



UNIVERSIDAD DE MENDOZA
FACULTAD DE INGENIERÍA
Sub-Sede San Rafael

Modelos y Simulación
Apuntes de Cátedra

Ing. Córdoba Diego

San Rafael - Noviembre de 2022

v0.71

MODELOS Y SIMULACIÓN - *Apuntes de cátedra*

Ing. Diego Córdoba

Esta obra está bajo una Licencia **Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional** (CC BY-NC-SA 4.0 Internacional). No puede usar este archivo excepto en conformidad con la Licencia. Puede obtener una copia de la Licencia en <https://creativecommons.org/licenses/by-nc-sa/4.0/>.

Este libro fue escrito en L^AT_EX, un sistema de preparación de documentos libre.



Copyright CC BY-NC-SA 2022 Diego Córdoba



Índice general

0. Introducción	1
0.1. TO-DO	3
1. Modelado de fenómenos [TODO]	4
1.1. Modelos matemáticos y computacionales	4
1.2. Etapas del modelado	6
1.2.1. Elementos de los modelos	7
1.2.2. Etapas del modelado	7
1.3. Validación de los modelos	10
1.4. Desarrollo de modelos simples	10
1.4.1. Influencia de la estructura y parámetros	10
1.4.2. Variables y tasas de cambio	10
1.4.3. Sistemas de segundo orden	10
1.4.4. Sistemas no lineales simples	10
2. Sistema de bicicletas compartidas	11
2.1. Modelado del sistema	11
2.2. Modelado iterativo	17
2.3. Introduciendo métricas	19
2.4. Barrido de parámetros	21
2.5. Notas finales del modelado	24
3. Modelo Poblacional	26
3.1. Crecimiento poblacional lineal	28
3.2. Simulación y graficación	28
3.3. Modelando el crecimiento proporcional	29
3.4. Modelo de crecimiento proporcional	32
3.5. Creando una función de <i>step</i>	34
3.6. Utilizando el parámetro α	35
3.7. Modelando el crecimiento cuadrático	36

3.8. Función cuadrática de población	37
3.9. Predicciones y proyecciones	40
3.9.1. Calculando proyecciones: otra aproximación	44
3.10. Análisis matemático	49
3.10.1. Relaciones recurrentes	49
3.10.2. Ecuaciones Diferenciales	51
3.10.3. Análisis vs simulación	54
3.10.4. Ecuaciones diferenciales con sympy	56
3.10.5. Ecuación logística	56
4. Modelo Epidemiológico	64
4.1. El modelo SIR	65
4.2. Las ecuaciones SIR	66
4.3. Implementación	67
4.4. La función de actualización (step)	68
4.5. Corriendo la simulación	69
4.6. Recolectando los resultados	70
4.7. Optimizando	73
4.7.1. Métricas	74
4.7.2. Inmunización	74
4.8. ¿Nos lavamos las manos?	77
4.9. Optimización	81
4.10. Barriendo dos parámetros	83
4.10.1. Barriendo Beta	83
4.10.2. Barriendo gamma	85
4.10.3. Usando SweepFrame	85
4.11. Análisis	88
4.11.1. Explorando los resultados	89
4.12. Número de contacto (Beta/Gamma)	92
4.13. Análisis y simulación	95
4.13.1. Estimando el número de contacto	96
5. Osciladores	98
5.1. Método de Euler	98
5.1.1. odeint: resolviendo ODE's con Python	100
5.2. Sistemas no lineales de segundo orden	101
5.3. Osciladores	102
5.4. Solución numérica	104
5.5. Una solución mágica: Euler-Cromer	108
5.6. Agregando amortiguación y fuerza externa	112
5.7. Amortiguación lineal	115
5.8. Usando fricción como amortiguación	118

5.9. Oscilador de Van der Pol	121
5.10. Oscilador caótico de Duffing	126
5.10.1. Fricción y amortiguación	128
5.10.2. Transición al Caos	129
5.10.3. Pequeños cambios en la entrada	132
5.10.4. Secciones de Poincaré	134
5.10.5. Función potencial	138
6. Ecuaciones de Lotka–Volterra	140
6.1. Mejorando el modelo	144
7. Control y Aptitud de modelos	148
7.1. Fuentes de la incertidumbre	149
7.2. Análisis de Incertidumbre	151
7.3. Análisis de sensibilidad	152
7.4. Pasos del análisis de incertidumbre	152
7.4.1. Caracterización de las entradas	153
7.4.2. Ejecución del modelo	153
7.4.3. Caracterización de las salidas	153
7.5. Método de Monte Carlo	153
7.5.1. Muestreo estratificado	158
8. Bibliografía de consulta	161

CAPÍTULO 0

Introducción

El modelado de fenómenos de la vida real no es una tarea sencilla, y requiere de un trabajo muchas veces interdisciplinar que se valga de las ventajas de la computación para facilitar la comprensión de los sistemas reales.

El modelado y simulación no se trata únicamente de una disciplina matemática o física, es una disciplina que involucra diferentes habilidades, como la abstracción, el análisis, la simulación y validación, habilidades que son fundamentales en muchos campos de estudio tales como la ingeniería, la medicina, las ciencias sociales, la economía, la física, entre tantos otros.

Una muy buena manera de aprender Modelos y Simulación es, efectivamente, modelando fenómenos, cambiando parámetros, analizando los cambios de comportamiento del modelo, generando modelos alternativos, introduciendo nuevas variables, analizando su sensibilidad, comparando los resultados con resultados analíticos, etc. En fin, una buena forma de aprender esta cátedra es llevándola a la práctica, generando códigos que resuelvan problemas, y volcando los conocimientos matemáticos necesarios con la intención de utilizarlos para mejorar los algoritmos y la precisión de las soluciones.

En el presente apunte se utilizará el lenguaje de programación Python para llevar los modelos a código, y se hará uso de varios módulos y librerías de programación científica, entre los que se cuentan:

- **NumPy** para cálculos básicos (ver <https://www.numpy.org/>).
- **SciPy** para programación científica (ver <https://www.scipy.org/>).
- **Matplotlib** para visualización y gráficas (ver <https://matplotlib.org/>).
- **Pandas** para trabajar con volúmenes de datos (ver <https://pandas.pydata.org/>).
- **Sympy** para computación simbólica (ver <https://www.sympy.org>).
- **Pint** para administrar unidades de medida (ver <https://pint.readthedocs.io>).

Este apunte está basado en libro *Modeling and Simulation in Python* de Allen B. Downey, al que se le han eliminado algunos capítulos, y se han agregado otros de autoría propia para adaptarlo a la cátedra Modelos y Simulación de 5to año de la carrera Ingeniería en Informática impartida por la Universidad de Mendoza (Sede San Rafael, Argentina).

Los primeros modelos a analizar son **modelos discretos**, tales como el modelo de bicicletas compartidas, y modelos poblacionales simples.

Luego se introducen **sistemas de primer orden**, es decir, aquellos representados por ecuaciones diferenciales de grado 1. Un ejemplo son los modelos de enfermedades infecciosas y modelos epidemiológicos.

Finalmente se introducen los **modelos de segundo orden**, tales como los sistemas oscilatorios como un péndulo, un sistema masa-resorte-amortiguador.

En todos los casos se realiza el **modelado** del fenómeno utilizando **técnicas computacionales** y valiéndose de las ventajas operativas que brinda Python en este sentido, y se realiza su **comparativa con las soluciones tradicionales** provistas por el **análisis matemático**, distinguiendo ventajas y desventajas de cada metodología, y cómo es posible lograr una integración para obtener mejores resultados.

Se analizan modelos clásicos como la ecuación logística, diagramas de Feigenbaum, modelos caóticos, comportamientos fractales, interpretación de parámetros y de resultados,

y se analiza visualmente, y mediante distintos tipos de representaciones, la información obtenida por simulación.

Finalmente se estudian técnicas de **validación y de control de aptitud** de los modelos desarrollados. Se estudia el origen del error o **incertidumbre** entre los modelos y los sistemas reales estudiados, la **sensibilidad** de parámetros y variables, y la generación de datos pseudo-aleatorios basados en una distribución de probabilidad mediante el método de **Montecarlo**, con la intención de facilitar la generación de valores de entrada a nuestros modelos.

0.1 TO-DO

Este libro representa una versión inicial de los temas de cátedra, con varios faltantes, ya que se fue desarrollando durante el cursado de Modelos y Simulación en la Facultad de Ingeniería de la Sede San Rafael de la Universidad de Mendoza, Argentina.

Al estar en una versión preliminar, se han agregado en títulos de capítulos y dentro del texto notas TO-DO, pendientes que se irán completando durante en un futuro cercano. Entre los principales se cuentan. Los capítulos que no están presentes igual se dictaron durante el cursado 2020 de la materia, pero no se han plasmado en el libro todavía.

