio porque: como ya mencionamos antes, SPICE responde a un problema de dominio en particular, o permite

desarrollar una técnica de representación o resolución de problemas específico.
* Es orientado a objetos porque: aunque no responde 100% a la filosofía orientada a objetos, en el sentido de crear clases, implementar herencia, manipular el polimorfismo, si es cierto que si tomamos la definición formal de objeto como “algo” que ofrece un comportamiento, definición, gestión y uso especifico con propiedades puntuales, podríamos decir, que SPICE participa en parte de la programación orientada a objetos, por ejemplo: si tomamos una resistencia o definimos una fuente, estos tienen propiedades que son especificas de cada uno de ellos que alteran de una u otra forma el comportamiento **del** circuito y obligan al usuario a definirlos de una forma concreta asignando valores a sus propiedades de una determinada forma pre definida.
* NO es funcional (ni imperativa) en su sentido formal porque: no está basado en el uso de funciones matemáticas **in** situ, ni se centra en los cambios de estado ni interactúa a nivel de eventos en la mutación de variables. Si bien es cierto que responde en cierta forma a la premisa o definición formal de subrutina (Entrada - Proceso - Salida) ya que definimos un circuito entre los cuales definimos un punto de inicio o nodo inicial en una secuencia, tomamos el “proceso” como el circuito mismo y sus componentes y como alteran a la simulación y por último esperamos obtener una salida, la representación formal de Funcional e Imperativa no es una característica distintiva de SPICE.
* NO es procedural porque: si nos remitimos a la definición formal de SPICE, recordamos que es un lenguaje de dominio, utilizado para simular y modelizar circuitos, en contraposición a las estructuras de los lenguajes procedurales en donde definimos una suerte de biblioteca de funciones o conjuntos de procedimientos frecuentemente utilizados.

Veamos los principales comandos de simulación:

- `.op` es el comando más sencillo que podemos emplear en. Devuelve el voltaje e intensidad en cada ramal y componente **del** circuito. Este comando no necesita parámetros.
- `.dc` es uy parecido al comando `.op` pero nos permite cambiar el valor **del** voltaje de una fuente de alimentación en pasos consecutivos entre el valor A y el valor B.
En el caso de que la fuente tuviera asignada ya un valor para su voltaje, este sería ignorado. Ejemplo:

```spice
* Variamos el valor del voltaje
* de la fuente "v" de 1 a 1000
* en pasos de 5 voltios
v 1 0 type=vdc vdc=10
.dc v 1 start=1 stop=1000 step=20
v2a 2 4 type=vdc vdc=9
```

```
 * Igual para v2a. Se ignora su voltaje de 9V
 .dc v2a start=0 stop=10 step=2
 ```

 - El comando `.tran` realiza un análisis en el tiempo de los parámetros del
 circuito. Si no se emplea la directiva `uic` (*use initial conditions*) o esta es igual a cero, este análisis se realiza desde el punto estable de funcionamiento del circuito hasta un tiempo `tfinal`.
 y en intervalos `tstep`. Si empleamos un varlor distinto para parámetro `uic`,
 entonces se hará uso de las condiciones iniciales definidas para cada componente
  (típicamente `ic=X` en el caso de los condensadores, que da cuenta de la carga incial que estos pudieran tener).


 ```
 * Hacemos avanzar el tiempo entre
 * tinicial y tfinal en pasos tstep
 .tran tstart=X tstop=Y tstep=Z uic=0/1/2/3
 ```

 `X`, `Y` y `Z` tienen, evidentemente unidades de tiempo en el S.I. (segundos).

 > **Pregunta**: El parámetro `uic` puede tener varios valores y cada uno significa una cosa. Detállalo usando un celda Markdown y consultando la [documentación de Ahkab](https://buildmedia.readthedocs.org/media/pdf/ahkab/latest/ahkab.pdf).

Para poder definir el comportamiento de UIC antes debemos definir:
* Netlist: es un archivo de texto en donde se describen los circuitos.
* .tran (Análisis Transitorio): Realiza un análisis transitorio desde tstart (por defecto es 0) hasta tstop, utilizando los pasos proporcionados como paso inicial y el método especificado (si lo hay, de lo contrario, por defecto es Euler implícito).
* .op (Punto Operativo): es un tipo de análisis en donde se busca buscar una solución de corriente continua a través de un método de iteración llamado Newton Rhapson.
* .ic (Condición Inicial): describe la condición inicial de un circuito (por defecto es 0)
* ic_label: nombre o definición de la condición inicial.

El comando UIC (Usar Condiciones Iniciales) se utiliza en conjunto con el análisis del tipo .tran, es decir analizar el circuito en un tiempo determinado, mediante este comando, podemos

alterar los valores iniciales del circuito ya que por defecto se define que el análisis comienza desde el punto estable del circuito hasta un tiempo final (tfinal). En otras palabras, este comando nos permite definir el estado del circuito en un tiempo t (tstart). Los valores disponibles son:
* Uic = 0: especifica que todos los voltajes, corrientes y valores iniciales serán cero en el inicio o tiempo de comienzo (tstart).
* Uic =1: especifica que todos los voltajes, corrientes y valores iniciales en el tiempo de comienzo (tstart) serán los mismos que el ultimo resultado del análisis .op.
* Uic =2: especifica que todos los voltajes, corrientes y valores iniciales en el tiempo de comienzo (tstart) serán los mismos que el ultimo resultado del análisis .op en donde se establecieron los valores de las corrientes a través de inductores y los voltajes en los condensadores especificados en su propio .ic.
* Uic = 3: básicamente se importa un .ic predefinido para actuar de entrada, tiene como requisitos que exista una directiva .ic en el netlist y que el nombre del .ic y ic_label coincidan.

