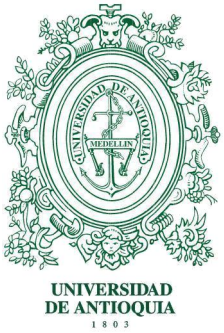
Suplemento Computacional

Física de Oscilaciones y Ondas

Sebastian Bustamante Jaramillo

macsebas33@gmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Antioquia

Índice general

Capítulo 1

Preliminares

1.1. Motivación

La física ha evolucionado hasta un estado actual donde la mayoría de cálculos teóricos necesarios para realizar investigación de frontera requieren de una gran componente computacional. Desde la corroboración entre teoría y experimento, la predicción y control de los resultados de un experimento hecho a posteriori y la recreación de condiciones imposibles de lograr experimentalmente, tales como simulaciones cosmológicas del universo a gran escala o complejos sistemas atómicos. Estos son sólo algunos ejemplos representativos del papel de la computación en la física moderna. Debido a esto, el principal objetivo del suplemento computacional es la introducción temprana en los cursos de física básica de herramientas computacionales que serán de utilidad a los estudiantes en este curso específico y durante el transcurso de sus carreras científicas.

1.2. Instalación de Paquetes

En la totalidad de esta guía será usado el lenguaje de programación *Python* como referente para todos las prácticas y ejercicios computaciones. La principal motivación de esto es su facilidad de implementación en comparación a otros lenguajes también de amplio en ciencia. Además es un lenguaje interpretado, lo que permite una depuración más sencilla por parte del estudiante, sin necesidad de usar más complicados sistemas de depuración en el caso de lenguajes compilados como C o Fortran. *Python* es un lenguaje de código abierto, lo que permite la libre distribución del paquete y evita el pago de costosas licencias de uso, además la gran mayoría de paquetes que extienden enormemente la funcionalidad de *Python* son también código abierto y de libre distribución y uso.

A pesar de que *Python* es un lenguaje multiplataforma, permitiendo correr scripts python en Linux, Windows y Mac, acá solo se indicará el método de instalación para distribuciones Linux basadas en Debian.

La última versión de *Python* de la rama 2 es 2.7.4 y de la rama 3 es la 3.3.1, debido a ligeras incompatibilidades entre ambas ramas de desarrollo, será utilizada la rama 2 en una de sus últimas versiones. En orden, para instalar *Python* en una versión Linux basta con descargarlo directamente de los repositorios oficiales[1], en el caso de una distro basada en Debian el gestor de paquetes es `apt-get`, y desde una terminal se tiene

```
\$ apt-get install python2.7
```

también puede descargarse directamente desde la página oficial del proyecto `http://python.org/`.

Una vez instalada la última versión de *Python* , es necesario instalar los siguiente paquetes para el correcto desarrollo de las aplicaciones del curso:

iPython

iPython es un shell que permite una interacción más interactiva con los scripts de python, permitiendo el resaltado de sintaxis desde consola, funciones de autocompletado y depuración de código más simple. Para su instalación basta descargarlo de los repositorios oficiales

```
\$ apt-get install ipython
```

o puede de descargarse de la página oficial `http://ipython.org/`. También puede encontrarse documentación completa y actualizada en esta página, se recomienda visitarla frecuentemente para tener las más recientes actualizaciones.

NumPy

NumPy es una librería que extiende las funciones matemáticas de *Python* , permitiendo el manejo de matrices y vectores. Es esencial para la programación científica en *Python* y puede ser instalada de los repositorios

```
\$ apt-get install python-numpy
```

[1] En la mayoría de distribuciones Linux *Python* viene precargado por defecto.

La última versión estable es la 1.6.2. En la página oficial del proyecto puede encontrarse versiones actualizadas y una amplia documentación `http://www.numpy.org/`.

SciPy

SciPy es una amplia biblioteca de algoritmos matemáticos para *Python* , esta incluye herramientas que van desde funciones especiales, integración, optimización, procesamiento de señales, análisis de Fourier, etc. Al igual que los anteriores paquetes, puede ser instalada desde los repositorios oficiales

```
\$ apt-get install python-scipy
```

Una completa documentación del paquete puede ser encontrada en `http://docs.scipy.org/doc/scipy/reference/`. La última versión estable es la 0.11.0 y puede ser encontrada en la página oficial del proyecto `http://www.scipy.org/`.

Matplotlib

Matplotlib es una completa librería con rutinas para la generación de gráficos a partir de datos. Aunque en su estado actual está enfocada principalmente a gráficos 2D, permite un amplio control sobre el formato de las gráficas generadas, dando una amplia versatilidad a los usuarios. Su instalación puede realizarse a partir de los repositorios oficiales

```
\$ apt-get install python-matplotlib
```

La última versión estable es la 1.2.1. y puede encontrarse en la página oficial del proyecto `http://matplotlib.org/`. Una amplia documentación está disponible en `http://matplotlib.org/1.2.0/contents.html`.

MayaVi2

MayaVi2 es una librería para la visualización científica en python, en especial para gráficos 3D, permitiendo funciones avanzadas como renderizado, manejo de texturas, etc. Se encuentra en los repositorios oficiales

```
\$ apt-get install mayavi2
```

La versión 2 es una versión mejorada de la original, estando más orientada a la reutilización de código. Por defecto incluye una interfaz gráfica que facilita su manejo. La página oficial del proyecto es `http://mayavi.sourceforge.net/`.

Tkinter

TKinter es una librería para la gestión gráfica de aplicaciones in *Python* y viene por defecto instalada, aún así puede ser instalada de los repositorios oficiales

```
\$ apt-get install python-tk
```

La página oficial del proyecto es `http://wiki.python.org/moin/TkInter`. Para el desarrollo de entornos gráficos existen otras llamativas alternativas como PyGTK o PyQt, pero debido a la facilidad de uso y a ser la librería estándar soportada, *TKinter* será usada en este curso.

1.3. Ejemplo de Uso

En esta sección se ilustra un ejemplo sencillo que permite al estudiante identificar la manera estándar de ejecutar códigos en *Python* , además probar los paquetes instalados.

El código de ejemplo permite graficar dos funciones diferentes en una misma ventana y además un conjunto de datos aleatorios generados en el eje Y.

```
1  #!/usr/bin/env python
2  #============================================================
3  # EJEMPLO DE USO
4  # Grafica de funciones y datos aleatorios
5  #============================================================
6  import numpy as np
7  import scipy as sp
8  import matplotlib.pylab as plt
9
10 #Funcion 1
11 def Funcion1(x):
12     f1 = np.sin(x)/( np.sqrt(1 + x**2) )
13     return f1
14
15 #Funcion 2
16 def Funcion2(x):
```

```
17      f2 = 1/(1+x)
18      return f2
19
20  #Valores de x para evaluar
21  X = np.linspace( 0, 10, 100 )
22  #Evaluacion de funcion 1
23  F1 = Funcion1(X)
24  #Evaluacion de funcion 2
25  F2 = Funcion2(X)
26
27  #Grafica funcion 1
28  plt.plot( X, F1, label='Funcion 1' )
29  #Grafica funcion 2
30  plt.plot( X, F2, label='Funcion 2' )
31
32  #Datos aleatorios eje Y
33  Yrand = sp.random.rand( 100 )
34  #Grafica datos aleatorios
35  plt.plot( X, Yrand, 'o', label='Datos' )
36
37  plt.legend()
38  plt.show()
```

El resultado obtenido es la siguiente gráfica

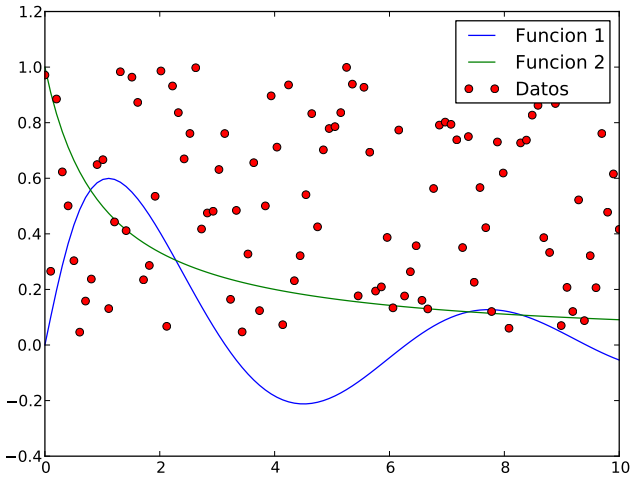


Figura 1.1: Resultado del ejemplo anterior, gráfica de dos funciones y datos aleatorios.

Para obtener el anterior script, el estudiante puede transcribirlo directamente de esta guía o puede descargarlo del repositorio oficial del curso[2] en el link `https://github.com/sbustamante/Computacional-Campos/raw/master/codigos/usage_01.py`. Una vez obtenido el archivo `usage_01.py`, abrir una terminal en la carpeta donde se ha guardado y escribir `ipython` para abrir el intérprete de *Python*

```
\$ ipython
```

Se debe obtener algo como

```
\$ ipython
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

Finalmente para ejecutar el script, usar el comando `run` seguido del nombre del código en el intérprete de código *Python*

```
In [1]: run usage_01.py
```

1.4. Consejos de Programación

En esta sección se dan algunos consejos útiles a la hora de programar. Estas buenas prácticas ayudan al programador a ser estructurado y tener un método coherente de desarrollar códigos.

- Asignar nombres a los archivos que sean coherentes con lo que realizan. Por ejemplo si está realizando una rutina que integra la trayectoria de una partícula en un campo eléctrico $\boldsymbol{E}$, un nombre adecuado para el código puede ser `trayectory_electric_field.py`. Evite usar nombres poco intuitivos y que tengan caracteres extraños, inclusive caracteres tildados como `í`, `ó`, `ñ`, etc. También evite el uso de espacios, si desea separar palabras, use el caracter _.

[2] Repositorio oficial en `https://github.com/sbustamante/Computacional-Campos`

- Una práctica casi obligatoria que debe realizar cualquier programador serio, consiste en comentar su código exhaustivamente. Los lenguajes de programación fueron inventados para facilitar la interacción entre el humano y la computadores, los compiladores e intérpretes de códigos *traducen* este lenguaje a lenguaje binario de máquina. Es por esta razón que cuando se crea un código, se debe pensar en que este sea comprensible, siempre recuerde que la máquina solo requiere código binario.

  Comente todo lo que usted crea conveniente. Es muy común que por minimizar líneas de código o simplemente por pensar que no es necesario, muchos programadores dejan de comentar su código y semanas más tarde retoman y no entienden que han hecho. La programación en equipo es cada vez más común en ciencia, existiendo grandes proyectos donde una comunidad activa está contribuyendo, comentar el código es indispensable para este tipo de actividades.

- Use el inglés siempre que pueda, para asignar nombres a sus archivos, para comentar su código, y para datos impresos en pantalla y en archivos de datos. Tenga en cuenta que en disciplinas científicas cualquier labor que sea realizada en inglés tendrá un potencial de impacto mayor que otro si se hace en otro idioma. Su código puede ser útil después a otras personas y esto puede ayudar a que usted sea reconocido en círculos académicos.

- Comparta su código, para esto puede usar páginas diseñadas para esto tales como `github.com` o `sourceforge.net`. Estas páginas además de permitir compartir su código a terceras personas, tiene potentes herramientas de control de versiones (git, svn, etc.) las cuales le ayudan a manejar su código de forma más estructurada, permitiendo el control de las diferentes versiones de un código o un paquete completo y el desarrollo entre varias personas.

- Use software open source, este tipo de software es de libre distribución y uso. Cuando se usan herramientas que necesitan licencia se está sujeto al pago de ingentes cantidades de dinero por funcionalidades que se pueden encontrar de forma gratuita. Este tipo de herramientas pagas limitan aspectos como el poder compartir códigos y resultados, por ejemplo si usted no tiene su licencia al día puede tener problemas a la hora de reportar en un artículo de investigación sus resultados y gráficas obtenidos con estos paquetes.

Capítulo 2

Oscilaciones

El concepto de oscilación es ampliamente usado en ciencia y se aplica para cualquier cantidad que presente fluctuaciones o perturbaciones en función del tiempo, ya sean periódicas o no. En este capítulo se presentan algunos ejemplos de oscilaciones para sistemas mecánicos y electromagnéticos, que a pesar de su simplicidad, permiten ahondar en los detalles físicos de este tipo de fenómenos.

La forma estándar de abordar un problema en una disciplina científica consiste en el estudio de situaciones ideales y muy particulares para llegar luego, usando refinamientos y consideraciones posteriores, a descripciones realistas y más generales. Por este motivo se comienza con demostraciones computacionales aplicadas a sistemas ideales, como el péndulo simple, el sistema masa resorte, etc. En demostraciones siguientes se abordarán aspectos cada vez más complejos de estos sistemas.

2.1. Demostración 1: Péndulo Simple Ideal

Como primer caso se aborda el péndulo simple. Este consiste en un sistema de una masa m con dimensión despreciable y bajo la acción de la gravedad $\boldsymbol{g}$, además pende de una cuerda tensa de longitud l y sujeta en un punto fijo. A partir de la figura 2.1, la ecuación de movimiento está dada por

$$m\frac{d^2\boldsymbol{r}}{dt^2} = m\boldsymbol{g} + \boldsymbol{T} \tag{2.1}$$

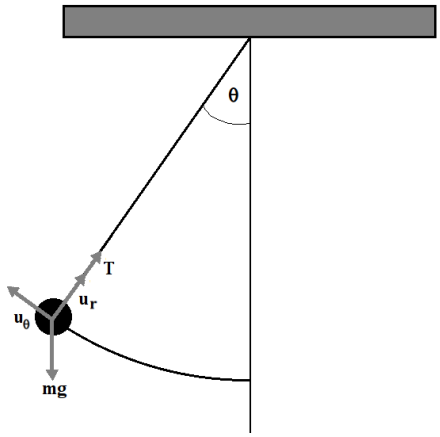


Figura 2.1: Péndulo simple bajo la acción del campo gravitacional.

En coordenadas polares se obtiene

$$ml\dot{\theta}^2 = T - mg\cos\theta \tag{2.2}$$
$$ml\ddot{\theta} = -mg\sin\theta \tag{2.3}$$

Como primera demostración computacional de este capítulo se solucionará el movimiento del péndulo simple a partir de las ecuaciones 2.2 y 2.3. Para esto se asumirá que la amplitud de oscilación del péndulo es pequeña de tal forma que $\theta \approx \sin\theta$ y $\cos\theta \approx 1$, obteniendo

$$\ddot{\theta} = -\frac{g}{l}\theta \tag{2.4}$$

Usando un ansatz de la forma $\theta(t) = e^{\lambda t}$ se llega a la solución

$$\theta(t) = \theta_0 \sin(\omega_0 t + \delta) \tag{2.5}$$

donde θ_0 y δ son la amplitud y la fase respectivamente y constituyen las condiciones iniciales. La frecuencia ω_0 está definida por

$$\omega_0 = \sqrt{\frac{g}{l}} \tag{2.6}$$

En el siguiente script de *Python* se grafica esta solución para diferentes valores de la amplitud y la fase.