- Escribir el **Capítulo 1**: Modelado de fenómenos, una introducción a la modelación de fenómenos físicos, naturales y poblacionales, y cómo plantear soluciones computacionales y matemáticas.
- **Regenerar algunas gráficas** y diagramas tomados de los apuntes de Enrique Puliafito y no volcados a python por el momento.
- **Reordenar el repositorio GIT** de la materia https://gitlab.com/d1cor/um_mys, con los códigos utilizados durante las clases.
- Utilizar **Jupyter Notebook** para facilitar la comprensión de algunos temas.
- Generar **bibliografía y referencias** en normas APA.
- Rediseñar y mejorar las prácticas propuestas de la materia.
- Mejorar el diseño y formato **LATEX** de este libro.

CAPÍTULO 1

Modelado de fenómenos [TODO]

1.1 Modelos matemáticos y computacionales

El análisis de un sistemas intenta mejorar la comprensión de su comportamiento, con la intención de optimizar su control, o facilitar la toma de decisiones. Analizar un sistema significa describirlo, conocer los procesos que lo conforman, con la intención de predecir o inferir su comportamiento bajo diversas condiciones.

Muchas veces una descripción estadística del sistema resulta muy limitada dado que representa solamente comportamientos históricos del mismo, dificultando o imposibilitando la inferencia de comportamientos futuros bajo otras condiciones. Por esto es importante comprender el sistema a través de modelos que intenten representarlo lo más fielmente posible, describan su comportamiento presente, y permita inferir su comportamiento bajo condiciones iniciales diferentes.

Aquí se debe aclarar que todo modelo de un sistema real será una simplificación, por lo que es muy importante establecer claramente los objetivos y alcances del modelo. La Figura 1 muestra el proceso de análisis de un sistema, y desarrollo de un modelo asociado.

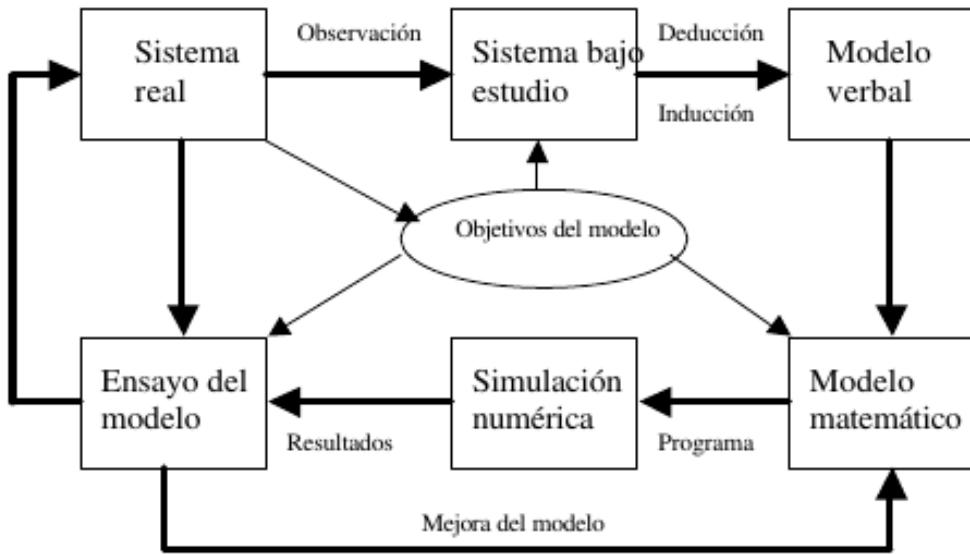


Figura 1: Desarrollo de modelos de sistemas

La construcción de modelos es motivada muchas veces de la necesidad de explicar **sries temporales de datos**. Una serie temporal es un conjunto de datos que representan la variación de una variable de estudio con el transcurso del tiempo. Por ejemplo, podría considerarse la variación de la cosecha de uva en los últimos 20 años, o la evolución de las ventas de computadoras en una ciudad o país.

Estas series a su vez tienen características que se desea estudiar, como tendencias a largo plazo variabilidad estacional, o variaciones irregulares. Las variaciones irregulares representan cambios aleatorios en el desarrollo de la serie temporal, que no pueden ser explicados por variaciones estacionales o tendencias a largo plazo. Por ejemplo, una inundación o una ola heladas podrían ser efectos irregulares sobre la serie temporal de cosecha de uva.

Un motivo para estudiar series temporales podría ser la estimación de tendencias, aislando componentes aleatorios y estacionales. También podría estudiarse la variación estacional y no las tendencias a largo plazo. Otro motivo podría ser la inferencia o predicción de valores futuros de la serie, ya que una de las razones principales para el modelado es la capacidad que tiene una simulación para realizar pronósticos.

Si la serie es univariable, como la cosecha de uva, podría el lector simplemente argumentar que la predicción de valores futuros podría simplemente estimarse mediante un

modelo de regresión. No obstante, estos modelos no permiten estudiar las variaciones estacionales, por ejemplo, por lo que un modelo para simulación podría ser una solución más acertada. Muchas veces es necesario incluir variables adicionales que ayuden a explicar series temporales, por lo que en la medida en que la serie represente una magnitud más compleja, el modelo será más complejo también. Por ejemplo, podría explicarse el nivel de empleo de una región, sus variaciones, por medio de otras variables intermedias como la productividad, el volumen de ventas, y los factores aleatorios.

La productividad puede ser difícil de calcular en este ejemplo, por lo que podría aproximarse con una variable de tendencia de los valores de productividad actuales.

1.2 Etapas del modelado

El problema del modelado puede descomponerse en cuatro subproblemas:

1. **El problema de la representación:** plantea cómo debe realizarse el modelado respecto al modelo matemático. Se deberá elegir un modelo estático o dinámico, lineal o no lineal, determinístico o estocástico, continuo o discreto, en el dominio del tiempo o de la frecuencia, etc. Se deberán seleccionar elementos como ecuaciones algebraicas, diferenciales ordinarias, funciones de transferencia, respuestas impulsivas, variables de estado, etc.
2. **El problema de la medición:** consiste en determinar qué cantidades físicas deben medirse y cómo, ya que no todas las variables implicadas pueden medirse. Normalmente se usan sensores para medir estas cantidades físicas. Aunque estos sensores sean muy precisos, igual introducen errores que deben tenerse en cuenta en el modelado.
3. **El problema de la estimación:** en este punto surge la necesidad de estimar aquellas cantidades que no pueden medirse directamente, y suele hacerse a partir de aquellas que sí se miden.
4. **El problema de la validación:** este problema consiste en demostrar la confianza del modelo. Entre otras técnicas puede compararse el resultado entregado por el modelo con resultados históricos medidos en la realidad.

1.2.1 Elementos de los modelos

Se pueden distinguir elementos comunes a todos los sistemas dinámicos. Los más significativos son:

- **Parámetros:** Son magnitudes que permanecen constantes durante la observación del sistema. En muchos casos son constantes de la misma naturaleza.
- **Efectos ambientales:** Son variables que influyen al sistema desde el exterior o entorno del sistema, pero no son influidas por el sistema. También llamadas variables exógenas o externas.
- **Variables de estado:** También se las conoce por elementos de memoria o almacenamiento del sistema. El valor instantáneo de estas variables describen el estado del sistema completamente en un momento determinado. En muchos casos estas pueden medirse directamente, pero en otros casos son muy difícil de medir y sólo pueden estimarse.
- **Valores iniciales de las variables de estado:** Describen el punto inicial de cada variable. Su conocimiento es crítico ya que determinan el desarrollo del sistema. Aún manteniendo todos los demás elementos constantes, éstas fijan las condiciones de contorno del sistema.
- **Tasa de cambio de las variables de estado:** Estas determinan la velocidad de cambio, crecimiento o disminución, de las variables bajo estudio y por unidad de tiempo. Su conocimiento es indispensable para determinar los estados del sistema.
- **Variables intermedias:** Son variables que surgen como consecuencia del desarrollo del modelo, pero que no son necesariamente salidas del sistema.

1.2.2 Etapas del modelado

El proceso de modelado de un sistema real sigue, en general, estos pasos:

1. **Descripción del problema:** El primer paso es realizar una descripción lo más exacta posible del sistema real.