Intérprete SPICE que vamos a usar: Ahkab

Tras un estándar siempre hay una o varias implementaciones. Ahkab no deja de ser una implmentación más en Python del estándar Spice.

> **Pregunta:** Comenta las distintas implementaciones de lenguajes y estándares que conozcas. Hazlo usando una tabla en Markdown. [Aquí](https://www.markdownguide.org/extended-syntax/#tables) tienes un poco de ayuda (aunque antes ya se ha puesto el ejemplo de una tabla).

Nombre	URL	Autor	Licencia
PySpice	https://github.com/FabriceSalvaire/PySpice	Fabrice Salvaire	GNU
spiceypy	https://github.com/AndrewAnnex/SpiceyPy	Andrew Annex	MIT

PySpice

PySpice: implementación en Python de SPICE, que permite tanto la simulación de circuitos analógicos y digitales, como la comprobación de su funcionamiento y posterior análisis y estudio. PySpice esta compuesto por:

* SCHEMATICS: editor gráfico que permite capturar y dibujar circuitos eléctricos para que puedan simularse en PSPICE y procesarse en PROBE (procesador gráfico).
* PSPICE: Permite realizar la simulación analógica y digital de un circuito.

* PROBE: Permite analizar y manipular los resultados de la simulación de forma gráfica.
* STMED: Permite generar estímulos analógicos y digitales que pueden utilizarse como generadores de entrada en la simulación de circuitos.
* PARTS : Permite modelar componentes electrónicos activos y pasivos.

Algunas de sus características son:

* Soporte para Ngspice y simulación de circuitos en Xyce
* Múltiple plataforma
* Licenciamiento bajo GPLv3
* Implementación parcial de SPICE NetList
* Exportación de las simulaciones a Numpy
* Manejo de multiplicidad de Unidades
* Editor intuitivo

spiceypy
spiceypy: es un wrapper en Python para la utilización de SPICE basado en SPICE Toolkit (NAIF)

Algunas de sus características son:

* Múltiple plataforma
* Licenciamiento bajo MIT
* Permite utilizar Anaconda, PIP, etc.
* Multiplicidad de extensiones y utilidades

> **Pregunta:** Describe brevemente este software (creador, objetivos, versiones, licencia, características principales, dependencias, etc.).

Introducción y Objetivo
Ahkab es un simulador de circuitos electrónicos **del** tipo SPICE escrito en Python 2 y 3, el mismo es una suerte de extensión, de librería para Python. Como bien menciona el creador, ahkab surgió como un experimento o una prueba de concepto, citándolo “No tenemos expectativas de que nuestra herramienta de simulación de circuito pequeño, a veces defectuosa y de prueba de concepto, reemplace a los simuladores de circuito convencionales: son convencionales por buenas razones y merecen mucho los elogios y el dinero que pagamos. Sería tonto pensar lo contrario”, lo cual es un resumen muy claro de su punto de vista. Ahora bien, Ahkab viene a desmitificar la complejidad **del** análisis y simulaciones de circuitos, es una simplificación, en algún punto, que permite verificar que sucede al ejecutar las simulaciones, diseñar circuitos mas fácilmente, mas generalmente ya que esta escrito en un lenguaje mundialmente utilizado, permitiéndonos como dice el auto “ver bajo el capot” de formas tales que nos permite, por ejemplo, obtener las matrices de ecuaciones utilizadas, modificar

los algoritmos que se utilizan para las simulaciones, corregirlos, modificarlos, adaptarlos, etc. Por otro lado, pone en manifiesto esos extensos documentos científicos que muchas veces carecen de implementaciones practicas por ende son una suerte de abstracción lejana de la práctica.

Creador

Ahkab (actualmente en la versión 0.18 2015) fue creado por Giuseppe Venturini[^9] un entusiasta de la electrónica y programación, Ingeniero y Licenciado y actualmente desarrollando su doctorado. A su vez, su creación tuvo contribuidores tales como Ian Daniher [^10], Rob Crowther[^11].

[^9]: http://ggventurini.io/

[^10]: https://github.com/itdaniher

[^11]: https://github.com/weilawei

Licencia

GNU General Public License[^12]

[^12]: https://ahkab.readthedocs.io/en/latest/misc/COPYING.html

Versiones Disponibles

Estas son las versiones “oficiales” que hay disponibles en github, los reléase notes[^13] están disponibles en la página **del** autor

* 0.18
* 0.17
* 0.16
* 0.15
* 0.14
* 0.13
* 0.12
* 0.11
* 0.10

[^13]: https://github.com/ahkab/ahkab/tags

Algunas Simulaciones Disponibles

* Numérica (.op - Punto de operación)
* Análisis transitorio (.tran)
* Análisis AC - Corriente Alterna
* Análisis PZ - Polo Cero
* Análisis periódico de estado estacionario

Dependencias

Algunas de sus dependencias son:

```
*   Dependencias Core
**  Python 2: Versiones a partir de 2.6
**  Python 3: Versiones a partir de 3.3
*   Dependencias para cálculos numéricos
**  Numpy: Versiones a partir de 1.7
**  Scipy: Versiones a partir 0.14
*   Análisis y Simulación Simbólico
**  Sympy: Versiones a partir 0.7.6
*   Dependencias para Graficas
**  Matplotlib: Versiones a partir 1.1.1
**  Tabulate: Versiones a partir 0.7.3
*   Dependencias para ejecución de Pruebas
**  Nose: no especificado
```

```
 # Trabajo práctico
 Muy bien, ahora toca definir circuitos y ejecutar simulaciones
sobre los mismos gracias a Ahkab.
 ## Instalación de bibliotecas necesarias
 Si estás utilizando Anaconda, asegúrate de tener su entorno
activado:

 ```cmd
 C:\> conda activate base (en el caso de Windows)
 ```
 ó

 ```bash
 $ source /usr/local/Caskroom/miniconda/base/bin/activate (en el
caso de macOS)
 ```

 En el caso de Windows tienes que tener en el PATH el directorio
donde se encuentre el comando `conda` (visita la sección de
[Environment Variables]
(https://superuser.com/questions/949560/how-do-i-set-system-
environment-variables-in-windows-10) del [Panel de Control]
(https://www.digitalcitizen.life/8-ways-start-control-panel-
windows-10)). Si has instalado Anaconda con [esta opción]
(https://docs.anaconda.com/_images/win-install-options.png)
marcada, ya no tienes que preocuparte por ello.

 Ahora ya puedes instalar Ahkab:

 ```
 (base) $ pip install ahkab
 ```

 Como siempre, una vez instalado cualquier framework para Python,
ya lo podemos utilizar, tanto desde el [REPL]
(https://en.wikipedia.org/wiki/Read-eval-print_loop) como desde
```

```
un entorno Jupyter (Jupyter, [Jupyterlab]
(http://jupyterlab.readthedocs.io/en/stable/), VS Code o
nteract). Recuerda que para usar el kernel Python (que viene con
Anaconda) desde nteract debes seguir las instrucciones que se
indican en su [documentación oficial]
(https://nteract.io/kernels).
```

[…]
```
import pylab as plt
import ahkab
```

También vamos a importar Sympy para hacer algún cálculo más *manual* más adelante:

[…]
```
import sympy.physics.units as u
from sympy.physics.units import Dimension
from sympy import *
from sympy.physics.units import convert_to
```

> **Pregunta:** ¿Qué es y para qué sirve PyLab?

PyLab es un conjunto de librerías o grupo de varias librerías entre las que se incluyen numpy, scipy, sympy, pandas, matplotlib e ipython que permite emular y utilizar Python como si estuviésemos en entornos Matlab y así convertir nuestros .py y tratarlos como scripts de Matlab. Ademas está comprobado, hablando en términos de performance, que PyLab ofrece mejores rendimientos en el calculo que MatLab en casi todas las operaciones por no mencionar el nivel de portabilidad ya que tiene también la ventaja de servir para Linux y para Windows

Circuitos sencillos para trabjar con la ley de Ohm:

La *mal llamada* ley de Ohm reza que el voltaje (la *energía por unidad de carga*) que se disipa en un tramo de un circuito eléctrico es equivalente a la intensidad (I) de la corriente (es decir, cuántos electrones circulan por unidad de tiempo) por la resistencia del material (R) en el que está desplazándose dicha corriente. Matemáticamente:

$$V = I \cdot R$$

> **Pregunta:** comprueba que la ecuación anterior está ajustada a nivel dimensional, es decir, que la naturaleza de lo que está a ambos lados del signo igual es la misma. Realiza este ejercicio con LaTeX en una celda Markdown.

Antes de comenzar debemos definir algunos conceptos:

- I - La cantidad de electricidad que pasa por un conductor en un segundo se llama intensidad de la corriente y se mide en AMPERIOS (A).
- R - La dificultad que ofrece el conductor al paso de una corriente eléctrica se llama resistencia eléctrica y semide en OHMIOS (Ω).
- V - A la diferencia de potencial entre dos puntos se le llama tensión o voltaje y se mide en VOLTIOS (V).

Ahora bien, la Ley de Ohm nos dice que, la intensidad es directamente proporcional a la tensión o voltaje e inversamente proporcional a la resistencia. Es decir que la intensidad crece cuando aumenta la tensión y disminuye cuando crece la resistencia.

Resistencia Eléctrica en terminos de Voltaje

Una diferencia de potencial

$$\Delta V = V_b - V_a$$

mantenida a través del conductor establece un campo eléctrico E y este campo produce una corriente I que es proporcional a la diferencia de potencial.

Si el campo se considera uniforme, la diferencia de potencial

$$\Delta V$$

se puede relacionar con el campo eléctrico E de la forma:

$$\Delta V = El$$

Por tanto, la magnitud de la densidad de corriente en el cable J se puede expresar como:

$$J = \sigma E = (1/\rho) \cdot E = (1/\rho) \cdot \Delta V/l$$

Puesto que J = I / A , la diferencia de potencial puede escribirse como:

$$\Delta V = \rho \cdot l \cdot J = \left(\frac{\rho \cdot l}{A} \right) \cdot I = RI$$

Como se puede observar en la última fórmula, la cantidad

$$R = \left(\frac{\rho \cdot l}{A} \right)$$

se denomina resistencia R del conductor. La resistencia es la razón entre la diferencia de potencial aplicada a un conductor

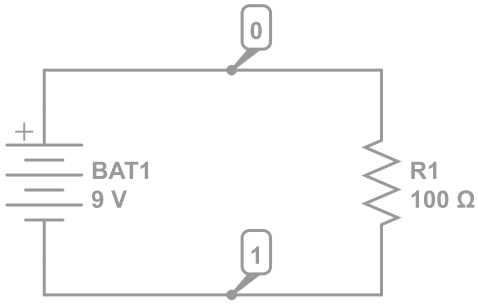
$$\Delta V$$

y la corriente que pasa por el mismo I :

$$R = \frac{V}{I}$$

Datos obtenidos de:
http://wikifisica.etsit.upm.es/index.php/LA_LEY_de_OHM_y_RESISTENCIA_ELECTRICA_DEFINITIVO

Comencemos con el circuito más sencillo posible de todos:



Vamos a escribir su contenido (componentes o *netlist*) en disco con el nombre `circuito sencillo.sp`. Esto lo podemos lograr directamente y en tiempo real desde una celda de Jupyter gracias a los *comandos mágicos* de este entorno de programación literaria. En concreto vamos a utilizar `%%writefile` que guarda los contenidos de una celda como un fichero.