```
#!/usr/bin/env python
#============================================================
# DEMOSTRACION 1
# Grafica de soluciones aproximadas del pendulo simple
#============================================================
import numpy as np
import matplotlib.pylab as plt

#Solucion
def Theta(t):
    theta = theta0*np.sin( omega0*t + delta )
    return theta

#Gravedad
g = 9.8
#Longitud
l = 1
#Frecuencia
omega0 = np.sqrt( g/l )
#Tiempos
tiempo = np.arange( 0, 10, 0.1 )

#SOLUCION 1
#Amplitud
theta0 = 0.05
#Fase
delta = 0.0
#Grafica
plt.plot( tiempo, Theta(tiempo), label='solucion 1' )

#SOLUCION 2
#Amplitud
theta0 = 0.05
#Fase
delta = np.pi
#Grafica
plt.plot( tiempo, Theta(tiempo), label='solucion 2' )

#SOLUCION 3
#Amplitud
theta0 = 0.1
#Fase
```

```
43 delta = 0.0
44 #Grafica
45 plt.plot( tiempo, Theta(tiempo), label='solucion 3' )
46
47 plt.legend()
48 plt.show()
```

El resultado que se obtiene es

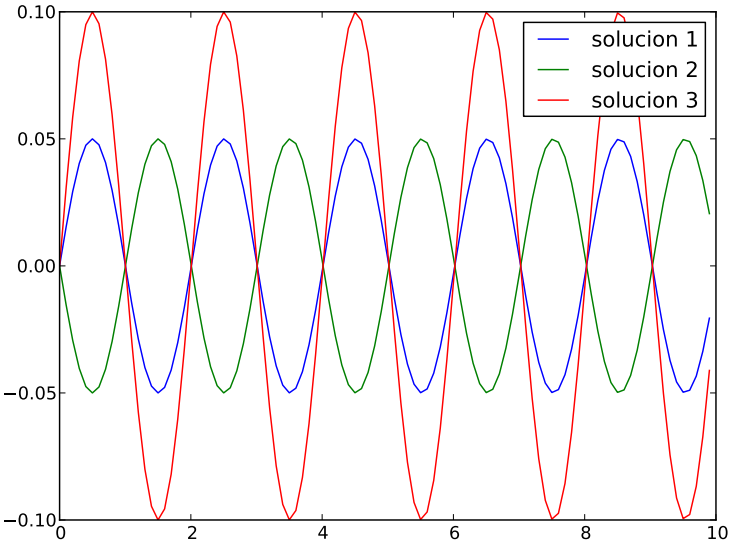


Figura 2.2: Soluciones aproximadas obtenidas para el péndulo simple.

En lo siguiente se explica cada porción de código.

```
import numpy as np
import matplotlib.pylab as plt
```

Esto corresponde al cargado de las librerías *NumPy* , bajo el alias de np y *Matplotlib* bajo el alias de plt, ambas son necesarias para el desarrollo de todo el código.

```
#Solucion
def Theta(t):
    theta = theta0*np.sin( omega0*t + delta )
    return theta
```

En este parte se define la solución obtenida en la ecuación 2.5 para cualquier tiempo dado. Las funciones matemáticas estándar, tales como exponencial, logaritmo, trigonométricas, hiperbólicas, etc. Son accedidas desde el módulo *NumPy* como `np.sin`, `np.sinh`, `np.exp`, `np.log`, etc.

```
#Gravedad
g = 9.8
#Longitud
l = 1
#Frecuencia
omega0 = np.sqrt( g/l )
#Tiempos
tiempo = np.arange( 0, 10, 0.1 )
```

Se define la gravedad, la longitud de la cuerda (en metros), la frecuencia de oscilación y finalmente, usando la función `arange` de la librería *NumPy* , se construye un arreglo de tiempos donde es evaluada la solución, de 0 a 10 segundos con saltos de 0.1.

```
#SOLUCION 1
#Amplitud
theta0 = 0.05
#Fase
delta = 0.0
#Grafica
plt.plot( tiempo, Theta(tiempo), label='solucion 1' )
```

La primera solución es obtenida para una amplitud de $\theta_0 = 0{,}05$ radianes y una fase $\delta = 0$. La última línea corresponde a la construcción de la gráfica de la solución. Para esto se usa la función `plot` de la librería *Matplotlib* . El primer argumento corresponde a los datos asociados al eje x, en este caso `tiempo`, mientras el segundo argumento son los datos asociados al eje y, en este caso la solución evaluada en el tiempo, es decir `Theta(tiempo)` . El argumento `label` indica el nombre que tendrá la solución en la gráfica final, esto se denomina etiqueta de la función.

```
plt.legend()
plt.show()
```

Finalmente se termina el script con la función `legend`, la cual muestra en pantalla las etiquetas puestas a cada gráfica. Y la función `show` que muestra en pantalla todas las soluciones.

2.2. Demostración 2: Solución Exacta del Péndulo

La solución propuesta en la demostración anterior está basada en la aproximación de pequeñas oscilaciones, para la cual $\sin\theta \approx \theta$ y $\cos\theta \approx 1$, aún así, a medida que el movimiento sea de mayor amplitud, esta aproximación deja de ser válida y se la solución general debe obtenerse directamente de las ecuaciones 2.2 y 2.3

Una forma conveniente de reescribir las ecuaciones de movimiento es derivando 2.2 respecto al tiempo e introduciendo la ecuación 2.3

$$\begin{aligned} \dot{T} &= \dot{\theta}\left(2ml\ddot{\theta} - mg\sin\theta\right) \\ \dot{T} &= -\dot{\theta}\left(2mg\sin\theta + mg\sin\theta\right) \\ \dot{T} &= -3\dot{\theta}mg\sin\theta \end{aligned} \tag{2.7}$$

Definiendo la velocidad angular $\omega = \dot{\theta}$, el sistema de ecuaciones originales junto con la ecuación 2.7 queda

$$\dot{\theta} = \omega \tag{2.8}$$

$$\dot{\omega} = -\frac{g}{l}\sin\theta \tag{2.9}$$

$$\dot{T} = -3\omega mg\sin\theta \tag{2.10}$$

Esta forma se denomina sistema de ecuaciones diferenciales lineales y es esencial para la solución exacta a partir de algoritmos de integración numérica.

Las condiciones iniciales para la solución aproximada son completamente determinadas a partir de la fase δ y la amplitud θ_0. En el caso de la solución exacta es necesario suministrar tres condiciones para cada una de las variables de las ecuaciones 2.8 - 2.10, es decir, la posición angular inicial del péndulo θ_{t_0}, la velocidad angular inicial ω_{t_0} y la tensión inicial T_{t_0}. Aún así, la tensión inicial depende de las otras dos condiciones a través de la ecuación 2.2

$$T_{t_0} = ml\omega_{t_0}^2 + mg\cos\theta_{t_0} \tag{2.11}$$

En esta demostración se calcularán ambas soluciones para un péndulo con condiciones iniciales fijadas. La solución numérica se realiza a partir del la rutina de integración numérica `odeint` del módulo `integrate` del paquete *SciPy* .

```
#!/usr/bin/env python
#============================================================
# DEMOSTRACION 2
# Comparacion de solucion completa y aproximada para el
# pendulo simple
#============================================================
import numpy as np
import matplotlib.pylab as plt
import scipy.integrate as integ

#Solucion aproximada
def Theta(t):
    theta = theta0*np.sin( omega0*t + delta )
    return theta

#Ecuaciones de movimiento
def dF(Y, t):
    #Valor anterior de theta
    theta = Y[0]
    #Valor anterior de omega
    omega = Y[1]
    #Valor anterior de la tension
    tension = Y[2]
    #Derivada de theta
    Dtheta = omega
    #Derivada de omega
    Domega = -g*np.sin( theta )/l
    #Derivada de la tension
    Dtension = -3*omega*m*g*np.sin( theta )
    return (Dtheta, Domega, Dtension)

#Gravedad
g = 9.8
#Longitud
l = 1
#Masa
m = 1
#Frecuencia
omega0 = np.sqrt( g/l )
#Tiempos
tiempo = np.arange( 0, 10, 0.01 )

```

```
#SOLUCION APROXIMADA
#Amplitud
theta0 = np.pi/4
#Fase
delta = np.pi/2.
#Grafica
plt.plot(tiempo, Theta(tiempo), label='solucion aproximada')

#SOLUCION NUMERICA
#Angulo inicial
theta_t0 = np.pi/4
#Velocidad angular inicial
omega_t0 = 0.0
#Tension inicial
tension_t0 = m*l*omega_t0**2 + m*g*np.cos( theta_t0 )
#Condiciones iniciales
cond_ini = ( theta_t0, omega_t0, tension_t0 )
#Solucion numerica
theta_t, omega_t, tension_t = np.transpose(
integ.odeint( dF, cond_ini, tiempo ) )
#Grafica
plt.plot( tiempo, theta_t, label='solucion numerica' )