2. **Objetivo del modelo:** Existen muchas posibilidades para describir un modelo. Muchas veces es contraproducente tratar de duplicar el sistema elemento por elemento, debido a la complejidad del sistema real. Se debe primeramente realizar una simplificación del modelo atendiendo a los objetivos de estudio.
3. **Entorno del sistema:** Una vez descrito el modelo y especificado su objetivo, es posible definir los límites del sistemas, es decir, qué variables son externas y cuáles son internas al sistema.
4. **Modelo verbal:** El primer paso en el desarrollo de un modelo debería ser la descripción de sus elementos, sus funciones y sus relaciones estructurales, en un lenguaje sencillo. Esta descripción verbal del modelo debe ser entendida por todos los integrantes del equipo de modelación interdisciplinaria, de manera que todos puedan participar en la discusión, la reformulación y la comprobación del modelo.
5. **Elementos del sistema:** En el modelo verbal ya aparecen los principales elementos del sistema con sus categoría correspondientes (parámetros externos, parámetros del sistema, variación temporal de las variables del entorno, condiciones iniciales, tasas de variaciones y variables intermedias).
6. **Relaciones estructurales:** Las relaciones y uniones estructurales entre los elementos deben incorporarse al modelo verbal. En cuanto se conocen estas relaciones, ya se pueden realizar el diagrama del sistema.
7. **Diagrama de efectos:** Este diagrama contiene los elementos del sistemas junto con sus relaciones estructurales. Este diagrama es la base para la elaboración del diagrama de simulación.
8. **Relaciones funcionales:** El diagrama de efectos, muestra los elementos con sus relaciones, pero no muestra las funciones propias. Estas relaciones funcionales pueden estar expresadas como funciones matemáticas, estadísticas o en forma de tablas.
9. **Cuantificación:** El desarrollo de un programa de simulación nos exige una cuantificación de las variables en juego. En especial deben considerarse las condiciones iniciales de las variables del sistema.

10. **Diagrama de simulación:** Una vez que se conocen los elementos, las funciones, la estructura del sistema y todos los parámetros, puede realizarse el diagrama de simulación.
11. **Programación:** El modelo de simulación, puede desarrollarse en diversos lenguajes de programación.
12. **Prueba, corroboración y validación del modelo:** Una vez que el modelo corre por primera vez, debe comprobarse la validez del mismo. Este paso debe hacerse exhaustivamente para descubrir posibles errores en la formulación. Normalmente se usan datos históricos, si existen, para comprobar la validez del modelo. De encontrarse alguna dificultad o no conformidad, este lazo debe volver a revisar todos los pasos anteriores, y deberá rehacerse tantas veces como sea necesario hasta obtenerse los márgenes de incertidumbres impuestos en el objeto del modelo. Para la validación del modelo es útil comprobar los resultados del modelo con otros modelos o datos, pero generados a través de otra metodología, otra instrumentación, etc.
13. **Simulación y aplicación:** Una vez probado el modelo, se ingresan todos los parámetros iniciales, tiempos, condiciones de entorno. A través de numerosas corridas del modelo, comienza a entenderse el sistema. Generalmente la simulación tiene por objeto comprobar otras condiciones que al momento pudieron no haberse dado. Debe tenerse siempre presente el objeto del modelo y su validez. No es apropiado extender las conclusiones más allá de la validez del modelo.

Estos pasos pueden verse reflejados en la Fig. 2.

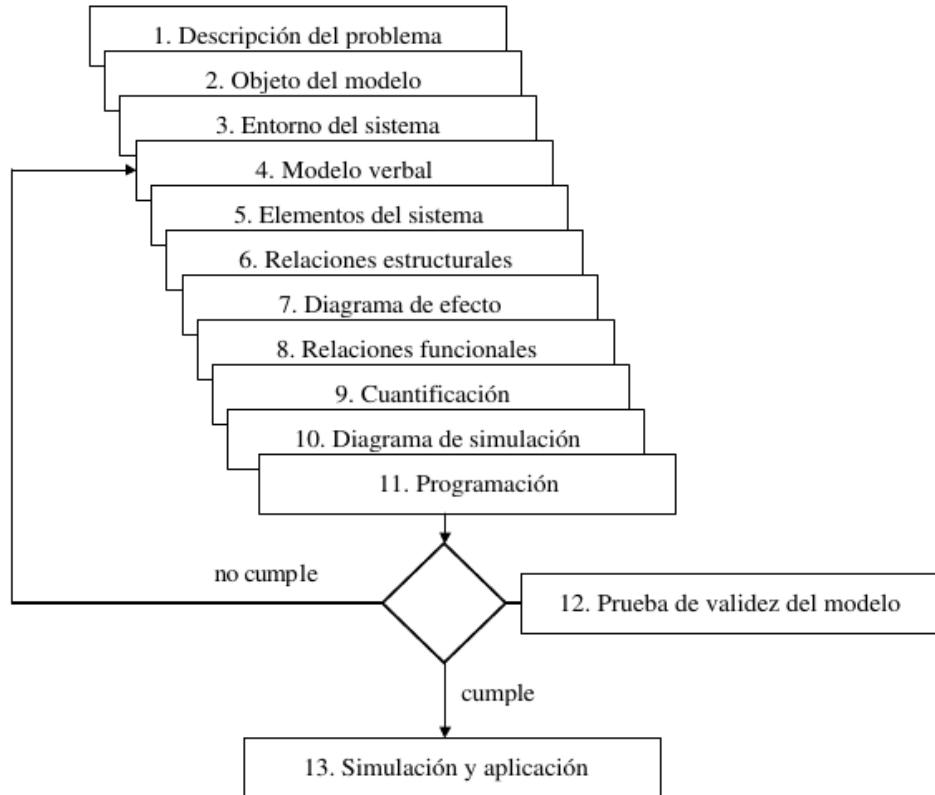


Figura 2: Etapas del modelado

1.3 Validación de los modelos

1.4 Desarrollo de modelos simples

1.4.1 Influencia de la estructura y parámetros

1.4.2 Variables y tasas de cambio

1.4.3 Sistemas de segundo orden

1.4.4 Sistemas no lineales simples

todo

CAPÍTULO 2

Sistema de bicicletas compartidas

2.1 Modelado del sistema

En este capítulo vamos a introducir el modelado de un sistema simple para comenzar a entender cómo realizar estas tareas en Python utilizando el módulo ModSimPy.

Imaginemos que tenemos estudiantes en la Universidad de Mendoza, que necesitan trasladarse al comedor universitario, a unas cuadras de distancia, y viceversa, estudiantes que se encuentran en el comedor y necesitan volver al edificio de la Universidad.

Supongamos que las autoridades proveen un bicicletero en las instalaciones de la Universidad, y otro en el comedor, con 10 bicicletas y 2 bicicletas respectivamente. Los estudiantes que necesiten trasladarse podrán tomar una bicicleta del bicicletero y utilizarla. Este evento hará que la cantidad de bicicletas en dicho bicicletero se reduzca en una unidad, y se incremente en el otro bicicletero también en una unidad.

En nuestras simulaciones vamos a registrar los cambios en cantidad de bicicletas en ambos bicicleteros, es decir, dichas cantidades serán las **variables de estado** a considerar.

Para esto necesitamos crear un objeto de tipo `State` del módulo `modsim`.

```
import modsim as ms
```

```
bikeshare = ms.State(um=10, comedor=2)
```

Las variables de estado, `um` y `comedor` representan el numero de bicicletas en cada uno de los bicicleteros respectivamente. Los valores iniciales serán 10 y 2. Este objeto lo hemos almacenado en la variable `bikeshare`.

Podemos actualizar el estado inicial restando a una de las variables, por ejemplo, una bicicleta, y sumándosela a la otra. Definamos una función que .envíeuna bicicleta desde la UM al Comedor y otra que realice el trabajo inverso:

```
def bike_to_comedor():
    print("Enviando una bicicleta al comedor")
    bikeshare.um -=1
    bikeshare.comedor +=1

def bike_to_um():
    print("Enviando una bicicleta a la UM")
    bikeshare.um +=1
    bikeshare.comedor -=1
```

Imaginemos ahora que aleatoriamente, en un instante de tiempo, un estudiante con, por ejemplo, 50 % de probabilidad, viajará desde la UM al comedor, y uno con una probabilidad del 30 % lo hará en sentido contrario. Podríamos definir una función de actualización, o *step*, que calcule valores aleatorios con estas probabilidades, y ejecute las funciones definidas anteriormente si da la probabilidad.