[...]
```
%%writefile "circuito sencillo.sp"
* Este es un circuito sencillo
r1 1 0 100
v1 0 1 type=vdc vdc=9
.op
.dc v1 start=0 stop=9 step=1
.end
```

Ahora vamos a leer su descripción con Ahkab, interpretar y ejecutar las simulaciones que en él estén descritas.

[...]
```
circuito_y_análisis =
ahkab.netlist_parser.parse_circuit('circuito sencillo.sp')
```

Separamos la información del netlist (componentes) de los análisis (uno de tipo `op` y otro de tipo `dc`):

[...]
```
circuito = circuito_y_análisis[0]
análisis_en_netlist = circuito_y_análisis[1]
lista_de_análisis = ahkab.netlist_parser.parse_analysis(circuito,
análisis_en_netlist)
print(lista_de_análisis)
```

> **Pregunta:** ¿qué tipo de estructura de Python es `lista_de_análisis`?

lista_de_análisis, es una variable del tipo "list", es decir, una lista ordenada y editable de elementos de cualquier tipo. La misma permite iterar sobre ella y ser indexada, pero, a diferencia del Array, es menos performante para almacenar gran cantidad de datos (ya que la forma de procesamiento y asignacion de memoria es diferente) y ademas, un Array, permite mayor cantidad de operaciones aritmeticas sobre el, por ejemplo: dividir un array en 3 y multiplicar las tuplas resultantes por 2.

Las simulaciones que implican listas de datos (`.dc`, `.tran`, etc.) necesitan de un fichero temporal (`outfile`) donde almacenar los resultados. Para ello tenemos que definir la propiedad `outfile`.

```
[…]  lista_de_análisis[1]['outfile'] = "C:\simulación dc.tsv"
```

> **Pregunta:** escribe el código Python necesario para identificar qué análisis de `lista_de_análisis` son de tipo `dc` ó `tran` y sólo añadir la propiedad `outfile` en estos casos. Aquí tenéis un post de Stackoverflow con algo de ayuda. Un poco más de ayuda: el siguiente código (sí, una única línea) devuelve el índice de la simulación que es de tipo `dc`. Para simplificar un poco el ejercicio, suponed que, como máximo, habrá un análisis de tipo `tran` y/o `dc`.

```
[…]  for element in lista_de_análisis:
        elementValues = element.values()
        if "tran" in elementValues and "outfile" not in
    elementValues:
            element['outfile'] = "C:\simulación tran.tsv"
        elif "dc" in elementValues and "outfile" not in
    elementValues:
            element['outfile'] = "C:\simulación dc.tsv"
```

Una vez que ya hemos separado netlists de simulaciones, ahora ejecutamos las segundas (¡todas a la vez!) gracias al método `.run` de Ahkab:

```
[…]  resultados = ahkab.run(circuito, lista_de_análisis)
```

Resultados de la simulación `.dc`

Imprimimos información sobre la simulación de tipo `.dc`:

```
print(resultados['dc'])
```

Veamos qué variables podemos dibujar para el caso del análisis `dc`.

```
print(resultados['dc'].keys())
```

Y ahora graficamos el resultado del análisis anterior. Concretamente vamos a representar el voltaje en el borne 1 (`V1`) con respecto a la intensidad del circuito (`I(V1)`).

```
figura = plt.figure()
plt.title("Prueba DC")
plt.ylabel('Intensidad (A)')
plt.xlabel('Voltaje (V)')
plt.plot(resultados['dc']['V1'], resultados['dc']['I(V1)'],
label="Voltaje (V1)")
print(resultados['op'])
```

> **Pregunta:** comenta la gráfica anterior… ¿qué estamos viendo exactamente? Etiqueta los ejes de la misma convenientemente. Así como ningún número puede *viajar* solo sin hacer referencia a su naturaleza, ninguna gráfica puede estar sin sus ejes convenientemente etiquetados. Algo de ayuda. ¿Qué biblioteca estamos usando para graficar? Una pista.

Estamos viendo la evolucion o los valores de potencial en los dos bornes o puntos de conexiòn cuando el circuito llega al punto de estabilizaciòn o equilibrio siendo estos puntos: V0 y V1. Es decir, se ve graficamente como se inicia en un valor de voltaje existente pero el mismo disminuye al pasar por la resistencia, en otras palabras podriamos decir que la resistencia le "quita" potencial al circuito.

La Libreria que estamos usando se llama matplotlib, una suerte librería de trazado utilizada para gráficos 2D en Python, es muy flexible y tiene muchos valores predeterminados incorporados que ayuda enormemente a la hora de realizar graficas en python cuya referencia puede encontrarse en (https://matplotlib.org/).

Resultados de la simulación .op El método .results nos devuelve un diccionario con los resultados de la simulación.

```
[...]  print(resultados['op'])
```

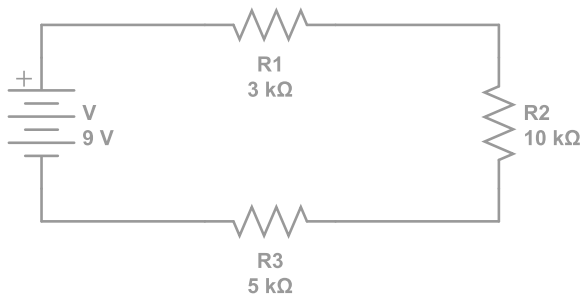
> **Pregunta:** justifica el sencillo resultado anterior (análisis op). Repite el cálculo con Sympy, atendiendo con mimo a las unidades y al formateo de los resultados (tal y como hemos visto en muchos otros notebooks en clase).