plt.legend()
plt.show()
```

Para ambas soluciones se ha asumido una amplitud de $\theta_0 = 45^o = \pi/4$ y las condiciones iniciales son definidas de tal forma que el péndulo sea soltado desde su máxima amplitud. En el caso de la solución aproximada, esto implica una amplitud $\theta_0 = \pi/4$ y una fase de $\delta = \pi/2$. Para la solución aproximada, las condiciones equivalentes son $\theta_{t_0} = \pi/4$ y $\omega_{t_0} = 0$. El resultado de la integración para 10 segundos es mostrado en la figura 2.3.

Cambiando la amplitud inicial a un valor $\theta = 10^o \approx 0{,}17$ ambas soluciones son muy similares, indicando el rango de la validez de la aproximación inicial. El resultado para 10 segundos es mostrado en la figura 2.4.

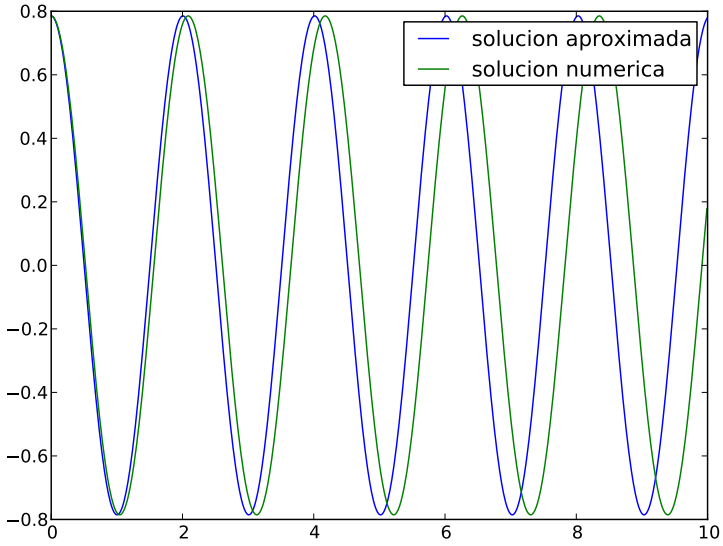


Figura 2.3: Comparación entre la solución exacta numérica y la solución aproximada. Debido a la mayor amplitud de oscilación, la aproximación de pequeñas oscilaciones deja de ser válida y difiere considerablemente de la solución real.

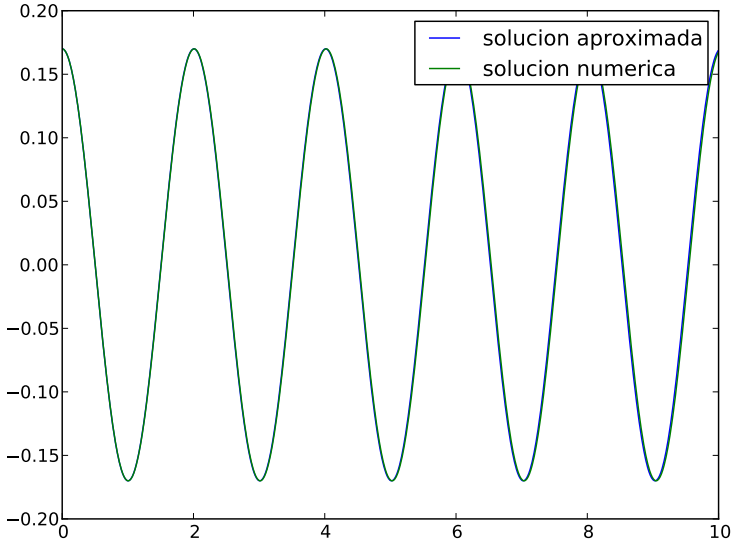


Figura 2.4: Comparación entre la solución exacta numérica y la solución aproximada. Para pequeñas amplitudes ambas soluciones son muy similares.

A continuación se explican las nuevas partes introducidas en el código en comparación con la demostración 1.

```
import scipy.integrate as integ
```

Esto corresponde al cargado del módulo `integrate` de *SciPy* bajo el alias de `integ` para las funcionalidades de integración numérica.

```
#Ecuaciones de movimiento
def dF(Y, t):
    #Valor anterior de theta
    theta = Y[0]
    #Valor anterior de omega
    omega = Y[1]
    #Valor anterior de la tension
    tension = Y[2]
    #Derivada de theta
    Dtheta = omega
    #Derivada de omega
    Domega = -g*np.sin( theta )/l
    #Derivada de la tension
    Dtension = -3*omega*m*g*np.sin( theta )
    return (Dtheta, Domega, Dtension)
```

Se define la función que contiene las derivadas de cada una de las variables del sistema, acorde al sistema 2.8 - 2.10. `Y` es un arreglo que contiene las variables θ, ω y T evaluadas en el tiempo anterior respectivamente y finalmente el argumento `t` es el tiempo actual.

```
#SOLUCION NUMERICA
#Angulo inicial
theta_t0 = np.pi/4
#Velocidad angular inicial
omega_t0 = 0.0
#Tension inicial
tension_t0 = m*l*omega_t0**2 + m*g*np.cos( theta_t0 )
#Condiciones iniciales
cond_ini = ( theta_t0, omega_t0, tension_t0 )
#Solucion numerica
theta_t, omega_t, tension_t = np.transpose(
integ.odeint( dF, cond_ini, tiempo ) )
#Grafica
```

```
plt.plot( tiempo, theta_t, label='solucion numerica' )
```

En esta parte se realiza la integración de la solución exacta del péndulo simple. Se dan las condiciones iniciales para el ángulo y la velocidad angular inicial, para la tensión se usa la ecuación 2.11. Luego, en un arreglo `cond_ini` se dan las tres condiciones para luego llamar la función `integ.odeint`. Esta tiene como primer argumento la función `dF` con las ecuaciones de movimiento del sistema, segundo argumento el arreglo de condiciones iniciales y como tercero el arreglo de tiempo donde se desea evaluar la solución. Finalmente se grafica la solución.

2.3. Demostración 3: Sistema de Péndulos Acoplados

En esta demostración se abordarán aspectos más complejos de computación, donde se comenzará a realizar animaciones y representaciones 3D a partir de la librería *MayaVi2* . Como tercera demostración se plantea entonces un sistema acoplado de 4 péndulos simples de diferente longitud tal como se muestra en la figura 2.5.

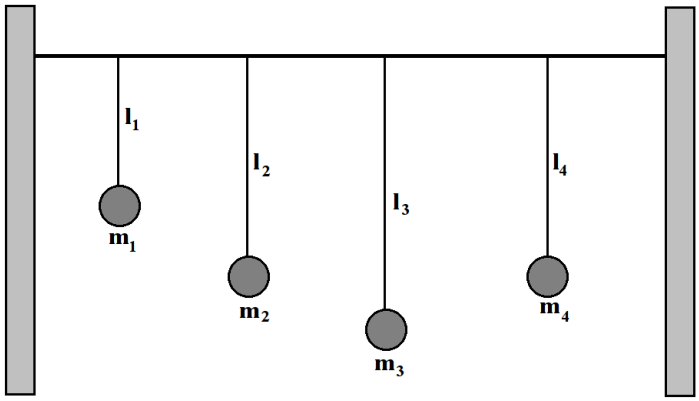


Figura 2.5: Sistema de 4 péndulos acoplados.

Todos los péndulos son de dimensión despreciable de tal forma que pueden ser considerados puntuales, además todos penden de una cuerda tensionada y no de un cuerpo rígido debido a que la cuerda actúa como agente de acoplamiento en el sistema, permitiendo el intercambio de energía entre los péndulos.
Para modelar el término de acoplamiento entre los péndulos se supondrá que cada péndulo solo interactúa de forma apreciable con sus vecinos más cercanos y que

la fuerza de interacción solo actúa en la dirección horizontal determinada por el eje *x*, siendo proporcional a la solución de los vecinos próximos. Teniendo esto en consideración, a ecuación de movimiento para el i-ésimo péndulo es

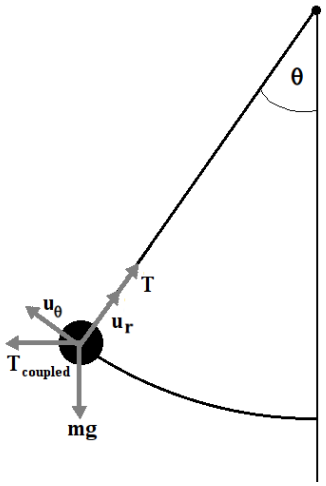


Figura 2.6: i-ésimo péndulo del sistema acoplado. La fuerza de acoplamiento actúa en el eje horizontal.

$$m\frac{d^2\boldsymbol{r}_i}{dt^2} = m\boldsymbol{g} + \boldsymbol{T}_i + \boldsymbol{T}_{c,i-1} + \boldsymbol{T}_{c,i+1} \tag{2.12}$$

donde $\boldsymbol{T}_{c,i+1}$ y $\boldsymbol{T}_{c,i-1}$ son los términos de acoplamiento asociados a los péndulos vecinos, en caso de no existir uno de los vecinos, el término es nulo. Por simplicidad y sin pérdida de generalidad se asumirán péndulos de masas iguales. Finalmente, con el objetivo de cuantificar la fuerza de acoplamiento, se usará el peso de los péndulos de tal forma que

$$\boldsymbol{T}_{c,j} = fmg\theta_j(t)\boldsymbol{i} \tag{2.13}$$

con $\theta_j(t)$ la solución del péndulo j y f un factor que indica la magnitud de la fuerza de acoplamiento en unidades mg. En principio, este factor depende de la tensión de la cuerda principal de la cual penden los péndulos, pero por simplicidad se tomará como un parámetro libre de tal forma que $f \ll 1$.

Descomponiendo las ecuaciones de movimiento en coordenadas polares del i-ésimo péndulo tal como se hizo para el péndulo simple en la demostración 1 2.1 se llega a

$$\begin{aligned} ml\dot{\theta}_i^2 &= T_i - mg\cos\theta_i - fmg\left(\theta_{i-1}+\theta_{i+1}\right)\sin\theta_i && (2.14)\\ ml\ddot{\theta}_i &= -mg\sin\theta_i + fmg\left(\theta_{i-1}+\theta_{i+1}\right)\cos\theta_i && (2.15)\end{aligned}$$

Al igual que la demostración 2 2.2, no se realizará la aproximación de pequeñas oscilaciones y las ecuaciones serán resueltas de forma numérica. Es necesario entonces llevar las ecuaciones 2.14 2.15 a una sistema de ecuaciones diferenciales lineales, para esto se deriva la ecuación 2.14 respecto al tiempo y se reemplaza el termino en $\ddot{\theta}_i$ por la ecuación 2.15.

$$\begin{aligned} \dot{T}_i &= \dot{\theta}_i\left[2ml\ddot{\theta}_i - mg\sin\theta_i + fmg\left(\theta_{i-1}+\theta_{i+1}\right)\cos\theta_i\right]\\ &\quad + fmg\left(\dot{\theta}_{i-1}+\dot{\theta}_{i+1}\right)\sin\theta_i\\ \dot{T}_i &= 3\dot{\theta}_i\left[fmg\left(\theta_{i-1}+\theta_{i+1}\right)\cos\theta_i - mg\sin\theta_i\right]\\ &\quad + fmg\left(\dot{\theta}_{i-1}+\dot{\theta}_{i+1}\right)\sin\theta_i \qquad (2.16)\end{aligned}$$

Introduciendo de nuevo la definición de velocidad angular $\omega_i = \dot{\theta}_i$ se obtiene

$$\begin{aligned} \dot{\theta}_i &= \omega_i && (2.17)\\ \dot{\omega}_i &= -\frac{g}{l}\sin\theta_i + f\frac{g}{l}\left(\theta_{i-1}+\theta_{i+1}\right)\cos\theta_i && (2.18)\\ \dot{T}_i &= 3\omega_i\left[fmg\left(\theta_{i-1}+\theta_{i+1}\right)\cos\theta_i - mg\sin\theta_i\right]\\ &\quad + fmg\left(\omega_{i-1}+\omega_{i+1}\right)\sin\theta_i && (2.19)\end{aligned}$$

Se llega entonces a un sistema de $3\times 4 = 12$ ecuaciones diferenciales acopladas. Note que haciendo el parámetro de acoplamiento nulo $f = 0$ se llega de nuevo al péndulo simple de la demostración 2.

El conjunto de condiciones iniciales se definirá de tal forma que el sistema presente resonancia entre dos péndulos, para esto los tres primeros péndulos partirán del reposo mientras el cuarto péndulo comenzará oscilando con una amplitud de $\theta_4(t = 0) = 45^o = \pi/4$ rad. La longitud de los péndulos serán respectivamente $l_1 = 1$ m, $l_2 = 2$ m, $l_3 = 3$ m y $l_4 = 2$ m, de esta forma se garantiza una resonancia entre los péndulos 2 y 4. El estudiante puede modificar estas condiciones en los códigos y así recrear las situaciones que desee.

Esta tercera demostración se dividirá en dos diferentes códigos, el primer script realizará la integración numérica de los péndulos y guardará las soluciones en un archivo de datos (`demo2_03_1.py`), el segundo script cargará estos archivos de datos y realizará la animación 3D del sistema (`demo2_03_2.py`) [1].