Para esto utilizaremos la función `flip()` del módulo `modsim`, simula el lanzamiento de una moneda en la que uno de los lados es verdadero (`True`) y el otro falso (`false`). La función `flip()` retornará verdadero con la probabilidad pasada por argumento.

```
def step():
    if flip(0.5):
        bike_to_comedor()
    if flip(0.3):
        bike_to_um()
```

Podemos correr un paso de nuestra simulación simplemente ejecutando esta función:

```
step()
```

Ahora... esto se vería mejor sin las probabilidades hardcodeadas, cierto? Utilicemos parámetros mejor:

```
def step(state, p1, p2):
    """Simulate one minute of time.

    state: bikeshare State object
    p1: probability of an um->comedor customer arrival
    p2: probability of a comedor->um customer arrival
    """
    if flip(p1):
        bike_to_comedor(state)

    if flip(p2):
        bike_to_um(state)
```

Entonces ahora podremos correr el paso de simulación usando:

```
step(0.5, 0.3)
```

Y si simulamos varios ciclos? Por ejemplo, uno por minuto para 10 minutos? Claro, podemos hacerlo de esta forma:

```
for i in range(10):
    step(0.5, 0.3)
```

Pero obtendríamos únicamente el último estado del sistema luego de los 10 ciclos, es decir, la cantidad de bicicletas resultante en cada locación luego de la simulación.

Nuestra intención a la hora de simular debe ser analizar todo el proceso y el comportamiento de nuestro modelo a lo largo del tiempo, por lo que sería muy importante almacenar los resultados intermedios del sistema.

Imaginemos que en esta oportunidad nos interesa conocer el comportamiento del modelo en la universidad, es decir, los cambios de estado en el bicicletero de la UM. Podríamos utilizar una variable de tipo `TimeSeries()`, que puede contener secuencias de estampas de tiempo y su correspondiente valor, es decir, la cantidad de bicicletas en la UM para cada minuto.

```

resultado = ms.TimeSeries()
for i in range(10):
    step(0.5, 0.3)
    resultado[i] = bikeshare.um

```

La TimeSerie resultado tendrá algo similar a esto:

```

1   11
2   11
3   12
4   11
5   11
6   11
7   12
8   12
9   12

```

Y ya que estamos, podemos graficar estos resultados. En el siguiente código se ve la función `plot` de `modsim`, utilizada para generar el gráfico de los valores, y la función `decorate`, también de `modsim`, para agregarle las etiquetas y leyendas a dicho gráfico.

```

plot(resultado, label="UM")
decorate(title="UM-Comedor - Bicicletas compartidas", xlabel="Tiempo
(minutos)", ylabel="Número de bicicletas en la UM")
savefig("/tmp/bikeshare_um.jpg")

```

Esto nos generará una gráfica similar a la de la Fig. 3.

Cabe aclarar que el objeto `TimeSeries` `resultado` posee varios métodos para calcular medidas estadísticas, lo que puede servir si queremos caracterizar los resultados. Por ejemplo, podemos obtener la media utilizando `resultado.mean()`, y varias medidas utilizando `resultado.describe()`.

Para analizar los valores de estado de la UM y del Comedor, podríamos crear dos `TimeSeries`, uno para cada locación, y luego ejecutar una cierta cantidad de pasos de simulación, supongamos, 60 minutos, e ir almacenando en dichas variables los resultados. Veamos el siguiente código:

```

pyplot.clf()
um = TimeSeries()

```

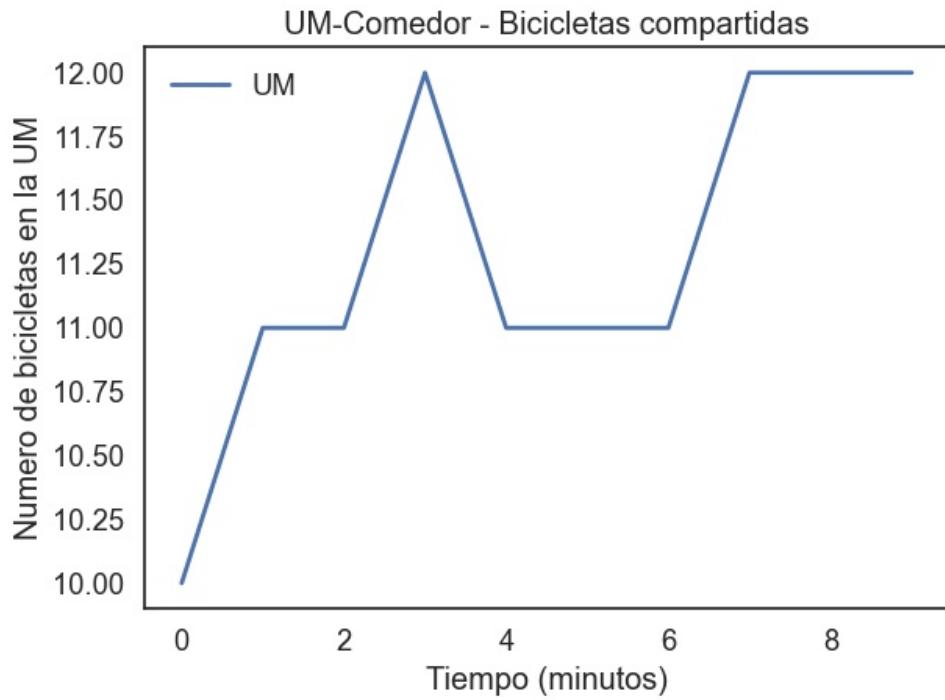


Figura 3: **Bicicletas compartidas - UM**

```

comedor = TimeSeries()

for i in range(60):
    step(bikeshare, 0.3, 0.2)
    um[i] = bikeshare.um
    comedor[i] = bikeshare.comedor

plot(um, label='UM')
plot(comedor, label='Comedor')
decorate(title='Bicicletas compartidas UM-Comedor', xlabel='Time
step (minutos)', ylabel='Number of bicicletas')

savefig("/tmp/bikeshare_umcomedor.jpg")

```

La primer línea limpia el gráfico para que otros *ploteos* anteriores no se vean reflejados en este nuevo gráfico. Definimos dos objetos `TimeSeries`, uno para la UM y otro para el Comedor, y luego ciclamos 60 veces con probabilidades del 30 % y del 20 % respectivamente, almacenando los resultados intermedios. Luego, finalmente creamos la gráfica y la guardamos en un archivo. En este caso, los resultados se ven en la Fig. 4.

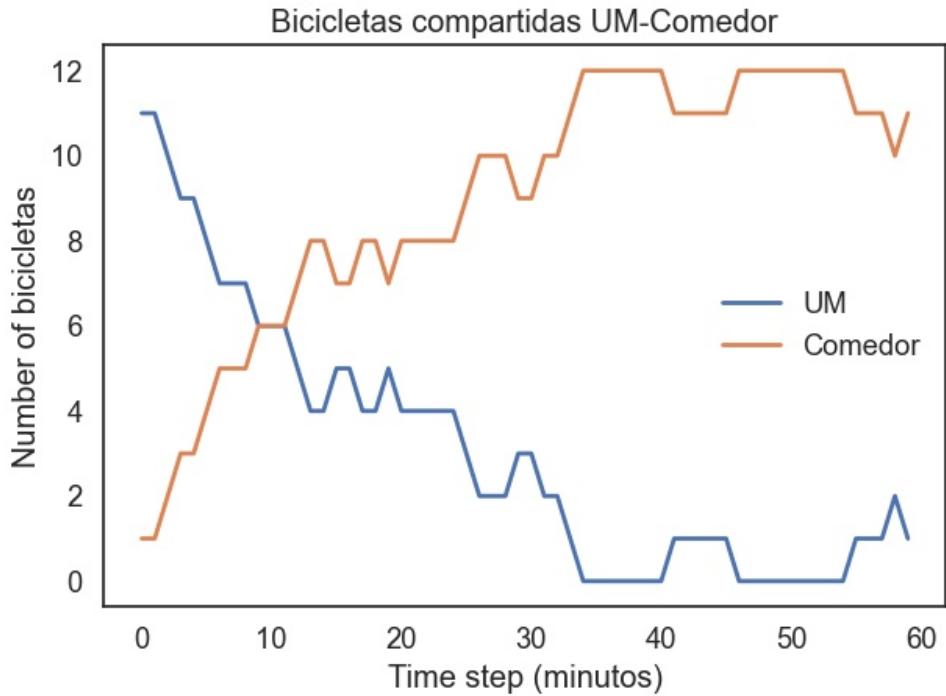


Figura 4: **Bicicletas compartidas UM-Comedor**

Ahora, para simplificar las simulaciones, creemos una función de simulación llamada, por ejemplo, `run_simulation()`, que reciba por argumento los parámetros utilizados, `p1` y `p2` como las probabilidades de que una bicicleta se mueva de una ubicación a la otra, y `steps` como la cantidad de pasos de simulación que queremos ejecutar.

Otro parámetro que deberíamos pasarle es la variable de estado `State`, de modo que no necesitemos registrarla como global. Esto va a requerir modificar también nuestra función `step()` y las funciones `bike_to_um()` y `bike_to_comedor()`, ya que también deberían recibir esta variable `State` para poder actualizar su estado.