Como vemos si analizamos el circuito y realizamos el calculo con Sympy, aplicando la falsamente llamada Ley de Ohm, podemos verificar que la intensidad de corriente por su resistencia coincide con el voltaje del circuito siendo este 9V segun la definicion del mismo.

```
[...]  r1 = 100*u.ohms
       print("Ohm(R1): 100")
       print("I(V1):" + str(resultados['op']['I(V1)'][0][0]))
       intensidad_ahkab = resultados['op']['I(V1)'][0][0]*u.ampere
       print('Aplicamos la falsamente Ley de Ohm: V = I * R')
       v1 = convert_to(intensidad_ahkab*r1, [u.volt])
       pprint(v1)
```

Análisis de circuito con resistencias en serie

Vamos a resolver (en punto de operación) el siguiente circuito:



Al igual que antes, grabamos el netlist en disco desde Jupyter gracias a la *palabra mágica* `%writefile`.

```
[...]  %%writefile "resistencias en serie.net"
       * circuito con tres resistencias en serie
```

```
v1 1 0 type=vdc vdc=9
R1 1 2 3k
R2 2 3 10k
R3 3 0 5k
* análisis del circuito
.op
.end
```

```
[…]  circuito_y_análisis =
ahkab.netlist_parser.parse_circuit('resistencias en serie.net')
circuito = circuito_y_análisis[0]
análisis_en_netlist = circuito_y_análisis[1]
lista_de_análisis = ahkab.netlist_parser.parse_analysis(circuito,
análisis_en_netlist)
resultados = ahkab.run(circuito, lista_de_análisis)
```

Imprimos los resultados del análisis `.op`:

```
[…]  print(resultados['op'])
```

Los cantidades `V1`, `V2` y `V3` hacen referencia a los distintos valores del potencial que se ha perdido en cada uno de los bornes que has elegido para describir el netlist (`1`, `2`, etc.). Por ejemplo, podemos calcular el *potencial consumido* por la resistencia `R1` y verás que coincide con el del punto `V2` devuelto por Ahkab.

```
[…]  r1 = 3E3*u.ohms
r2 = 10E3*u.ohms
r3 = 5E3*u.ohms
intensidad_ahkab = resultados['op']['I(V1)'][0][0]*u.ampere
v2r1 = convert_to(intensidad_ahkab*r1, [u.volt])
v3r2 = convert_to(intensidad_ahkab*(r2), [u.volt])
v4r3 = convert_to(intensidad_ahkab*(r3), [u.volt])
pprint(v2r1)
pprint(v3r2)
pprint(v4r3)
```

> **Pregunta**: reproduce el resto de los valores anteriores de manera *manual* mediante Sympy (es decir, aplicando la ley de Ohm, pero con tun *toque computacional*). Te pongo aquí un ejemplo del que puedes partir… En él sólo calculo la corriente que circula por el circuito (sí, justo la que antes Ahkab ha devuelto de manera automática). Para ello necesito previamente computar la resistencia total (`r_total`). Faltarían el resto de resultados y

> convertirlos a unidades más *vistosas* (mediante la orden `convert_to` y `.n()`).

[…]
```
v1 = 9*u.volts
r1 = 3E3*u.ohms
r2 = 10E3*u.ohms
r3 = 5E3*u.ohms
r_total = r1 + r2 + r3

intensidad = u.Quantity('i')
intensidad.set_dimension(u.current)

ley_ohm = Eq(v1, intensidad*r_total)

solucion_para_intensidad = solve(ley_ohm, intensidad)

iv1 = convert_to(solucion_para_intensidad[0], [u.ampere]).n(2)

intensidad_ahkab = resultados['op']['I(V1)'][0][0]*u.ampere

vv2 = convert_to(intensidad_ahkab*r1, [u.volt])
vv3 = convert_to(intensidad_ahkab*(r1 + r2), [u.volt])
vv4 = convert_to(intensidad_ahkab*r_total, [u.volt])

print('Resistencia Total del Circuito: ' + str(r_total))
print('V(V0) - Carece de Sentido realizar un calculo ya que es
igual al voltaje de la fuente porque no existe ningun componente
mas que la propia perdida (fuga) del material del cableado: ' +
str(v1))
print('V(V1):' + str(vv2))
print('V(V2):' + str(vv3))
print('V(V4):' + str(vv4))
pprint('I(V1):' + str(iv1))
```

[…]
```
> **Pregunta**: Demuestra que se cumple la Ley de Kirchhoff de la
energía en un circuito, es decir, que la suma de la energía
suministrada por las fuentes (pilas) es igual a la consumida por
las resistencias. Realiza la operación con Sympy.

$$
\sum_i^N V_{\text{fuentes}} = \sum_j^M V_{\text{consumido en
resistencias}}
$$

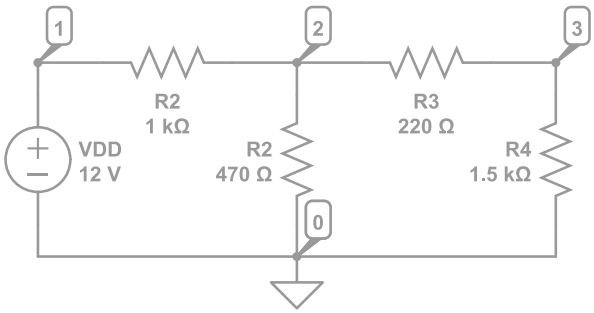
Ten en cuenta que en este caso sólo hay una fuente.
```