```
#!/usr/bin/env python
#============================================================
# DEMOSTRACION 3: Parte 1
# Solucion numerica de 4 pendulos simples acoplados
#============================================================
from __future__ import division
import numpy as np
import matplotlib.pylab as plt
import scipy.integrate as integ

#Ecuaciones de movimiento
def dF(Y, t):
    #Valor anterior de theta pendulo 1
    theta1 = Y[0]
    #Valor anterior de omega pendulo 1
    omega1 = Y[1]
    #Valor anterior de la tension pendulo 1
    tension1 = Y[2]

    #Valor anterior de theta pendulo 2
    theta2 = Y[3]
    #Valor anterior de omega pendulo 2
    omega2 = Y[4]
    #Valor anterior de la tension pendulo 2
    tension2 = Y[5]

    #Valor anterior de theta pendulo 3
    theta3 = Y[6]
    #Valor anterior de omega pendulo 3
    omega3 = Y[7]
    #Valor anterior de la tension pendulo 3
    tension3 = Y[8]

    #Valor anterior de theta pendulo 4
```

[1] Recuerde que puede descargar los códigos originales del repositorio oficial del curso en `https://github.com/sbustamante/Computacional-OscilacionesOndas/tree/master/codigos`

```
35      theta4 = Y[9]
36      #Valor anterior de omega pendulo 4
37      omega4 = Y[10]
38      #Valor anterior de la tension pendulo 4
39      tension4 = Y[11]
40
41      #Derivada de theta pendulo 1
42      Dtheta1 = omega1
43      #Derivada de omega pendulo 1
44      Domega1 = -(g/l1)*np.sin( theta1 ) + \
45      f*(g/l1)*( theta2 )*np.cos( theta1 )
46      #Derivada de la tension pendulo 1
47      Dtension1 = -3*omega1*(-m*g*np.sin( theta1 ) + \
48      f*m*g*( theta2 )*np.cos( theta1 )) + \
49      f*m*g*( omega2 )*np.sin( theta1 )
50
51      #Derivada de theta pendulo 2
52      Dtheta2 = omega2
53      #Derivada de omega pendulo 2
54      Domega2 = -(g/l2)*np.sin( theta2 ) + \
55      f*(g/l2)*( theta1 + theta3 )*np.cos( theta2 )
56      #Derivada de la tension pendulo 2
57      Dtension2 = 3*omega2*(-m*g*np.sin( theta2 ) + \
58      f*m*g*( theta1 + theta3 )*np.cos( theta2 )) + \
59      f*m*g*( omega1 + omega3 )*np.sin( theta2 )
60
61      #Derivada de theta pendulo 3
62      Dtheta3 = omega3
63      #Derivada de omega pendulo 3
64      Domega3 = -(g/l3)*np.sin( theta3 ) + \
65      f*(g/l3)*( theta2 + theta4 )*np.cos( theta3 )
66      #Derivada de la tension pendulo 3
67      Dtension3 = 3*omega3*(-m*g*np.sin( theta3 ) + \
68      f*m*g*( theta2 + theta4 )*np.cos( theta3 )) + \
69      f*m*g*( omega2 + omega4 )*np.sin( theta3 )
70
71      #Derivada de theta pendulo 4
72      Dtheta4 = omega4
73      #Derivada de omega pendulo 4
74      Domega4 = -(g/l4)*np.sin( theta4 ) + \
75      f*(g/l4)*( theta3 )*np.cos( theta4 )
76      #Derivada de la tension pendulo 4
77      Dtension4 = 3*omega4*(-m*g*np.sin( theta4 ) + \
```

```
78      f*m*g*( theta3 )*np.cos( theta4 )) + \
79      f*m*g*( omega3 )*np.sin( theta4 )
80
81      return (Dtheta1, Domega1, Dtension1, \
82      Dtheta2, Domega2, Dtension2, \
83      Dtheta3, Domega3, Dtension3, \
84      Dtheta4, Domega4, Dtension4)
85
86  #Gravedad
87  g = 9.8
88  #Masa de todos los pendulos
89  m = 1.
90  #Longitud pendulo 1
91  l1 = 1.0
92  #Longitud pendulo 2
93  l2 = 2.0
94  #Longitud pendulo 3
95  l3 = 3.0
96  #Longitud pendulo 4
97  l4 = 2.0
98  #Factor de acoplamiento
99  f = 0.1
100 #Tiempos
101 tiempo = np.arange( 0, 200, 0.1 )
102
103
104 #SOLUCION NUMERICA
105 #Condicion inicial pendulo 1
106 #Angulo inicial
107 theta1_t0 = 0.0
108 #Velocidad angular inicial
109 omega1_t0 = 0.0
110 #Tension inicial
111 tension1_t0 = m*l1*omega1_t0**2 + m*g*np.cos( theta1_t0 )
112
113 #Condicion inicial pendulo 2
114 #Angulo inicial
115 theta2_t0 = 0.0
116 #Velocidad angular inicial
117 omega2_t0 = 0.0
118 #Tension inicial
119 tension2_t0 = m*l2*omega2_t0**2 + m*g*np.cos( theta2_t0 )
120
```

```
121 #Condicion inicial pendulo 3
122 #Angulo inicial
123 theta3_t0 = 0.0
124 #Velocidad angular inicial
125 omega3_t0 = 0.0
126 #Tension inicial
127 tension3_t0 = m*l3*omega3_t0**2 + m*g*np.cos( theta3_t0 )
128
129 #Condicion inicial pendulo 4
130 #Angulo inicial
131 theta4_t0 = np.pi/4
132 #Velocidad angular inicial
133 omega4_t0 = 0.0
134 #Tension inicial
135 tension4_t0 = m*l3*omega3_t0**2 + m*g*np.cos( theta3_t0 )
136
137 #Condiciones iniciales de todos los pendulos
138 cond_ini = ( theta1_t0, omega1_t0, tension1_t0, \
139 theta2_t0, omega2_t0, tension2_t0,\
140 theta3_t0, omega3_t0, tension3_t0,
141 theta4_t0, omega4_t0, tension4_t0)
142 #Solucion numerica de todos los pendulos
143 theta1_t, omega1_t, tension1_t, \
144 theta2_t, omega2_t, tension2_t, \
145 theta3_t, omega3_t, tension3_t, \
146 theta4_t, omega4_t, tension4_t = \
147 np.transpose( integ.odeint( dF, cond_ini, tiempo ) )
148
149 #Guardado de resultados en archivo externo
150 datos = np.transpose((tiempo, theta1_t, theta2_t, \
151 theta3_t, theta4_t))
152 np.savetxt( 'amplitudes.txt', datos )
153
154 #Grafica de las soluciones
155 plt.plot( tiempo, theta1_t, label = 'pendulo 1', \
156 color='red', linewidth = 2 )
157 plt.plot( tiempo, theta2_t, label = 'pendulo 2', \
158 color='blue', linewidth = 2 )
159 plt.plot( tiempo, theta3_t, label = 'pendulo 3', \
160 color='green', linewidth = 2 )
161 plt.plot( tiempo, theta4_t, label = 'pendulo 4', \
162 color='black', linewidth = 2 )
163
```

```
164 #Formato de grafica
165 plt.title('Soluciones de pendulos acoplados')
166 plt.xlabel('tiempo')
167 plt.ylabel('Angulo oscilacion [rad]')
168 plt.grid()
169 plt.legend()
170 plt.show()
```

Una vez corrido el programa se debe obtener el archivo `amplitudes.txt` con las soluciones angulares, y la siguiente figura

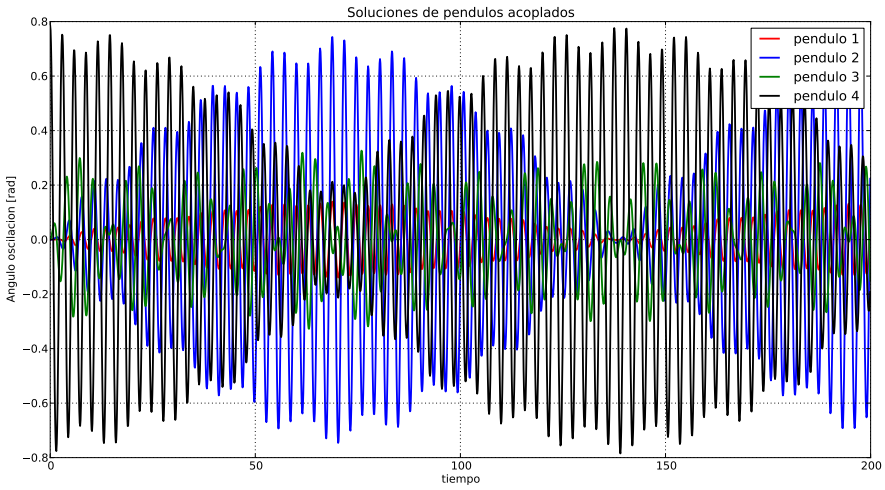


Figura 2.7: Solución de la amplitud para cada uno de los cuatro péndulos acoplados del sistema. Puede notarse que a pesar de que los péndulos 1 y 3 comienzan a oscilar, absorbiendo parte de la energía del péndulo 4, el péndulo 2 lo hace más eficientemente debido a tener la misma longitud, presentando resonancia.

A continuación se explica el código anterior

```
from __future__ import division
import numpy as np
import matplotlib.pylab as plt
import scipy.integrate as integ
```

La primera línea hace que *Python* divida entre enteros de forma adecuada ya que por defecto cuando se dividen dos enteros se retorna el residuo entero entre la división, por ejemplo sin la primera línea `3/2 = 1`, y con ella, `3/2 = 1.5`. Se

recomienda siempre usarla. Las demás lineas corresponden a las librerías ya conocidas importadas bajo los alias estándares.

```
#Guardado de resultados en archivo externo
datos = np.transpose((tiempo, theta1_t, theta2_t, \
theta3_t, theta4_t))
np.savetxt( 'amplitudes.txt', datos )
```

En la primera línea se crea un arreglo con las diferentes columnas del archivo que se desea guardar, en este caso es el arreglo de tiempo en la primera y los arreglos con la solución de los ángulos para los 4 péndulos en las siguientes. El operador backslash \ permite saltar de línea en *Python* sin que haya problemas de sintaxis y sirve para organizar el código y hacerlo más legible. La función de *NumPy* `transpose` se usa para transponer la matriz (arreglo) de datos para que el archivo externo esté organizado por columnas y no por filas. Finalmente la función `savetxt` permite guardar todos los datos en un archivo externo llamado `amplitudes.txt`.

```
#Grafica de las soluciones
plt.plot( tiempo, theta1_t, label = 'pendulo 1', \
color='red', linewidth = 2 )
plt.plot( tiempo, theta2_t, label = 'pendulo 2', \
color='blue', linewidth = 2 )
plt.plot( tiempo, theta3_t, label = 'pendulo 3', \
color='green', linewidth = 2 )
plt.plot( tiempo, theta4_t, label = 'pendulo 4', \
color='black', linewidth = 2 )
```

En esta parte se grafican en una misma ventana todas las soluciones para los 4 péndulos. Se usa de nuevo el operador backslash \ para ordenar el código y se aplican nuevos argumentos de la función `plot` de la librería *Matplotlib* . El argumento `color` permite definir el color de la curva mientras que `linewidth` define el grosor, el valor de `1` se toma por defecto si no se especifica.

```
#Formato de grafica
plt.title('Soluciones de pendulos acoplados')
plt.xlabel('tiempo')
plt.ylabel('Angulo oscilacion [rad]')
plt.grid()
plt.legend()
plt.show()
```

En esta última parte se usan funciones de *Matplotlib* para dar formato a las gráficas. En primer lugar se asigna un título con la función `title`, luego se asignan etiquetas en cada eje con las funciones `xlabel` y `ylabel`, se activa una malla sobre la ventana con la función `grid` y finalmente se muestran las etiquetas de cada curva con `legend` y se muestra el resultado en pantalla con `show`.

Como segunda parte de esta demostración se usaran los datos integrados y almacenados en el archivo `amplitudes.txt` para representar una animación 3D del sistema. El script en *Python* para realizar esto es el siguiente

```
#!/usr/bin/env python
#============================================================
# DEMOSTRACION 3: Parte 2
# Animacion grafica de 4 pendulos acoplados
#============================================================
import numpy as np
import enthought.tvtk.tools.visual as visual

#Cargando datos de los pendulos
tiempo, theta1_t, theta2_t, theta3_t, theta4_t = \
np.transpose( np.loadtxt('amplitudes.txt') )

#Gravedad
g = 9.8
#Longitud pendulo 1
l1 = 1.0
#Longitud pendulo 2
l2 = 2.0
#Longitud pendulo 3
l3 = 3.0
#Longitud pendulo 4
l4 = 2.0
#Radio de cada esfera
radio = 0.2

#Creando pendulo 1
pendulo1 = visual.sphere( radius=radio, \
color=(0.0, 0.0, 1.0) )
pendulo1.pos = [ 1, -l1, 0 ]
pendulo1.t = 0
pendulo1.dt = 1
```

```
cuerda1 = visual.curve( points=[(1,0,0), (1,-l1,0)], \
radius=0.02 )

#Creando pendulo 2
pendulo2 = visual.sphere( radius=radio, \
color=(0.0, 0.0, 1.0) )
pendulo2.pos = [ 2, -l2, 0 ]
pendulo2.t = 0
pendulo2.dt = 1
cuerda2 = visual.curve( points=[(2,0,0), (2,-l2,0)], \
radius=0.02 )

#Creando pendulo 3
pendulo3 = visual.sphere( radius=radio, \
color=(0.0, 0.0, 1.0) )
pendulo3.pos = [ 3, -l3, 0 ]
pendulo3.t = 0
pendulo3.dt = 1
cuerda3 = visual.curve( points=[(3,0,0), (3,-l3,0)], \
radius=0.02 )