Veamos las nuevas versiones:

```
def step(state, p1, p2):
    if flip(p1):
        bike_to_comedor(state)

    if flip(p2):
        bike_to_um(state)

def bike_to_comedor(state):
```

```

state.um -= 1
state.comedor += 1

def bike_to_um(state):
    state.comedor -= 1
    state.um += 1

def run_simulation(state, p1, p2, steps):
    results = TimeSeries()
    for i in range(steps):
        step(state, p1, p2)
        results[i] = state.um

    plot(results, label='um')
    decorate(title='um-comedor Bikeshare',
             xlabel='Tiempo (minutos)',
             ylabel='Número de bicicletas')

    savefig('/tmp/grafico.jpg')

```

2.2 Modelado iterativo

Partamos de una premisa: "Todos los modelos están mal, pero algunos modelos están peores que otros".

Como hemos visto, partimos de un modelo inicial muy simple, luego fuimos mejorando nuestro código, añadiendo parámetros, y dividiendo en funciones para facilitar las simulaciones. La intención es la de identificar pequeños problemas en cada instancia de nuestro modelado, e ir corrigiendo dichas anomalías para generar un modelo mejor. A esto se lo denomina **modelado iterativo**.

Teniendo esto en mente, identifiquemos algunas imperfecciones de nuestro modelo de bicicletas compartidas:

- El modelo asume que el comportamiento es igual en cada minuto del día, pero en la realidad no es lo mismo a la noche que al mediodía, ni los fines de semana.

- El modelo no considera los estudiantes que están viajando, el cambio es instantáneo.
- El modelo no considera la cantidad de bicis que quedan antes de moverlas de una locación a otra, por lo que puede quedar un número negativo de bicicletas en algún bicicletero.

Algunas de estas decisiones pueden ser razonables para paliar estos inconvenientes:

- El primer problema podría solucionarse simulando para una hora particular del día.
- El segundo no es muy realista, pero dependiendo de lo que necesitemos analizar, puede que no cambie el resultado. Es decir, si lo que necesitamos analizar es la cantidad de bicicletas disponibles en cada bicicletero, este modelo está bien. Ahora, si lo que queremos es estudiar el movimiento de estudiantes entre la UM y el comedor, cuánto demoran en llegar, y cuántas bicicletas están ^{en} viaje no deberíamos contarlas en ninguna de las dos locaciones, deberíamos mejorar el modelo.
- El tercer problema sí puede complicarnos nuestro estudio, pero es relativamente sencillo de solucionar.

Podríamos solucionar el problema de las bicicletas negativas podríamos verificar cuántas bicicletas quedan en un bicicletero antes de "mover" una de ellas hacia el otro. Esto podría realizarse de esta forma:

```
def bike_to_comedor(state):
    if state.um > 0:
        state.um -= 1
        state.comedor += 1

def bike_to_um(state):
    if state.comedor > 0:
        state.comedor -= 1
        state.um += 1
```

Si ahora corremos una simulación, por ejemplo, para 60 minutos, con alta probabilidad de que un estudiante vaya de la UM al Comedor, tendremos una gráfica como la mostrada en la Fig. 5, en la que la cantidad de bicicletas nunca se hace negativa.

```

resultado = TimeSeries()
for i in range(60):
    step(bikeshare, 0.6, 0.2)
    resultado[i] = bikeshare.um

print(resultado.mean)
plot(resultado, label="UM")
decorate(title="UM-Comedor - Bicicletas compartidas", xlabel="Tiempo (minutos)", ylabel="Número de bicicletas en la UM")
savefig("/tmp/bikeshare_um_nonulo.jpg")

```

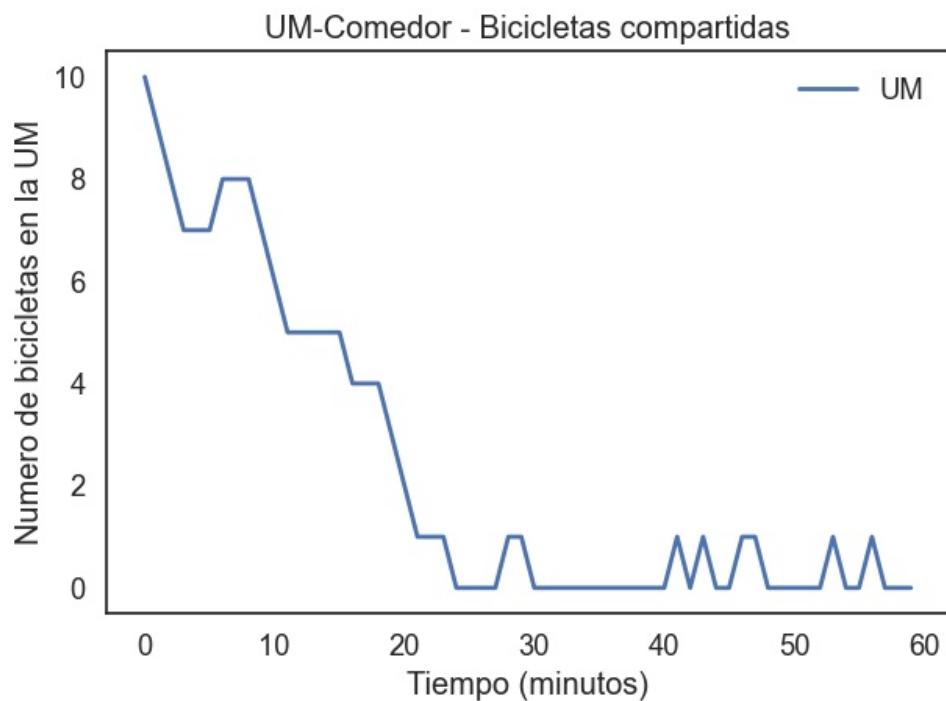


Figura 5: Límite de bicicletas negativas.

2.3 Introduciendo métricas

La llegada de bicicletas a cada una de las locaciones es aleatoria, por lo que este modelo se denomina **estocástico** o **probabilístico**. Esto quiere decir que cada vez que realicemos la simulación podríamos obtener valores distintos. Si en cada simulación los resultados son los mismos, se dice que el modelo es **determinístico**.

Supongamos que queremos predecir (o inferir) cuán bien funcionará un sistema de bicicletas compartidas para mejorar su diseño en la realidad, de modo que se adapte mejor al comportamiento de los estudiantes.

Primero debemos especificar qué significa cuán bueno "mejor.^{es} nuestro sistema. Desde el punto de vista de los alumnos, el sistema podría ser bueno en la medida en que el alumno encuentre al menos una bicicleta disponible cuando la necesite. Desde el punto de vista del dueño de la administración el sistema podría ser bueno cuando se minimizar el numero de alumnos que necesitan bicicletas y no tienen disponible, y a su vez, se maximice el uso de bicicletas, es decir, se reduzca la cantidad de bicicletas ociosas, lo cual permite también reducir costos iniciales y de mantenimiento.

Para poder tomar este tipo de decisiones es necesario recolectar información del modelo. Estas medidas estadísticas que podemos obtener se denominan **métricas**.

Veamos un ejemplo simple. Supongamos que estamos analizando el modelo desde el punto de vista de los estudiantes, y que un estudiante se encuentra disconforme con el sistema cuando necesita trasladarse en bicicleta, pero en el bicicletero no se encuentra ninguna disponible. Podríamos modificar nuestras funciones `bike_to_um()` y `bike_to_comedor()` de modo que contabilicen la cantidad de estudiantes disconformes al finalizar la simulación.

Para ello vamos a agregar dos nuevas variables en nuestro State `bikeshare`, una que identifique las veces que el bicicletero de la UM quedó vacío, `um_vacio`, y otra para el bicicletero del comedor, `comedor_vacio`.

```
def bike_to_comedor(state):
    if state.um > 0:
        state.um -= 1
        state.comedor += 1
    else:
        # no tenemos bicicletas
        state.um_vacio+=1

def bike_to_um(state):
    if state.comedor > 0:
```

```

state.comedor -= 1
state.um += 1
else:
    # no tenemos bicicletas
    state.comedor_vacio+=1

```

De esta manera necesitaremos redefinir los valores iniciales de las variables de estado especificando en 0 los valores iniciales de estas dos métricas.

```
bikeshare = State(um=10, comedor=2, um_vacio=0, comedor_vacio=0)
```

Y ahora, simplemente deberemos correr alguna simulación para analizar los resultados:

```

run_simulation(bikeshare, 0.6, 0.2, 30)

print(f"Um vacio: {bikeshare.um_vacio}")
print(f"Comedor vacio: {bikeshare.comedor_vacio}")

```

Con lo que tendremos una salida similar a esta:

```

Um vacio: 5
Comedor vacio: 0

```

Recordemos, se trata de un sistema estocástico, por lo que los resultados seguramente cambien entre simulaciones sucesivas.