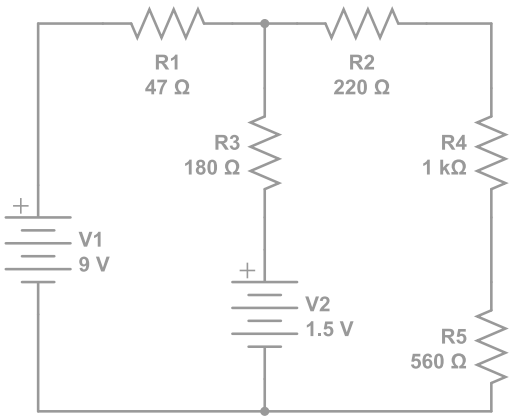
```
[...]  v2r1 = convert_to(intensidad_ahkab*r1, [u.volt])
       v3r2 = convert_to(intensidad_ahkab*(r2), [u.volt])
       v4r3 = convert_to(intensidad_ahkab*(r3), [u.volt])
       print('Si sumamos las energias consumidas por cada una de las
       resistencias, siendo estas v2R1, v3R2, V4R3, tenemos:'+ str(v2r1
       + v3r2 + v4r3))
       print('Siendo el voltaje de la fuente de alimentacion V1 igual
       a:' + str(v1))
       print('Se comprueba que la suma de las corrientes entrantes es
       igual a la suma de las corrientes consumidas por las
       resistencias.')
```

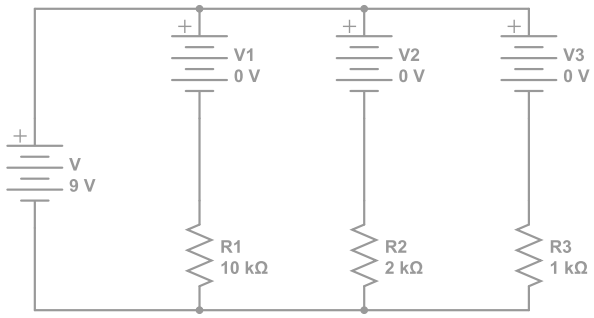
Análisis `.op` de circuitos con resistencias en paralelo

Vamos a complicar un poco el trabajo añadiendo elementos en paralelo.

> **Pregunta**: realiza los análisis `.op` de los siguientes circuitos. Para ello crea un netlist separado para cada uno donde queden correctamente descritos junto con la simulación (`.op`). Comenta los resultados que devuelve Ahkab (no imprimas los resultados de las simulaciones *sin más*).







Aquí tienes el análisis del primer circuito, para que sirva de ejemplo:

```
[...]  %%writefile "resistencias en paralelo 1.cir"
       * resistencias en paralelo
       vdd 0 1 vdc=12 type=vdc
```

```
r2 1 2 1k
r3 2 3 220
r4 3 0 1.5k
r5 2 0 470
.op
.end
```

```
circuito_y_análisis =
ahkab.netlist_parser.parse_circuit('resistencias en paralelo
1.cir')
circuito = circuito_y_análisis[0]
análisis_en_netlist = circuito_y_análisis[1]
lista_de_análisis = ahkab.netlist_parser.parse_analysis(circuito,
análisis_en_netlist)
resultados = ahkab.run(circuito, lista_de_análisis)
```

Imprimimos los resultados del análisis `.op`. Como puedes comprobar, Ahkab sólo reporta la intensidad de corriente en las ramas en las que hay una pila (en este caso, la rama donde está la pila `VDD`).

```
print(resultados['op'])
```

```
%%writefile "resistencias en paralelo 2.cir"
* resistencias en paralelo 2
vdd1 1 0 vdc=9 type=vdc
vdd2 4 0 vdc=1.5 type=vdc
r1 1 2 47
r2 2 3 220
r3 2 4 180
r4 3 5 1k
r5 5 0 560
.op
.end
```

```
circuito_y_análisis_2 =
ahkab.netlist_parser.parse_circuit('resistencias en paralelo
2.cir')
circuito_2 = circuito_y_análisis_2[0]
análisis_en_netlist_2 = circuito_y_análisis_2[1]
lista_de_análisis_2 =
ahkab.netlist_parser.parse_analysis(circuito_2,
análisis_en_netlist_2)
resultados_2 = ahkab.run(circuito_2, lista_de_análisis_2)
```

```
print(resultados_2['op'])
```

```
%%writefile "resistencias en paralelo 3.cir"
* resistencias en paralelo 3
vdd 1 0 vdc=9 type=vdc
vdd1 1 4 vdc=0 type=vdc
vdd2 2 5 vdc=0 type=vdc
vdd3 3 6 vdc=0 type=vdc
r1 4 0 10K
r2 5 0 2K
r3 6 0 1k
.op
.end
```

```
circuito_y_análisis_3 =
ahkab.netlist_parser.parse_circuit('resistencias en paralelo
3.cir')
circuito_3 = circuito_y_análisis_3[0]
análisis_en_netlist_3 = circuito_y_análisis_3[1]
lista_de_análisis_3 =
ahkab.netlist_parser.parse_analysis(circuito_3,
análisis_en_netlist_3)
resultados_3 = ahkab.run(circuito_3, lista_de_análisis_3)
```

```
print(resultados_3['op'])
```