#Creando pendulo 4
pendulo4 = visual.sphere( radius=radio, \
color=(1.0, 0.0, 0.0) )
pendulo4.pos = [4, -l4*np.cos(theta4_t[0]), \
l4*np.sin(theta4_t[0])]
pendulo4.t = 0
pendulo4.dt = 1
cuerda4 = visual.curve( points=[(4,0,0), \
(4, -l4*np.cos(theta4_t[0]), l4*np.sin(theta4_t[0]))], \
radius=0.02 )

#Creando caja contenedora y cuerda de suspension
muro1 = visual.box( pos=(0., -1., 0.), size=(0.3, 5, 1), \
color=(0.6, 0.3, 0.0) )
muro2 = visual.box( pos=(5., -1., 0.), size=(0.3, 5, 1), \
color=(0.6, 0.3, 0.0) )
visual.curve( points=[(0,0,0), (5,0,0)], radius=0.02 )

#ITERACION DEL SISTEMA
def anim():
    #Evolucion del pendulo 1
    pendulo1.t = pendulo1.t + pendulo1.dt
```

```
75      i = pendulo1.t
76      pendulo1.pos = visual.vector( 1, \
77      -l1*np.cos(theta1_t[i]), l1*np.sin(theta1_t[i]) )
78      delta_thetha1 = theta1_t[i-1] - theta1_t[i]
79      cuerda1.rotate( 180*delta_thetha1/np.pi, [1.,0.,0] )
80
81      #Evolucion del pendulo 2
82      pendulo2.t = pendulo2.t + pendulo2.dt
83      i = pendulo2.t
84      pendulo2.pos = visual.vector( 2, \
85      -l2*np.cos(theta2_t[i]), l2*np.sin(theta2_t[i]) )
86      delta_thetha2 = theta2_t[i-1] - theta2_t[i]
87      cuerda2.rotate( 180*delta_thetha2/np.pi, [1.,0.,0] )
88
89      #Evolucion del pendulo 3
90      pendulo3.t = pendulo3.t + pendulo3.dt
91      i = pendulo3.t
92      pendulo3.pos = visual.vector( 3, \
93      -l3*np.cos(theta3_t[i]), l3*np.sin(theta3_t[i]) )
94      delta_thetha3 = theta3_t[i-1] - theta3_t[i]
95      cuerda3.rotate( 180*delta_thetha3/np.pi, [1.,0.,0] )
96
97      #Evolucion del pendulo 4
98      pendulo4.t = pendulo4.t + pendulo4.dt
99      i = pendulo4.t
100     pendulo4.pos = visual.vector( 4, \
101     -l4*np.cos(theta4_t[i]), l4*np.sin(theta4_t[i]) )
102     delta_thetha4 = theta4_t[i-1] - theta4_t[i]
103     cuerda4.rotate( 180*delta_thetha4/np.pi, [1.,0.,0] )
104
105 a = visual.iterate(10, anim)
106 visual.show()
```

De esto se obtiene una animación 3D del sistema de péndulos, tal como la figura 2.8

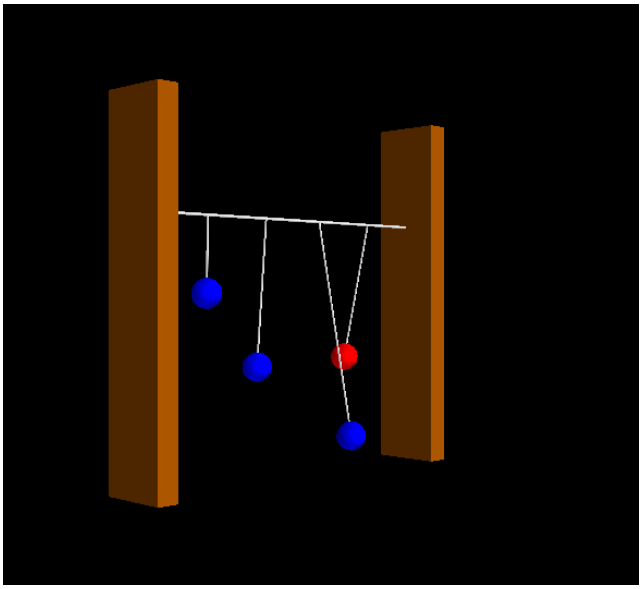


Figura 2.8: Animación 3D del sistema de péndulos acoplados.

A continuación se explica el código anterior

```
import enthought.tvtk.tools.visual as visual
```

En esta línea se importa la librería `tvtk` de *MayaVi2* , esta contiene todas las funcionalidades para realizar la animación 3D. Se usa el alias de `visual`.

```
#Cargando datos de los pendulos
tiempo, theta1_t, theta2_t, theta3_t, theta4_t = \
np.transpose( np.loadtxt('amplitudes.txt') )
```

En esta parte se cargan los arreglos de tiempo y soluciones de los 4 péndulos. Para esto se usa la función `loadtxt` de la librería *NumPy* . Como único argumento de esta función se da el nombre del archivo de datos (`amplitudes.txt`) generados por el script 1 de esta demostración. Para cargar los datos acorde a las columnas del archivo externo y no a sus filas se debe usar la función `transpose`.

```
#Creando pendulo 1
pendulo1 = visual.sphere( radius=radio, \
color=(0.0, 0.0, 1.0) )
pendulo1.pos = [ 1, -l1, 0 ]
pendulo1.t = 0
pendulo1.dt = 1
```

```
cuerda1 = visual.curve( points=[(1,0,0), (1,-l1,0)], \
radius=0.02 )
```

Se definen las propiedades de los objetos gráficos asociados al péndulo 1. Primero se define un objeto tipo `sphere` de la librería `visual`, como argumentos se dan el radio de la esfera `radius` y el color de esta `color`. El formato de color está en código RGB donde un color es un arreglo de tres números que toman valores entre 0 y 1. La primera componente determina la cantidad de rojo, con 0 nulo y 1 máximo, de igual forma la segunda componente para el verde y la tercera para el azul. Como ejemplos, el rojo equivale a `(1,0,0)`, mientras que el negro `(0,0,0)`, el blanco `(1,1,1)`, el violeta `(1,0,1)`, etc.
Luego se define la posición inicial del péndulo como `pendulo1.pos = [ 1, -l1, 0 ]`, lo que significa que el péndulo tiene componente $x = 1$, mientras que $y = -l_1$, o equivalentemente el péndulo comienza del punto de equilibrio. En las dos siguientes líneas se establece el tiempo inicial y el salto de tiempo `pendulo1.dt = 1`. Se debe tener en cuenta que el salto de tiempo es un número entero y es `1` y no corresponde al `dt` dado en el script 1 de esta demostración, esto debido a que el tiempo en este caso es discreto ya que está asociado a la posición en los arreglos pre-cargados y no al tiempo físico.
Finalmente se define el objeto `cuerda1` como una curva tipo `curve`, esta tiene como argumentos la posición de los dos extremos de la cuerda `points`, correspondientes a la base del péndulo y al centro de la esfera, y también el radio de la cuerda `radius`. Como argumento opcional puede definirse un color, pero en este caso se deja el valor por defecto correspondiente a blanco.
Esto se repite para los 4 péndulos del sistema.

```
#Creando caja contenedora y cuerda de suspension
muro1 = visual.box( pos=(0., -1., 0.), size=(0.3, 5, 1), \
color=(0.6, 0.3, 0.0) )
muro2 = visual.box( pos=(5., -1., 0.), size=(0.3, 5, 1), \
color=(0.6, 0.3, 0.0) )
visual.curve( points=[(0,0,0), (5,0,0)], radius=0.02 )
```

Se crean los objetos asociados al soporte de los péndulos. Para el muro 1 se crean un objeto tipo `visual.box` de la siguiente forma `muro1 = visual.box( pos=(0., -1., 0.), size=(0.3, 5, 1), color=(0.6, 0.3, 0.0) )`. Como primer argumento se da la posición del centro geométrico de la caja, el segundo argumento es un arreglo con el grosor, el alto y el ancho de la caja, en el mismo orden. Finalmente se da el color en formato RGB, con `(0.6, 0.3, 0.0)` equivalente al color café. Por último se crea la cuerda que sostiene los péndulos, se da las dos posiciones asociadas a los extremos y el radio de la cuerda.

```
#ITERACION DEL SISTEMA
def anim():
    #Evolucion del pendulo 1
    pendulo1.t = pendulo1.t + pendulo1.dt
    i = pendulo1.t
    pendulo1.pos = visual.vector( 1, \
    -l1*np.cos(theta1_t[i]), l1*np.sin(theta1_t[i]) )
    delta_thetha1 = theta1_t[i-1] - theta1_t[i]
    cuerda1.rotate( 180*delta_thetha1/np.pi, [1.,0.,0] )
```

Función de iteración `anim`. Esta función itera el sistema completo, creando la animación de cada péndulo a partir de la solución numérica pre-cargada. En la primera línea de la función se define el tiempo *discreto* actual como `pendulo1.t = pendulo1.t + pendulo1.dt` y corresponde a la posición `i` en los arreglos de datos.
En la línea siguiente se actualiza la nueva posición del péndulo, para esto se accede a la solución del ángulo en el tiempo actual como `theta1_t[i]` y se convierte de coordenadas polares a coordenadas cartesianas como `visual.vector( 1, -l1*np.cos(theta1_t[i]), l1*np.sin(theta1_t[i]) )`
Las últimas dos líneas rotan la posición de la cuerda para que esté acorde a la posición del péndulo, para esto se define una nueva variable como `delta_thetha1 = theta1_t[i-1] - theta1_t[i]`. En la última línea se rota la cuerda un angulo `delta_thetha1` respecto al eje `[1.,0.,0]`, para esto se usa el método `rotate` del objeto `cuerda1`. Como primer argumento se da el ángulo que se desea rotar la cuerda, este debe ser dado en unidades de grados, no de radianes. La razón para restar el ángulo actual del péndulo y el angulo del tiempo anterior en la definición de `delta_thetha1` se debe a que la cuerda en el tiempo anterior ya estaba rotada, así el nuevo ángulo no debe ser medido desde el origen sino desde la última posición de la cuerda. El segundo argumento corresponde al eje sobre el cual se desea hacer la rotación, este debe ser perpendicular al plano en el cual se mueve el péndulo y es en este caso `[1.,0.,0]`.
Este procedimiento se repite para cada uno de los péndulos del sistema.