Ejer-
cicio

2.4 Barrido de parámetros

Hemos visto cómo cuantificar el rendimiento del modelo utilizando métricas. Ahora surge la pregunta: ¿Cómo afectan los parámetros del modelo a dichas métricas? Por ejemplo, si la probabilidad de que un alumno requiera tomar una bicicleta en la UM cambia del 60 % del ejemplo anterior, al 80 %, ¿cómo afecta esto a las métricas de cantidad de usuarios disconformes en la UM?

Para analizar este punto primero vamos a reescribir la función `run_simulation()` de modo que almacene en la variable de estado los valores, y los retorne (en vez de graficarlos) para poder analizarlos con posterioridad.

```

def run_simulation(state, p1, p2, steps):
    for i in range(steps):
        step(state, p1, p2)

    return state

```

Con esto podríamos inicializar nuevamente nuestro bikeshare, y correr una simulación de 60 pasos de esta forma:

```

bikeshare = State(um=10, comedor=2, um_vacio=0, comedor_vacio=0)
state = run_simulation(bikeshare, 0.4, 0.2, 60)

```

La variable `state` que retorna la función `run_simulation()` contendrá el estado final del modelo en la simulación. Por ejemplo, podremos obtener los valores de bicicletas de la UM y del Comedor al terminal la simulación accediendo simplemente a `state.um` y `state.comedor` respectivamente. Lo mismo con los valores de alumnos disconformes.

Ahora analicemos la siguiente situación. Supongamos que podemos aumentar o reducir el tiempo de recreo de los alumnos. A recreos más largos se aumenta la probabilidad de que los alumnos viajen al comedor en bicicleta, y a recreos más cortos, se reduce. Intentemos responder a la siguiente pregunta: Sabiendo que podemos manipular la probabilidad de que un alumno tome una bicicleta en la UM para trasladarse al comedor, ¿cuál sería la probabilidad óptima para maximizar el uso de las bicicletas, y a la vez, minimizar la cantidad de alumnos disconformes?

Para responder esto deberíamos probar varias probabilidades `p1` y luego cambiar el tiempo de recreo según la probabilidad que nos resulte óptima.

Supongamos que queremos analizar las probabilidades `p1` con un paso del 10 %, es decir, con valores de 0.1, 0.2, 0.3, ..., 0.8, 0.9, 1.0. Podemos usar un objeto de tipo `linspace`, un arreglo NumPy modelado dentro de `modsim`.

```

p1_array = ms.linspace(0, 1, 11)

```

`linspace` con estos parámetros va a crear una lista de 0.0 a 1.0 dividida en 11 partes iguales, lo que hace que haya una diferencia de fija de 0.1 entre valor y valor (son 11 valores ya que estamos considerando al 0.0).

Ahora, podemos fijar el resto de los parámetros, por ejemplo, una probabilidad p_2 de que los alumnos se trasladen desde el comedor hasta la UM, en 0.2, y una cantidad de minutos de simulación en 60. Podemos calcular las simulaciones y mostrar la cantidad de alumnos disconformes en la UM como resultado de cada una.

```
p2 = 0.2

bikeshare = State(um=10, comedor=2, um_vacio=0, comedor_vacio=0)

for p1 in p1_array:
    state = run_simulation(state, p1, p2, 60)
    print(p1, state.um_vacio)
```

Esto nos entregará un resultado similar a este:

```
0.0 0
0.1 0
0.2 0
0.3 0
0.4 0
0.5 9
0.6 12
0.7 19
0.8 30
0.9 31
1.0 35
```

Aquí se ve, para cada una de las probabilidades de que un estudiante necesite trasladarse desde la UM hacia el Comedor, la cantidad de alumnos disconformes. Se ve claramente que en la medida que la probabilidad aumenta, es decir, más estudiantes vayan al comedor, mayor será la cantidad de estudiantes que no encuentren bicicleta disponible.

Ahora vayamos un poco más allá. Usemos un nuevo tipo de datos para almacenar los resultados: la SweepSeries de `modsim`. Esto nos permitirá, por ejemplo, graficar los datos obtenidos.

```
p2 = 0.2

bikeshare = State(um=10, comedor=2, um_vacio=0, comedor_vacio=0)
sweep = ms.SweepSeries()
```

```

for p1 in p1_array:
    state = run_simulation(state, p1, p2, 60)
    sweep[p1] = state.um_vacio

```

Mejoremos el código, creemos una función llamada `sweep_p1` que permita hacer el barrido del parámetro `p1` para un valor fijo de simulaciones y de `p2`.

```

def sweep_p1(bikeshare, p1_array, p2, steps):
    sweep = ms.SweepSeries()
    for p1 in p1_array:
        run_simulation(bikeshare, p1, p2, steps)
        sweep[p1] = bikeshare.um_vacio
        bikeshare.um_vacio = 0
    return sweep

```

Finalmente, podemos invocar a dicha función, y graficar los resultados que obtengamos:

```

p2 = 0.2
steps = 60
bikeshare = State(um=10, comedor=2, um_vacio=0, comedor_vacio=0)

sweep = sweep_p1(bikeshare, p1_array, p2, steps)

plot(sweep, label="UM")
decorate(title='um-comedor Bikeshare', xlabel='Probabilidad p1',
        ylabel='Alumnos disconformes')

```

Los resultados se ven en la Fig. 6.

Ejer-
cicio

2.5 Notas finales del modelado

Como hemos visto, el trabajo de la simulación por computadora suele partir de una aplicación sencilla, y luego, a modo de espiral, vamos incorporando nuevas funcionalidades y complejidades, creando aplicaciones más grandes.

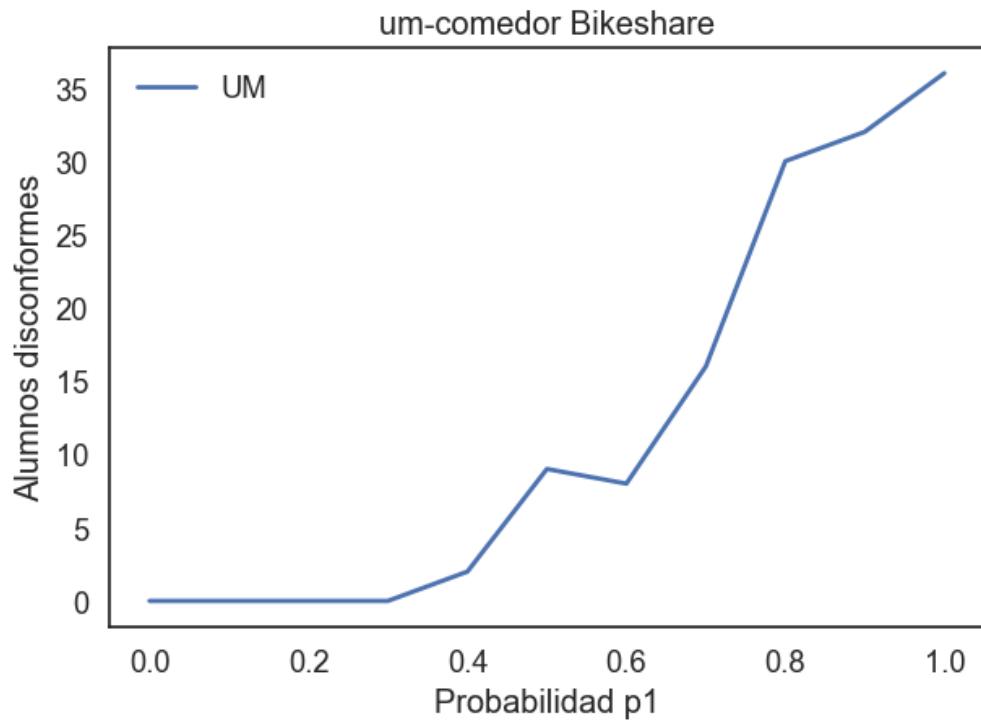


Figura 6: **Alumnos disconformes en la UM para el barrido de p1**

Esto facilita la depuración de la aplicación. En general lo que se busca es partir de una aplicación sencilla, tomada de un libro, de una fuente confiable, o que esté debidamente probada en su funcionamiento y resultados.

Luego se incorporan pequeños cambios estables que permitan incrementar la información que nos entrega, y sea fácil de depurar. Se ejecuta el programa y se comprueba que el funcionamiento sea el correcto. Luego se vuelven a incorporar nuevos cambios, pequeños y seguros.