> **Pregunta:** inserta dos *pilas virtuales* de 0 voltios en el resto de ramas del circuito (`Vdummy1` en la rama donde está `R5` y `Vdummy2` en la rama donde está `R3` y `R4`) para que Ahkab nos imprima también la corriente en las mismas. Es muy parecido al tercer circuito que tienes que resolver, donde `V1`, `V2` y `V3` tienen cero voltios. Estas *pilas nulas* son, a todos los efectos, *simples cables*. Una vez que ya tienes las corrientes en todas las ramas, comprueba que se cumple la Ley de Kirchhoff para las corrientes:

$$I_{\text{entrante}} = \sum_{i}^{N} I_{\text{salientes}}$$

COMENTARIO Cuando indica "(`Vdummy1` en la rama donde está `R5` y `Vdummy2` en la rama donde está `R3` y `R4`)" por ende solo aplica para el segundo circuito ya que el resto no tiene R5 y si bien las fuentes con 0 voltios son simples cables no es lo mismo medir un valor previo a una resistencia que posterior, por ende no me queda claro, siendo muy probable mi poco uso de SPICE en toda mi vida informática.

Repite lo mismo para los otros dos circuitos. Realiza además los cálculos con Sympy (recalcula los mismos voltajes que devuelve Ahkab a partir de la corriente que sí te devuelve

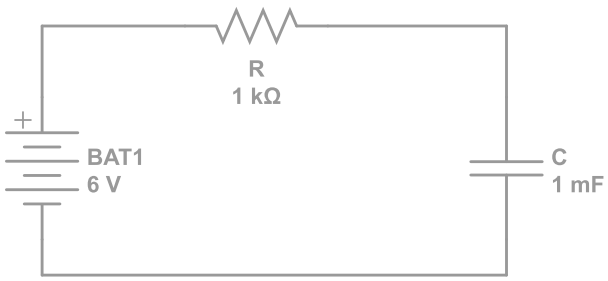
la simulación) y cuidando de no olvidar las unidades. Recuerda que el objeto `resultados` alberga toda la información que necesitas de manera indexada. Ya han aparecido un ejemplo más arriba. Es decir: no *copies* los números *a mano*, trabaja de manera informáticamente elegante (usando la variable `resultados`).

COMENTARIO Cuando se refiere a "Repite lo mismo para los otros dos circuitos" ¿puntualmente a que parte desea que pongamos las fuentes dummy?. Por mi parte no logre entender el enunciado.

Circuitos en DC que evolucionan con el tiempo

Carga de un condensador

Vamos a ver qué le pasa a un circuito de corriente continua cuando tiene un condensador en serie.



Al igual que antes, primero guardamos el circuito en un netlist externo:

```
[…]  %%writefile "condensador en continua.ckt"
     * Carga condensador
     v1 0 1 type=vdc vdc=6
     r1 1 2 1k
     c1 2 0 1m ic=0
     .op
     .tran tstep=0.1 tstop=8 uic=0
     .end
```

> **Pregunta:** ¿qué significa el parámetro `ic=0`? ¿qué perseguimos con un análisis de tipo `.tran`?

Determina el valor de voltaje inicial, en este caso determina que el condensador inicialmente tendra una carga igual a 0.

Leamos el circuito:

```
[...]  circuito_y_análisis =
       ahkab.netlist_parser.parse_circuit("condensador en continua.ckt")
```

Separamos el netlist de los análisis y asignamos un fichero de almacenamiento de datos (`outfile`):

```
[...]  circuito = circuito_y_análisis[0]
       análisis_en_netlist = circuito_y_análisis[1]
       lista_de_análisis = ahkab.netlist_parser.parse_analysis(circuito,
       análisis_en_netlist)
       lista_de_análisis[1]['outfile'] = "simulación tran.tsv"
```

Ejecutamos la simulación:

```
[...]  resultados = ahkab.run(circuito, lista_de_análisis)

       print('Anàlsis OP.')
       print(resultados['op'])

       print('Anàlsis Tran.')
       print(resultados['tran'])
```

Dibujamos la gráfica de carga del condensador con el tiempo, centrándonos en la intensidad que circula por la pila.

```
[...]  figura = plt.figure()
       plt.title("Carga de un condensador")
       plt.ylabel('Corriente de Carga (V)')
       plt.xlabel('Tiempo (T)')
       plt.plot(resultados['tran']['T'], resultados['tran']['I(V1)'],
       label="Una etiqueta")

       figura_v1 = plt.figure()
       plt.title("Gráfica con el voltaje en el borne V1")
       plt.ylabel('Voltaje (V)')
       plt.xlabel('Tiempo (T)')
```

```
plt.plot(resultados['tran']['T'], resultados['tran']['V1'],
label="Una etiqueta")

figura_v2 = plt.figure()
plt.title("Gráfica con el voltaje en el borne V2")
plt.ylabel('Voltaje (V)')
plt.xlabel('Tiempo (T)')
plt.plot(resultados['tran']['T'], resultados['tran']['V2'],
label="Una etiqueta")
```

> **Pregunta:** Etiqueta los ejes convenientemente y comenta la gráfica. Dibuja otra gráfica con el voltaje en el borne `V1`. ¿Por qué son *opuestas*? ¿Qué le ocurre al voltaje a medida que evoluciona el circuito en el tiempo? Dibuja las gráficas en un formato estándar de representación vectorial (SVG, por ejemplo). Algo de ayuda aquí. ¿Qué valores devuelve el análisis de tipo `.op`? Justifícalo.