```
a = visual.iterate(10, anim)
visual.show()
```

Finalmente se ejecuta la función `visual.iterate`, esta presenta la animación del sistema de forma gráfica. Como primer argumento se da el tiempo en milisegundos que se quiere entre cada iteración del sistema, así por ejemplo un valor bajo como `10` implica un tiempo de 10 ms entre frame y frame, produciendo una

animación más fluida, mientras que un valor más alto como `50` produce una animación en cámara lenta. El segundo argumento es el nombre de la función donde está la evolución de los péndulos, en este caso `anim`.
Finalmente se ejecuta la función `visual.show()` la cual muestra en pantalla todo el resultado obtenido. Note que esta función `show` pertenece a la librería `tvtk` de *MayaVi2* y no debe confundirse con la función `show` de *Matplotlib* la cual representa en pantalla gráficas y no animaciones.

2.4. Ejercicios

Ejercicio 1 $(\star)$

Soluciones del sistema masa-resorte
Considere el sistema masa resorte que se ilustra en la figura 2.9 y considere las siguientes situaciones

- Sistema masa-resorte ideal, la fricción es despreciable con la superficie horizontal.
- Sistema masa-resorte amortiguado, la fricción con la superficie es significativa y el coeficiente de fricción está dado por μ.
- Sistema masa-resorte forzado, la fricción es de nuevo despreciable pero existe un agente forzante externo con una forma $\boldsymbol{F}_{ext} = \boldsymbol{F_0}\sin(\omega_f t)$.

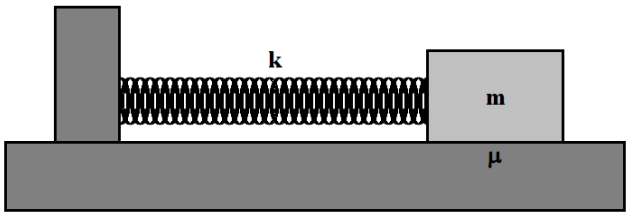


Figura 2.9: Sistema masa resorte.

A partir de las situaciones descritas, realizar los siguientes numerales

a) Solucionar analíticamente el sistema péndulo-resorte para las tres situaciones y realizar un código que grafique las tres soluciones, usando la misma condición inicial, en una misma ventana. Tanto las condiciones iniciales como los parámetros del sistema (m, k, μ, etc.) son de libre elección, además puede escoger el rango de tiempo en el cual quiere presentar las soluciones.

Realice este mismo numeral para otro conjunto de condiciones iniciales, en este caso realice una copia del código con las nuevas condiciones.

b) Graficar en una misma ventana el comportamiento de la energía en cada una de las tres situaciones, para esto use el mismo conjunto de condiciones iniciales del numeral pasado.

Realice este mismo numeral para el segundo conjunto de condiciones iniciales, en este caso realice una copia del código con las nuevas condiciones.

Ejercicio 2 $(\star)$

Solución numérica del péndulo-resorte
El sistema péndulo resorte consiste en una masa atada a un resorte y a su vez este pende de un punto fijo, tal como es mostrado en la figura 2.10.

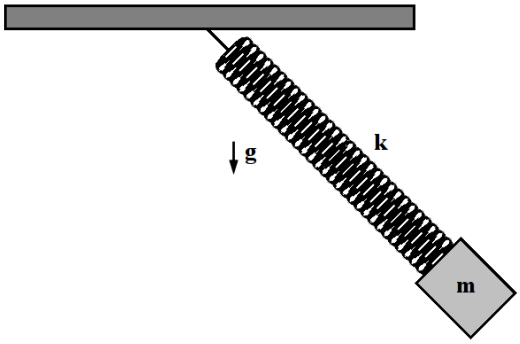


Figura 2.10: Sistema péndulo resorte.

Plantee las ecuaciones de movimiento de este sistema en coordenadas polares y llegue a un sistema de ecuaciones diferenciales lineales. Sin usar la aproximación de pequeñas oscilaciones, haga un código que resuelva numéricamente este problema para un conjunto de condiciones iniciales de su preferencia. Finalmente grafique la solución, en una ventana muestre la solución angular vs tiempo y en otra muestre la solución radial vs tiempo. Guarde en un archivo externo *.txt* los datos obtenidos con el formato *[tiempo, solución radial, solución angular]*. Escoja los valores numéricos de los parámetros del sistema que usted desee.

BONUS $(\star)$
Este ejercicio puede ser considerado de dos estrellas si se realiza lo siguiente. Haga otro código que cargue el archivo de datos de la solución obtenida del péndulo resorte y haga una animación 3D en *MayaVi2* que ilustre la evolución del sistema.

Ejercicio 3 $(\star)$

Solución numérica de péndulos acoplados.
El sistema de dos péndulos acoplados por un resorte es ilustrado en la figura siguiente 2.11.

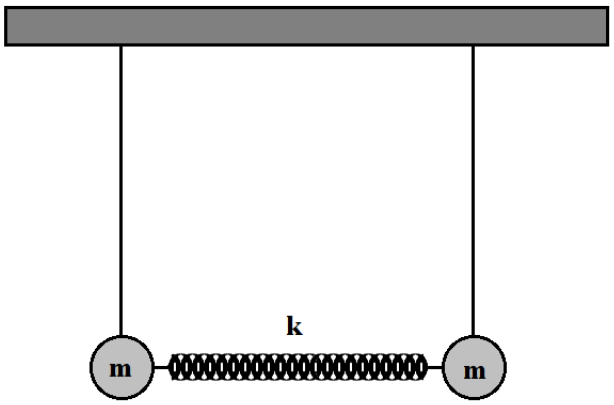


Figura 2.11: Sistema péndulo resorte.

Plantee las ecuaciones de movimiento de este sistema y llegue a un sistema de ecuaciones diferenciales lineales. Sin usar la aproximación de pequeñas oscilaciones, haga un código que resuelva numéricamente este problema para un conjunto de condiciones iniciales de su preferencia. Finalmente grafique la solución de los dos péndulos en una misma ventana, observa resonancia? Guarde en un archivo externo *.txt* los datos obtenidos con el formato *[tiempo, solución péndulo 1, solución péndulo 2]*. Escoja los valores numéricos de los parámetros del sistema que usted desee.

BONUS $(\star)$
Este ejercicio puede ser considerado de dos estrellas si se realiza lo siguiente. Haga otro código que cargue el archivo de datos de la solución obtenida del sistema de péndulos acoplados por resorte y haga una animación 3D en *MayaVi2* que ilustre la evolución del sistema.

Ejercicio 4 $(\star)$

Figuras de Lissajous
Las figuras de Lissajous son un conjunto de curvas que se forman cuando se superponen dos movimientos armónicos en direcciones ortogonales. Una forma conveniente de recrear esto es a partir de un sistema de una masa y dos resortes ortogonales atados a móviles, tal como se ilustra en la figura 2.13

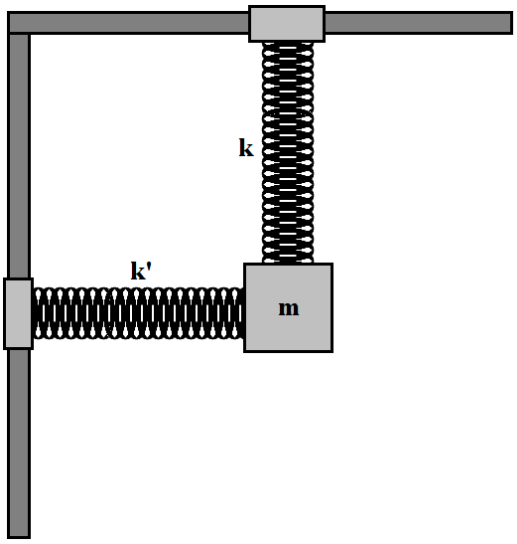


Figura 2.12: Sistema de masa y dos resortes ortogonales.

Para este ejercicio, plantee las ecuaciones de movimiento y resuelva de forma analítica el problema. Tenga en cuenta que el sistema es ideal y no presenta forzamiento. Una vez solucionadas las ecuaciones, grafique en una misma ventana 4 diferentes soluciones variando las constantes de los resortes y las condiciones iniciales, en especial intente encontrar constantes k y k' tal que se produzcan algunas de las figuras de Lissajous. Finalmente guarde la solución en un archivo externo *.txt* con el formato *[tiempo, solución coordenada X, solución coordenada Y]*.

BONUS $(\star)$
Este ejercicio puede ser considerado de dos estrellas si se realiza lo siguiente. Otro código que cargue el archivo de datos de la solución obtenida del sistema de la figura 2.11 y realice una animación 3D en *MayaVi2* que ilustre la evolución del sistema.

Ejercicio 5 $(\star\star)$

Sistema de masas resortes acoplados
En la siguiente figura se muestra un sistema de 5 masas acopladas a través de resortes.

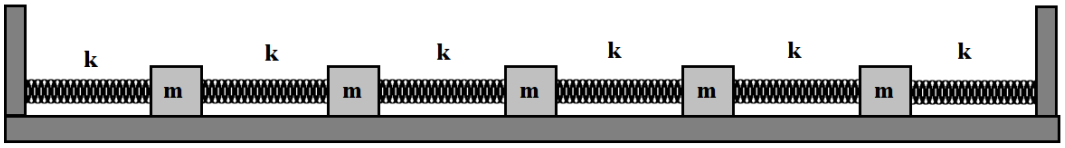


Figura 2.13: Sistema de 5 masas acopladas por resortes.

Plantee las ecuaciones de movimiento para este sistema y haga un código en el cual se resuelvan numéricamente teniendo en cuenta una fricción con la superficie horizontal. Para esto asuma que todos las masas están inicialmente en reposo y en equilibrio excepto la primera masa de uno de los extremos, la cual comienza con una cierta amplitud respecto a la posición de equilibrio. Usando la función `savefig` de la librería *Matplotlib* guarde una gráfica donde se superpongan las soluciones de la amplitud de los péndulos respecto al tiempo. Finalmente en el mismo código, realice una animación 3D usando *MayaVi2* de la evolución de todo el sistema. Asuma las 5 masas m iguales y las constantes de los 6 resortes k.

Note como la evolución del sistema conlleva a una propagación del movimiento de la primera masa hacia las demás, esto permite ilustrar de forma básica el concepto de onda.

Capítulo 3

Ondas Elásticas

3.1. Demostración 2: Efecto Doppler

El efecto doppler es un fenómeno físico que se presenta en sistemas donde hay fuentes y observadores en movimiento relativo. En esta demostración se abordará el efecto dopplet acústico simulando la señal obtenida por el movimiento de una fuente respecto a un obsevador en reposo.

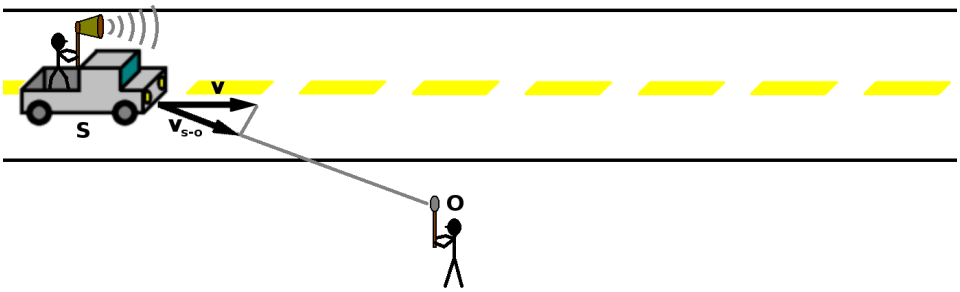


Figura 3.1: Ilustración de efecto doppler acústico con una fuente en movimiento.

Por simplicidad se asumirá que la señal emitida por la fuente $\boldsymbol{S}$ es sinusoidal con la forma

$$P(r,t) = P_0 \sin\left(2\pi f_S t + \delta\right) \tag{3.1}$$

donde $P(r,t)$ es la función de presión de la onda, medida respecto a la presión media de fondo (presión atmosférica), P_0 la amplitud de la onda sonora y f su frecuencia.

La expresión 3.3 es válida para un sistema de referencia que se mueve con la fuente. Para determinar la señal percibida por el observador en reposo se hace uso de las siguientes expresiones para la transformación de frecuencias

$$f_O = \begin{cases} f_S v_s/(v_s - v_{SO}) & \text{Si la fuente se aleja} \\ f_S v_s/(v_s + v_{SO}) & \text{Si la fuente se acerca} \end{cases} \tag{3.2}$$

donde f_O es la frecuencia detectada por el observador, f_S la frecuencia original de la fuente, v_s es la velocidad del sonido en el aire ($v_s = 331{,}5$ m/s) y v_{SO} la velocidad de la fuente en la dirección del observador.

Sin pérdida de generalidad se toma la velocidad de la fuente constante, y se asume un decaimiento exponencial del volumen en función de la distancia, así, la señal obtenida por el observador está dada por

$$P'(r,t) = P_0 e^{-\lambda d} \sin\left[2\pi f_O(v)t + \delta\right] \tag{3.3}$$

o equivalentemente

$$P'(r,t) = P_0 e^{-\lambda d} \begin{cases} \sin\left[2\pi f_S t \frac{v_s}{v_s - v_{SO}}\right] & \text{Si la fuente se aleja} \\ \sin\left[2\pi f_S t \frac{v_s}{v_s + v_{SO}}\right] & \text{Si la fuente se acerca} \end{cases} \tag{3.4}$$

Donde λ es la distancia característica de decaimiento de la señal.

Para ejecutar el código de esta demostración se debe instalar el paquete *pyaudio* . Este paquete es una amplia librería para manejo de audio en *Python* . Se encuentra en los repositorios oficales, de manera que para su instalación basta con correr la siguiente línea en bash

```
\$ sudo apt-get install python-pyaudio
```

También debe usarse el archivo `AudioLib.py` el cual es un mask diseñado para un uso más sencillo de *pyaudio* .

El código se muestra a continuación

```
1 # -*- coding: utf-8 -*-
2 #!/usr/bin/env python
3 #============================================================
4 # DEMOSTRACION 2
5 # Efecto Doppler
6 #============================================================
```

```
7
8  #*********************************************************
9  #       MODULOS
10 #*********************************************************
11 import numpy as np
12 import os
13 import matplotlib.pylab as plt
14 import AudioLib as ad
15
16 #Velocidad del sonido [m/s]
17 vs = 331.5
18
19 #Creando objeto de audio
20 sonido = ad.audio()
21
22 #Intervalo de tiempo
23 dt = 1/(44100.)
24 #Frecuencia de nota [Hz]
25 freq0 = 110.
26
27 #Tiempo maximo  [s]
28 tmax = 6.
29 #Arreglo de tiempo
30 tiempo = np.arange( 0, tmax, dt )
31 #Nota a reproducir
32 nota = ad.Amplitude*np.sin( 2*np.pi*freq0*tiempo )
33
34 #Cargando nota de audio
35 sonido.load( nota )
36 #Reproduciendo sonido
37 sonido.play()
38
39 #EFECTO DOPPLER ============================================
40 #Creando objeto de audio asociado al observador
41 sonido_doppler = ad.audio()
42
43 #Distancia del obervador a la carretera [m]
44 Lo = 1.0
45 #Velocidad del carro [m/s]
46 v_car = 6.0
47 #Distancia del carro inicialmente [m]
48 d_car = 18.0
49 #Tiempo de efecto doppler [s]
```