La versión final de la aplicación no debe contener ese código que incorporamos en pasos intermedios, código que nos ayuda a mostrar resultados temporales y nos sirven para depurar el funcionamiento. Este código se denomina *scaffolding, andamios* que creamos para construir nuestra aplicación, y luego debemos quitar.

CAPÍTULO 3

Modelo Poblacional

Vamos a iniciar el estudio de modelos poblacionales partiendo del libro "The population bomb" publicado en 1968 por Paul Erlich, en el que pretendía predecir que el crecimiento de la población en los '70 iba a ser tal que la producción de alimentos, agricultura específicamente, no iba a dar a basto para toda la gente, y tendríamos dos décadas de hambruna.

Esto no fue así, pero el tópico de la población y los alimentos sigue vigente. Vamos a ver realizar un modelo poblacional desde los años '50, y vamos a "predecir" los próximos 50 o 100 años de población.

En [Wikipedia hay un artículo que habla de la población mundial](#), con datos y proyecciones, y lo utilizaremos para nuestros análisis. Para rescatar los datos utilizaremos la biblioteca Pandas de Python, con la función `read_html`. La tabla 2 de ese documento contiene los datos que vamos a analizar, por lo que primero debemos extraer dicha table en la variable `tabla2`.

```
from pandas import read_html
fd = 'https://en.wikipedia.org/wiki/
      Estimates_of_historical_world_population'
tables = read_html(fd, header=0, index_col=0, decimal='M')
```

```
table2 = tables[2]
```

Para simplificar los cálculos y el código, vamos a cambiar los encabezados de columna:

```
table2.columns = [ 'census' , 'prb' , 'un' , 'maddison' , 'hyde' , ,
   tanton' , 'biraben' , 'mj' , 'thomlinson' , 'durand' , 'clark' ]
```

Finalmente, cargaremos dos columnas que serán de interés para nuestro estudio, la del censo de los Estados Unidos (census), y la de las naciones unidas (un).

```
census = table2.census /1e9
un = table2.un /1e9
```

Podemos hacer algunas pruebas básicas de graficación de los datos usando `matplotlib`:

```
ms.plot(census, ':' , label='US Census')      # : linea de puntos
ms.plot(un, '--' , label='UN DESA')           # -- linea de rayas
ms.savefig('/tmp/census_vs_un.jpg')
```

Esto nos generará una gráfica en la que se verán comparadas las dos estimaciones. Esta gráfica se ve en la Figura 7.

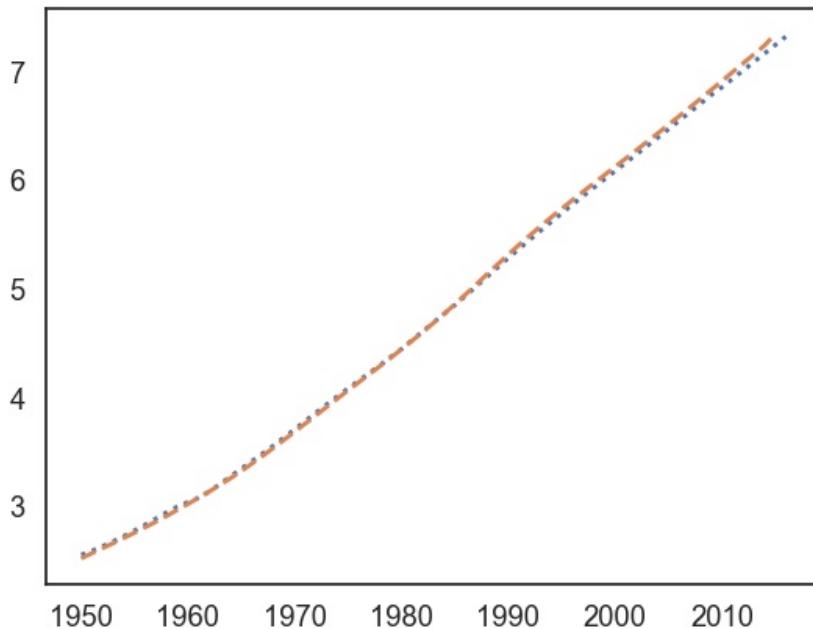


Figura 7: **US Census vs UN**

3.1 Crecimiento poblacional lineal

En esta sección analizaremos un primer modelo, primitivo pero funcional. Vamos a modelar la población mundial a través de un modelo de crecimiento constante, o lineal.

Primero debemos calcular el crecimiento total de la población desde el año 1950 hasta el 2016, rango de fechas que representan las tablas. Para ello vamos a parametrizar, como buena práctica, los tiempos de inicio y fin, y las poblaciones de inicio y fin de las tablas, usando el módulo ModSimPY.

```
import modsim as ms

# Primera y ultima fecha:
t_0 = ms.get_first_label(census)
t_end = ms.get_last_label(census)

# Cantidad de años:
anios = t_end - t_0

# Primera y última población:
p_0 = ms.get_first_value(census)
p_end = ms.get_last_value(census)

# Crecimiento total de la población:
crecimiento_total = p_end - p_0
```

Finalmente, podemos calcular el crecimiento anual constante de esta manera:

```
crecimiento_anual = crecimiento_total / anios
```

3.2 Simulación y graficación

Los sistemas dinámicos, por lo general, permiten obtener, y trabajan, con series temporales de datos, es decir, datos relativos a fechas, años, meses, semanas, días, horas, etc. Para facilitarnos el trabajo con series temporales de datos haremos uso de objetos de la clase `TimeSeries()` del módulo `modsim`. Por ejemplo, instanciamos un objeto para almacenar los resultados de la simulación:

```
resultado = ms.TimeSeries()
```

Ahora podemos almacenar la población inicial dentro de `resultado` en la fecha inicial:

```
resultado[t_0] = census[p_0]
```

Ahora podremos almacenar el resto de los valores en `resultado` teniendo en cuenta el crecimiento anual calculado en el apartado anterior. Para ello usaremos `linrange()`, una función de `modsim` que genera un arreglo NumPy de valores enteros entre un inicio y un fin:

```
for t in linrange(t_0, t_end):
    resultado[t+1] = resultado[t] + crecimiento_anual
```

En cada ciclo la variable `TimeSeries` `resultado` cargará un nuevo valor de población estimada para un año. Así, obtendremos un modelo lineal aproximado al real, en el que el valor poblacional del primero y último año va a coincidir con los datos históricos, no así los intermedios.

Si graficamos los valores de `resultado` veremos algo similar a lo mostrado en la Figura 8.

La Figura 8 representa la estimación de poblaciones para un modelo de crecimiento constante desde 1950 hasta 2016. Este modelo no se ajusta perfectamente a la curva real del censo de Estados Unidos o de las Naciones Unidas, pero resulta una aproximación inicial bastante buena.

Este modelo tiene dos problemas visibles. Este modelo no considera variables tan obvias de modificación de valores poblacionales como lo son las tasas de natalidad o mortalidad, y además, sugiere que la población crece linealmente hacia el infinito, lo cual no es muy razonable.

Ejer-
cicio

3.3 Modelando el crecimiento proporcional

Ya hemos creado una aproximación al sistema poblacional real, con sus defectos, pero una aproximación al fin. Ahora veremos cómo podemos mejorarlo.

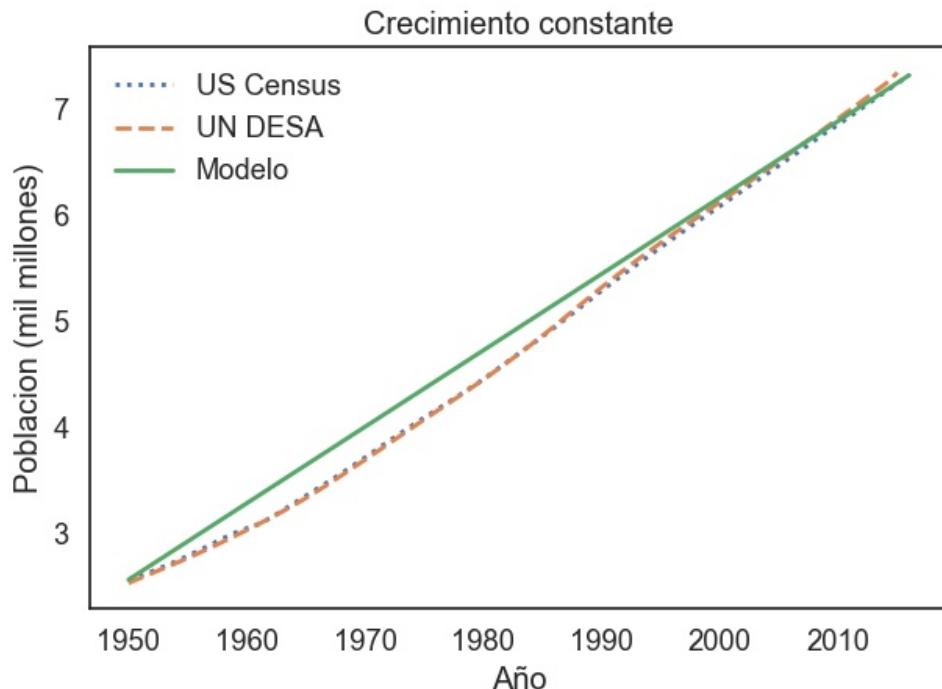


Figura 8: **Modelo de crecimiento constante**

Para ello haremos uso de un nuevo tipo de objeto del módulo ModSimPy, el objeto State. Este objeto nos va a permitir almacenar el estado del sistema. El objeto State contendrá variables de estado del sistema, y se irá actualizando durante la simulación.