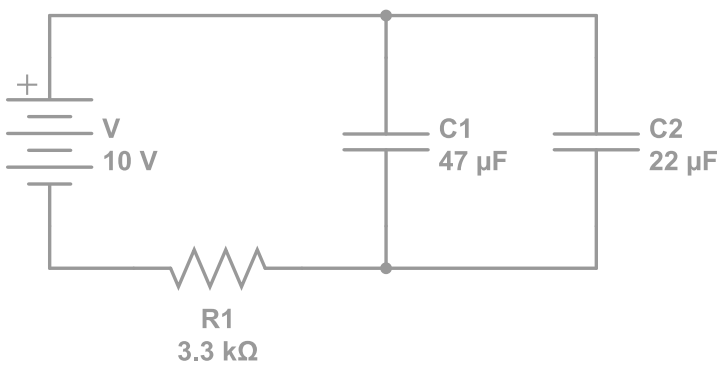
COMENTARIO Por alguna razon que desconosco, el valor en el borne V1 es constante

Las gráficas son opuestas ya que al conectar una batería con una resistencia y un condensador en serie, la corriente inicial es alta ya que la batería debe "mover" la carga entre las placas del condensador luego la carga de corriente alcanza asintóticamente el valor de cero a medida que el condensador se llena con la "carga" de la batería. Vale destacar que la tasa de carga se describe típicamente en función de la constante de tiempo y en este caso se justifica su análisis en un periodo de tiempo mediante la utilización del análisis ".tran"

```
[...]  print(resultados['op'])
```

Carrera de condensadores

Ahora tenemos un circuito con dos condensadores en paralelo:



> **Pregunta:** Crea el netlist de este circuito e identifica qué condensador se satura primero. Dibuja la evolución de la intensidad en ambas ramas de manera simultánea. Aquí tienes un ejemplo de cómo se hace esto en Matplotlib. Recuerda que para que Ahkab nos devuelva la corriente en una rama, debe de estar presente una pila. Si es necesario, inserta pilas virtuales de valor nulo (cero voltios), tal y como hemos comentado antes. Grafica también los voltajes (en otra gráfica, pero que aparezcan juntos).

El flujo de corriente de carga en el circuito se distribuye a todos los condensadores en el circuito, la corriente total de carga es igual a la suma de todas las corrientes de carga individuales de los condensadores en el circuito y tenemos un mayor valor de capacitancia total

```
%%writefile "2 condensadores en paralelo.ckt"
* Carga condensador
v0 0 1 type=vdc vdc=10
r1 0 2 3k
c1 2 3 47u ic=0
v1dummy 3 1 type=vdc vdc=0
c2 2 4 22u ic=0
v2dummy 4 1 type=vdc vdc=0
.op
.tran tstep=0.01 tstop=1 uic=0
.end
```

```
circuito_y_análisis = ahkab.netlist_parser.parse_circuit("2
condensadores en paralelo.ckt")
```

```
circuito = circuito_y_análisis[0]
análisis_en_netlist = circuito_y_análisis[1]
lista_de_análisis = ahkab.netlist_parser.parse_analysis(circuito,
análisis_en_netlist)
lista_de_análisis[1]['outfile'] = "simulación 2 condensadores en
paralelo.tsv"
```

```
resultados = ahkab.run(circuito, lista_de_análisis)
```

```
print('Anàlsis OP.')
print(resultados['op'])

print('Anàlsis Tran.')
print('Resultados Keys')
print(resultados['tran'].keys())
print('Resultados V1')
```

```
print(resultados['tran']['V1'])
print('Resultados V2')
print(resultados['tran']['V2'])
print('Resultados V3')
print(resultados['tran']['V3'])
print('Resultados V4')
print(resultados['tran']['V4'])
```

```
figura,ax1 = plt.subplots()
plt.title("Carga de un condensador")

color = 'tab:red'
ax1.set_xlabel('Tiempo (T)')
ax1.set_ylabel('Corriente de Carga (V)', color=color)
ax1.plot(resultados['tran']['T'], resultados['tran']
['I(V1DUMMY)'], color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the
same x-axis

color = 'tab:blue'
ax2.set_ylabel('Corriente de Carga (V)', color=color)
ax2.plot(resultados['tran']['T'], resultados['tran']
['I(V2DUMMY)'], color=color)
ax2.tick_params(axis='y', labelcolor=color)

figura.tight_layout()  # otherwise the right y-label is slightly
clipped
plt.show()
```

```
figura,ax1 = plt.subplots()
plt.title("Carga de un condensador")

color = 'tab:red'
ax1.set_xlabel('Tiempo (T)')
ax1.set_ylabel('Voltaje en Borne V1 (V)', color=color)
ax1.plot(resultados['tran']['T'], resultados['tran']['V1'],
color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the
same x-axis

color = 'tab:blue'
ax2.set_ylabel('Voltaje en Borne V2 (V)', color=color)
ax2.plot(resultados['tran']['T'], resultados['tran']['V2'],
color=color)
ax2.tick_params(axis='y', labelcolor=color)
```

```
ax3 = ax1.twinx()  # instantiate a second axes that shares the
same x-axis

color = 'tab:purple'
ax3.set_ylabel('Voltaje en Borne V3 (V)', color=color)
ax3.plot(resultados['tran']['T'], resultados['tran']['V3'],
color=color)
ax3.tick_params(axis='y', labelcolor=color)

ax4 = ax1.twinx()  # instantiate a second axes that shares the
same x-axis

color = 'tab:brown'
ax4.set_ylabel('Voltaje en Borne V3 (V)', color=color)
ax4.plot(resultados['tran']['T'], resultados['tran']['V4'],
color=color)
ax4.tick_params(axis='y', labelcolor=color)

figura.tight_layout()  # otherwise the right y-label is slightly
clipped
plt.show()
```

Empty markdown cell, double click me to add content.