```
50 tmaxD = 6.0
51 #Arreglo de tiempo
52 tiempoD = np.arange( 0, tmaxD, dt )
53 #Numero de intervalos
54 Nt = int(len(tiempoD)/2.)
55
56 #Funcion de posicion del carro
57 def pos( t ):
58     return -d_car + v_car*t
59
60 #Funcion de velocidad del carro dirigida a la fuente
61 def vel( t ):
62     return v_car*1/np.sqrt( 1 + Lo**2/pos(t)**2 )
63
64 #Funcion de decaimiento de intensidad
65 def I( d ):
66     return ad.Amplitude*np.exp(-abs(d)/20.)
67
68 #Transformacion de frecuencia Dopple
69 def freq( freq0, v, cond ):
70     if cond == "Aleja":
71         return freq0*vs/(vs - v)
72     if cond == "Acerca":
73         return freq0*vs/(vs + v)
74
75 #Calculo de efecto Doppler para la nota inicial
76 notaD = np.zeros( 2*Nt )
77 #Cuando la fuente se acerca
78 notaD[:Nt] = I( pos(tiempoD[:Nt]) )*\
79 np.sin( 2*np.pi*freq(freq0, vel( tiempoD[:Nt] ), 'Acerca')*
      tiempoD[:Nt] )
80 #Cuando la fuente se aleja
81 notaD[Nt:] = I( pos(tiempoD[Nt:]) )*\
82 np.sin( 2*np.pi*freq(freq0, vel( tiempoD[Nt:] ), 'Aleja')*
      tiempoD[Nt:] )
83
84 #Cargando nota de audio
85 sonido_doppler.load( notaD )
86 #Reproduciendo sonido
87 sonido_doppler.play()
88 #Graficando Sonido
89 sonido_doppler.plot()
```

El resultado obtenido es un audio correspondiente a la señal en el marco de referencia de la fuente, luego un audio y una gráfica de la señal en el marco del observador.

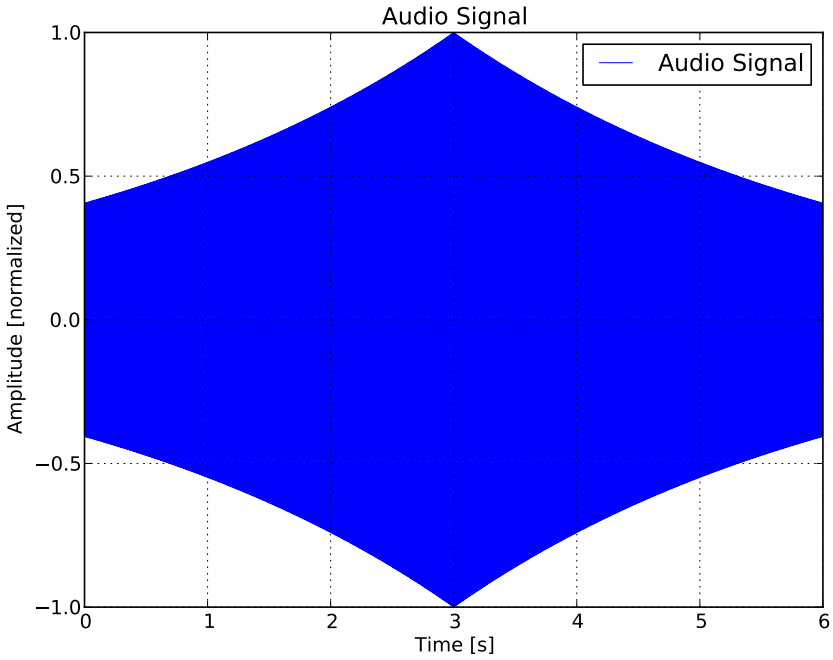


Figura 3.2: Señal medida por el observador.

A continuación se explica cada parte del código

```
import AudioLib as ad
```

Se importa el mask para el manejo de *pyaudio* , el archivo `AudioLib.py` puede ser descargado de la página del curso[1].

```
#Velocidad del sonido [m/s]
vs = 331.5

#Creando objeto de audio
sonido = ad.audio()
```

[1] `https://github.com/sbustamante/Computacional-OscilacionesOndas/tree/master/codigos`

```
#Intervalo de tiempo
dt = 1/(44100.)
#Frecuencia de nota [Hz]
freq0 = 110.

#Tiempo maximo  [s]
tmax = 6.
#Arreglo de tiempo
tiempo = np.arange( 0, tmax, dt )
#Nota a reproducir
nota = ad.Amplitude*np.sin( 2*np.pi*freq0*tiempo )

#Cargando nota de audio
sonido.load( nota )
#Reproduciendo sonido
sonido.play()
```

En esta parte se define la velocidad del sonido en el aire como `vs = 331.5`. Luego se crea un objeto tipo audio, en el cual será almacenada la señal asociada al marco de referencia dela fuente. El intervalo de tiempo `dt` se da en términos de la frecuencia 44100 Hz, la cual es un estándar para archivos de audio (Calidad CD)[2]

Se define la frecuencia original de la nota monofónica respecto al marco de la fuente, en este caso `freq0 = 110.`, correspondiente a una nota *La2*. Luego se da el tiempo máximo de muestreo del audio, en segundos, `tmax = 6.` y se define el arreglo de tiempos como `tiempo = np.arange( 0, tmax, dt ).`

En la siguiente línea se define la nota monofónica como `nota = ad.Amplitude *np.sin( 2*np.pi*freq0*tiempo )`, donde `ad.Amplitude` corresponde a la máxima amplitud que puede ser guardada en un archivo de audio [3]. Finalmente, usando el método `load` de la clase tipo audio, se carga esta nota en el objeto de audio creado inicialmente `sonido.load( nota )` y se reproduce `sonido.play()`.

```
#EFECTO DOPPLER ==========================================
#Creando objeto de audio asociado al observador
```

[2]Un valor inferior produce una calidad de sonido peor. Ver `http://en.wikipedia.org/wiki/44,100_Hz`

[3]Las unidades de `ad.Amplitude` no están relacionadas con unidades físicas, es solo un formato de datos que determina la máxima posible amplitud almacenada.

```
sonido_doppler = ad.audio()

#Distancia del observador a la carretera [m]
Lo = 1.0
#Velocidad del carro [m/s]
v_car = 6.0
#Distancia del carro inicialmente [m]
d_car = 18.0
#Tiempo de efecto doppler [s]
tmaxD = 6.0
#Arreglo de tiempo
tiempoD = np.arange( 0, tmaxD, dt )
#Numero de intervalos
Nt = int(len(tiempoD)/2.)
```

De forma análoga al primer objeto, se crea el objeto de audio para almacenar la señal percibida por el observador `sonido_doppler = ad.audio()`. A continuación se definen los parámetros físicos de la fuente respecto al observador, esto es: `Lo = 1.0` la distancia en metros del observador a la carretera, `v_car = 6.0` la velocidad constante del automóvil, en m/s, `d_car = 18.0` la distancia inicial en metros entre el automóvil y el observador, y finamente el tiempo de muestreo en segundos `tmaxD = 6.0`, el arreglo de tiempo `tiempoD = np.arange( 0, tmaxD, dt )` y la mitad del número de intervalos de tiempo `Nt = int(len(tiempoD)/2.)`.

```
#Funcion de posicion del carro
def pos( t ):
    return -d_car + v_car*t

#Funcion de velocidad del carro dirigida a la fuente
def vel( t ):
    return v_car*1/np.sqrt( 1 + Lo**2/pos(t)**2 )

#Funcion de decaimiento de intensidad
def I( d ):
    return ad.Amplitude*np.exp(-abs(d)/20.)

#Transformacion de frecuencia Dopple
def freq( freq0, v, cond ):
    if cond == "Aleja":
        return freq0*vs/(vs - v)
```

```
    if cond == "Acerca":
        return freq0*vs/(vs + v)
```

En esta parte del código se definen diferentes funciones para el cálculo del efecto doppler. La función `pos( t )` determina la posición del automóvil en función del tiempo respecto al observador, `I( d )` da el decaimiento de la intensidad en función de la distancia y `freq( freq0, v, cond )`, donde `freq0` es la frecuencia original, `v` la velocidad relativa entre fuente y observador y `cond` puede tomar los valores de `'Aleja'` o `'Acerca'` en función de la situación física que se presente.

```
#Calculo de efecto Doppler para la nota inicial
notaD = np.zeros( 2*Nt )
#Cuando la fuente se acerca
notaD[:Nt] = I( pos(tiempoD[:Nt]) )*\
np.sin( 2*np.pi*freq(freq0, vel( tiempoD[:Nt] ), 'Acerca')*
   tiempoD[:Nt] )
#Cuando la fuente se aleja
notaD[Nt:] = I( pos(tiempoD[Nt:]) )*\
np.sin( 2*np.pi*freq(freq0, vel( tiempoD[Nt:] ), 'Aleja')*
   tiempoD[Nt:] )

#Cargando nota de audio
sonido_doppler.load( notaD )
#Reproduciendo sonido
sonido_doppler.play()
#Graficando Sonido
sonido_doppler.plot()
```

Se crea un arreglo un arreglo para almacenar la señal en el marco de referencia del observador y posteriormente se calculan las dos contribuciones, inicialmente cuando el automóvil de acerca `notaD[:Nt] = I( pos(tiempoD[:Nt]) )*np.sin( 2*np.pi*freq(freq0, vel( tiempoD[:Nt]), 'Acerca')*tiempoD[:Nt] )`, y cuando se aleja `notaD[Nt:] = I( pos(tiempoD[Nt:]) )*np.sin( 2*np.pi*freq(freq0, vel( tiempoD[Nt:] ), 'Aleja')*tiempoD[Nt:] )`. Finalmente se carga la señal al objeto de audio, se reproduce y se grafica.

Capítulo 4

Reflexión, Refracción, Óptica e Interferencia

4.1. Demostración 1: Interferencia de ondas en una superficie

La interferencia es un fenómeno que se presenta en movimientos ondulatorios y consiste en la superposición de campos, ya sea de forma constructiva o destructiva. En esta demostración se calcula la interferencia de dos fuentes puntuales sobre una superficie plana (por ejemplo un fluido).

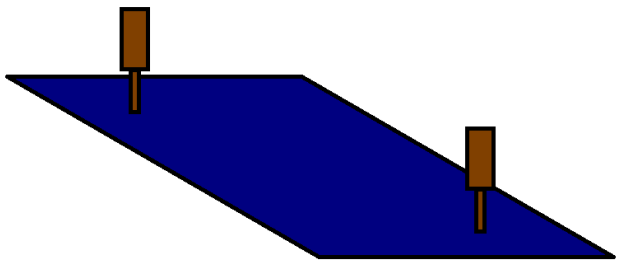


Figura 4.1: Dos fuentes puntuales sobre un fluido.

Por simplicidad se describirá la oscilación producida por las fuentes como una onda radial de la forma

$$z(r,t) = A_0 \sin(\omega t - kr) = A_0 \sin\left[2\pi\left(\frac{t}{f} - \frac{r}{\lambda}\right)\right] \tag{4.1}$$

donde $z(r,t)$ es la función de amplitud en z de la onda, A_0 la amplitud máxima del desplazamiento, ω la frecuencia angular, f la frecuencia y λ la longitud de onda. El valor de r es medido desde el origen de la perturbación, es decir, en la posición de cada fuente. Para calcular la amplitud total se usa el principio de superposición, donde se suman las contribuciones de cada fuente.

En orden para correr el código de la demostración apropiadamente, es necesario instalar el siguiente programa

```
\$ sudo apt-get install ffmpeg
```

Este programa permite generar videos a partir de un conjunto de imágenes nombradas apropiadamente.

A continuación de ilustra el código de la demostración

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python
#============================================================
# DEMOSTRACION 1
# Interferencia de Ondas
#============================================================

#************************************************************
#       MODULOS
#************************************************************
import numpy as np
import os
import matplotlib.pylab as plt

#Funcion de amplitud de una fuente
def Z_amplitud(x, y, x0, y0, A0, t):
    #Posicion de la fuente
    r0 = np.array([x0, y0])
    #Posicion donde se evalua el campo
    r = np.array([x, y])
    return A0*np.sin( 2*np.pi*(t/f - np.linalg.norm(r-r0)/
       lamb ) )

#PARAMETROS
#frecuencia             [Hz]
f = 10.
```

```
26  #Longitud de onda       [m]
27  lamb = 1.0
28  #Tamano de la caja      [m]
29  L = 10.0
30  #Resolucion grid caja
31  NS = 100
32
33  #Posicion X fuente 1    [m]
34  x01 = 1.0
35  #Posicion Y fuente 1    [m]
36  y01 = 2.0
37  #Amplitud fuente 1      [m]
38  A01 = 0.1
39  #Posicion X fuente 2    [m]
40  x02 = 5.0
41  #Posicion Y fuente 2    [m]
42  y02 = 5.0
43  #Amplitud fuente 2      [m]
44  A02 = 0.1
45
46  #Tiempo final           [s]
47  Tmax = 20.0
48  #Numero de intervalos
49  Nstep = 41
50
51  #GRID Y EVOLUCION
52  XM = np.linspace( 0, L, NS )
53  YM = np.linspace( 0, L, NS )
54
55  k = 0
56  for t in np.linspace( 0, Tmax, Nstep ):
57      Z = np.zeros( (NS,NS) )
58      for i in xrange( 0, NS ):
59          for j in xrange( 0, NS ):
60              x = XM[i]
61              y = YM[j]
62              Z[-j,i] = Z_amplitud(x, y, x01, y01, A01, t) + \
63              Z_amplitud(x, y, x02, y02, A02, t)
64
65      plt.xlabel( 'X (0,L)' )
66      plt.ylabel( 'Y (0,L)' )
67      plt.title( 'Interfencia: t=%f' %(t) )
68      plt.imshow( Z, extent = (0,L,0,L), vmax = A01 + A02 )
```

```
69      print t
70      fname='_tmp-%03d.png'%k
71      k+=1
72      plt.savefig(fname)
73      plt.close()
74
75  print 'Making movie animation.mpg - this make take a while'
76  os.system("ffmpeg -f image2 -i _tmp-%03d.png  video.mpg")
77  os.system('rm -rf *.png')
```

El resultado obtenido es un video donde se ilustra la evolución del patrón de interferencia. Una toma del video es realizada en la siguiente figura

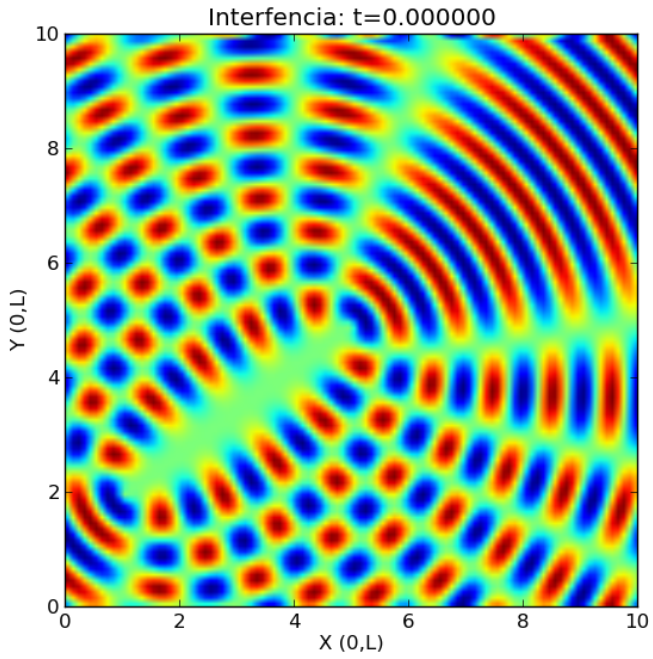


Figura 4.2: Patrón de interferencia de dos fuentes.

A continuación se explica cada parte del código

```
import numpy as np
import os
import matplotlib.pylab as plt
```

Se cargan las librerías estándar.

```
#Funcion de amplitud de una fuente
def Z_amplitud(x, y, x0, y0, A0, t):
    #Posicion de la fuente
    r0 = np.array([x0, y0])
    #Posicion donde se evalua el campo
    r = np.array([x, y])
    return A0*np.sin( 2*np.pi*(t/f - np.linalg.norm(r-r0)/
       lamb ) )
```

Se define la función de amplitud para una fuente puntual. Los argumentos son, la coordenada del punto donde se desea evaluar el campo (`x`, `y`), la coordenada de la fuente (`x0`, `y0`), la amplitud de la oscilación `A0` y el tiempo actual `t`.

```
#PARAMETROS
#frecuencia             [Hz]
f = 10.
#Longitud de onda       [m]
lamb = 1.0
#Tamano de la caja      [m]
L = 10.0
#Resolucion grid caja
NS = 100

#Posicion X fuente 1    [m]
x01 = 1.0
#Posicion Y fuente 1    [m]
y01 = 2.0
#Amplitud fuente 1      [m]
A01 = 0.1
#Posicion X fuente 2    [m]
x02 = 5.0
#Posicion Y fuente 2    [m]
y02 = 5.0
#Amplitud fuente 2      [m]
A02 = 0.1

#Tiempo final           [s]
Tmax = 20.0
#Numero de intervalos
Nstep = 41

#GRID Y EVOLUCION
```

```
XM = np.linspace( 0, L, NS )
YM = np.linspace( 0, L, NS )
```

Se definen todos los parámetros del sistema, incluyendo el grid donde se evaluará el campo de desplazamiento de la onda.

```
k = 0
for t in np.linspace( 0, Tmax, Nstep ):
    Z = np.zeros( (NS,NS) )
    for i in xrange( 0, NS ):
        for j in xrange( 0, NS ):
            x = XM[i]
            y = YM[j]
            Z[-j,i] = Z_amplitud(x, y, x01, y01, A01, t) + \
            Z_amplitud(x, y, x02, y02, A02, t)

    plt.xlabel( 'X (0,L)' )
    plt.ylabel( 'Y (0,L)' )
    plt.title( 'Interfencia: t=%f' %(t) )
    plt.imshow( Z, extent = (0,L,0,L), vmax = A01 + A02 )
    print t
    fname='_tmp-%03d.png' %k
    k+=1
    plt.savefig(fname)
    plt.close()
```

Se realiza un ciclo `for` para la iteración de cada tiempo del sistema. Se hace un barrido sobre las dos coordenadas `x = XM[i], y = YM[j]` y se calcula el valor del campo de las dos fuentes `Z[-j,i] = Z_amplitud(x, y, x01, y01, A01, t) + Z_amplitud(x, y, x02, y02, A02, t)`. Usando la función `plt.savefig(fname)` de *Matplotlib* , se guarda la grafica del campo en el tiempo actual, con el nombre `'_tmp-%03d.png' %k` donde `k` es un contador entero. Una vez guardada la gráfica, se cierra el entorno gráfico actual con la función `plt.close()` y se procede a calcular el siguiente tiempo.

```
os.system("ffmpeg -f image2 -i _tmp-%03d.png  video.mpg")
os.system('rm -rf *.png')
```

Finalmente se realiza el video a partir del programa `ffmpeg`. El formato de uso puede ser consultado en la página oficial del proyecto [1], aunque los parámetros usados en este código son óptimos para la generación de videos en *Python* .

[1] `http://www.ffmpeg.org/`

4.2. Ejercicios

Ejercicio 1 $(\star)$

Fuente puntual con reflexión
Considere el mismo sistema de la demostración 1, pero con una fuente puntual.

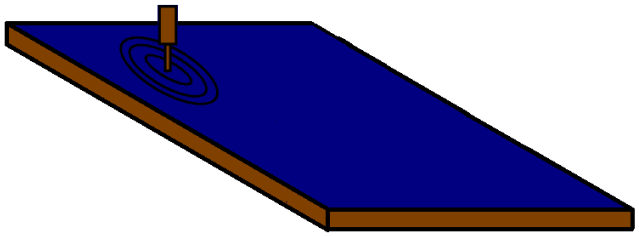


Figura 4.3: Reflexión en paredes.

A partir de este sistema, hacer un código que realice un video donde se muestre el patrón obtenido en el tiempo debido a la reflexión en las paredes de la caja.

Hint: para simular la reflexión de las ondas, puede considerar fuentes ficticias en el exterior de la caja que tengan posiciones simétricas respecto a las fuentes originales.

Ejercicio 2 $(\star)$

Refracción de sonido
Considere una onda sonora monocromática (sinusoidal) que se propaga en el aire, usando la librería `AudioLib` [2] simule el audio que sería percibido por un observador en el mismo medioo (aire) y otro audio por un observador que está sumergido en una piscina (agua). Es decir, simule el cambio que sufre el sonido por refracción en el agua.

Hint: todos los valores de los parámetros son de libre elección, la única condición es coherencia física en el resultado. La función `savewav` de `AudioLib` puede ser de utilidad (en ipython, después de importar la librería y crear un objeto `audio`, escribir `<object_name>.savewav?`).

[2] ver ejemplo de uso en demostración 2 del capítulo 3 (Efecto Dopler)

Ejercicio 3 $(\star)$

Simulación de trayectoria de luz en una lente
Considere un conjunto de N rayos de luz que atraviesan una de las lentes de la siguiente figura.

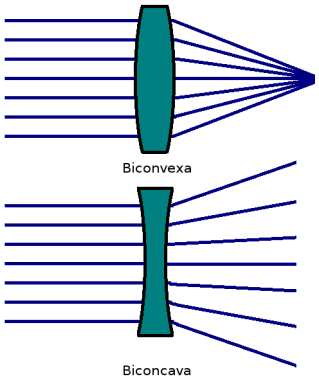


Figura 4.4: Efecto de una lente en haces de luz.

Realice un código que grafique (2D) la desviación de los haces de luz para cada uno de los dos casos (El usario del código debería poder escoger cual de los dos lentes usar). Los parámetros físicos son de libre elección, incluyendo el número de haces iniciales y la separación entre ellos.

Hint: La forma de los lentes puede omitirse en el gráfico, tan solo se puede ilustrar como una línea recta, o una línea divisoria que marque el punto donde los rayos se deflectan.[3]

[3]Un ejemplo algo más complejo que ilustra la trayetoria de haces en lentes puede encontrarse en `https://raw.github.com/sbustamante/Computacional-OscilacionesOndas/master/codigos/lens.py`

Ejercicio 4 $(\star)$

Refracción de fuente puntual.
Considere una fuente puntual en un medio con índice de refracción n_1 y un medio adyacente con índice n_2, de tal forma que la interfaz entre ambos medios es plana.

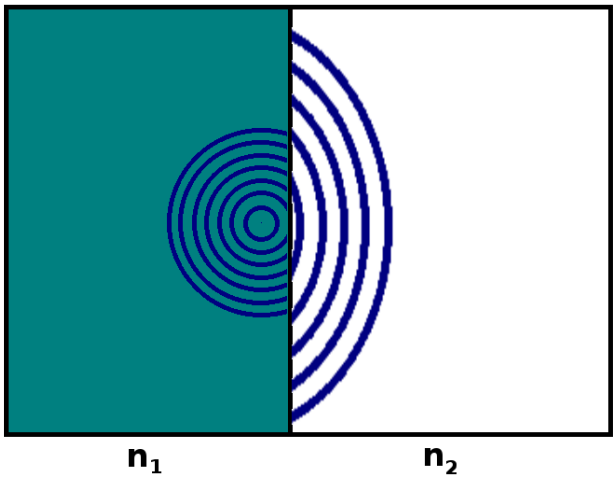


Figura 4.5: Refracción en un medio.

A partir de este sistema, hacer un código que realice un video donde se muestre el patrón obtenido en el tiempo debido a la refracción de un medio a otro.

Hint: para simular la refracción de las ondas, puede considerar una fuente ficticia que produzca el patrón refractado, pero solo evaluarlo en el medio 2.

Ejercicio 5 $(\star\star)$

Refracción Mecánica
El fenómeno de refracción puede ser demostrado de manera totalmente mecánica y se presenta sistemas como el descrito en la siguiente figura

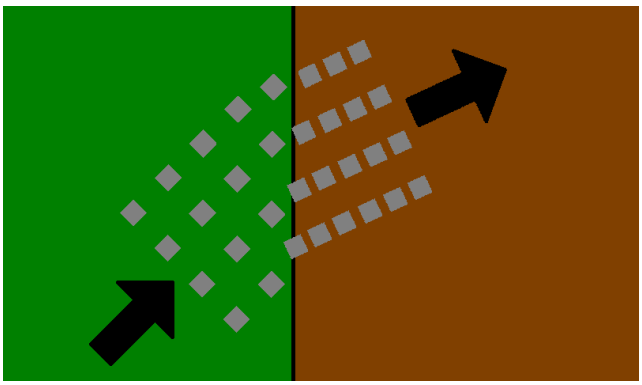


Figura 4.6: Refracción mecánica.

Considere un pelotón de soldados marchando de forma sincronizada, de tal forma que siempre están alineados en frentes. El pelotón tiene una disposición de $N \times M$ soldados. Inicialmente están marchando en un terreno poco agreste (césped) y el pelotón se mueve a una velocidad constante v_0. Más adelante en el camino, se encuentra un terreno pantanoso por el cual es más difícil transitar y el pelotón se mueve a una velocidad menor v_1. Si los soldados entran al terreno pantanoso formando un ángulo θ con la normal de la interfaz de los terrenos. Usando solo la condición de mantener los frentes alineados, puede demostrarse que el pelotón cambiará de orientación.

Basado en esta situación, realice un código donde se ilustre en una animación 3D este efecto.

Hint: Por simplicidad, los soldados pueden ser tomados como cubos. Puede usar la librería *MayaVi2* o *Vpython*.