Por otro lado incorporaremos un objeto System para facilitarnos el paso de parámetros entre funciones. El objeto System contendrá parámetros que en general no van a variar durante la simulación.

A modo de ejemplo, si nos remitimos al sistema de bicicletas compartidas, podríamos definir estos objetos de la siguiente manera:

- State: número de bicicletas en cada extremo.
- System: número de extremos, bicicletas en total, tasas de llegada y partida de las bicicletas, etc.

En el modelo poblacional podrían definirse de esta manera:

- State: población.
- System: tasa de crecimiento anual, población inicial, tiempo final de simulación, etc.

Supongamos que partimos de las variables definidas en la sección anterior:

```
t_0 = ms.get_first_label(census)
t_end = ms.get_last_label(census)
anios = t_end - t_0
print(str(anios) + " años")

p_0 = ms.get_first_value(census)
p_end = ms.get_last_value(census)
crecimiento_total = p_end - p_0

crecimiento_anual = crecimiento_total / anios
```

Algunos de estos valores son parámetros, por ejemplo, el tiempo inicial `t_0`, el tiempo final `t_end`, la población inicial `p_0` o el crecimiento anual constante `crecimiento_anual` por lo que vamos a almacenarlos en una variable tipo System:

```
sistema = ms.System(t_0=t_0, t_end=t_end, p_0=p_0, crecimiento_anual
=crecimiento_anual)
```

Ahora, esta variable `sistema` va a servirnos para parametrizar la ejecución de las simulaciones a través de funciones. Creemos entonces una función `run_simulation1` que englobe los pasos de simulación explicados anteriormente:

```
def run_simulation1(sistema):
    resultado = ms.TimeSeries()
    resultado[sistema.t_0] = sistema.p_0

    for t in ms.linrange(sistema.t_0, sistema.t_end):
        resultado[t+1] = resultado[t] + sistema.crecimiento_anual

    return resultado
```

Esta función almacenará los resultados de la simulación en una variable `TimeSeries` y la retornará.

Ahora creemos una función también para graficar, de modo que de aquí en adelante sea más simple la graficación de series de datos:

```

def plot_results(census, un, timeseries, title):
    pyplot.clf()
    ms.plot(census, ':', label='US Census')
    ms.plot(un, '--', label='UN DESA')
    ms.plot(timeseries, color='gray', label='Model')

    ms.decorate(xlabel='Year',
                ylabel='World population (billion)',
                title=title)
    ms.savefig('/tmp/grafico.jpg')

```

Ahora podremos correr la simulación y graficar los resultados mostrados en la Fig. 8 de esta forma:

```

res = run_simulation1(sistema)
plot_results(census, un, res, "Modelo de crecimiento constante")

```

3.4 Modelo de crecimiento proporcional

Un gran problema del modelo anterior, el de crecimiento constante, es que obviamente la gente no nace o muere toda junta una sola vez por año, y además, de un año a otro varía también la cantidad de personas que hacen que la población se modifique.

Una de las primeras aproximaciones que podríamos hacer para mejorar este modelo es definir que no siempre la cantidad de gente que nace es la misma que la que muere, por lo tanto, consideremos las tasas de natalidad y mortalidad.

Las tasas de natalidad y mortalidad son constantes, por lo que vamos a cargarlas dentro de nuestra variable `sistema`:

```

sistema.tasa_nat = 0.027
sistema.tasa_mor = 0.01

```

Creemos una función `run_simulation2` que contemple esta variación al momento de calcular la nueva población:

```

def run_simulation2(sistema):
    resultado = ms.TimeSeries()

```

```

resultado[sistema.t_0] = sistema.p_0

for t in ms.linrange(sistema.t_0, sistema.t_end):
    nacimientos = sistema.tasa_nat * resultado[t]
    muertos = sistema.tasa_mor * resultado[t]
    resultado[t+1] = resultado[t] + nacimientos - muertos

return resultado

```

Ahora podremos calcular la simulación invocando a esta nueva función:

```

res = run_simulation2(sistema)
plot_results(census, un, res, "Modelo de crecimiento proporcional")

```

Esta función generará el gráfico presentado en la Fig. 9

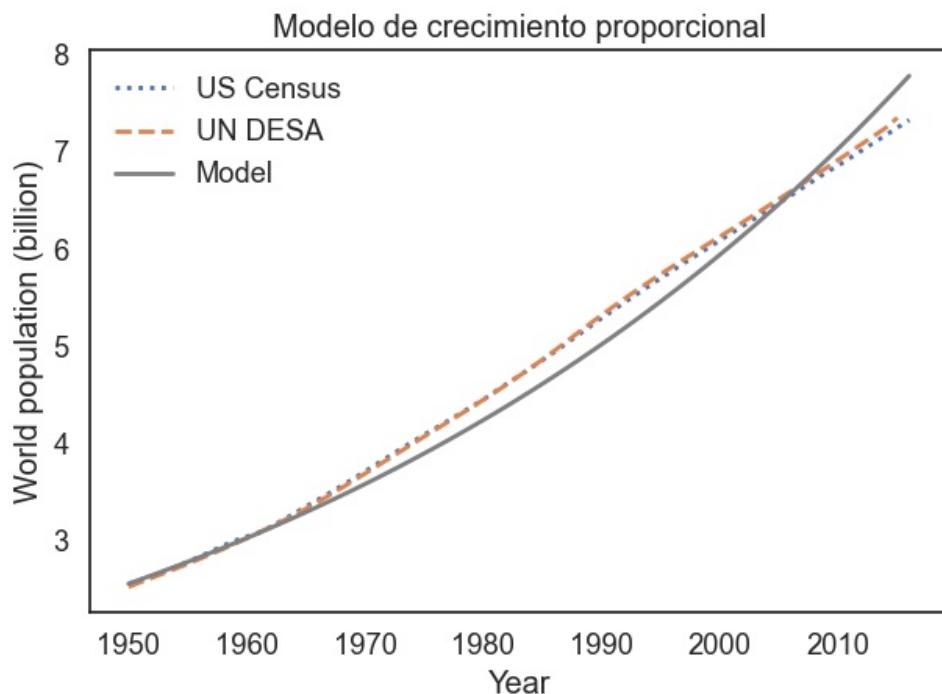


Figura 9: **Modelo de crecimiento proporcional**

Esta figura muestra que el modelo proporcional, a diferencia del de crecimiento constante, se ajusta mejor a los valores reales en los primeros años, pero luego comienza a diferir. Si bien el modelo constante se ajustaba mejor a los datos que el proporcional, este último va a permitirnos generar nuevos modelos que se adapten mejor al sistema real.

3.5 Creando una función de *step*

Las dos funciones de simulación anteriores, tanto `run_simulation1` como `run_simulation2`, son casi iguales, solamente difieren en la forma de generar nuevos valores poblacionales para llenar la serie temporal de datos. Podríamos englobar las líneas de código que generan nuevos valores en una nueva función, a la que llamaremos, función de actualización, o función de *step* (paso). Por ejemplo, para el modelo proporcional podríamos definir la siguiente función:

```
def func_actualizacion_proporcional(pob, t, sistema):
    nacimientos = sistema.tasa_nat * pob
    muertos = sistema.tasa_mor * pob
    return pob + nacimientos - muertos
```

Esta función recibe la variable de población, el tiempo, y la variable `System` con los parámetros. Esta función tomará la población pasada por argumento `pob` y generará el valor de la siguiente simulación. La variable `t` relativa al tiempo no se utiliza en esta función, pero vamos a considerarla para homogeneizar los parámetros con todas las funciones de actualización que podríamos definir en adelante.

De esta manera podríamos reescribir la nueva función de simulación... llamémosle `run_simulation`.

```
def run_simulation(sistema, func_act):
    resultado = ms.TimeSeries()
    resultado[sistema.t_0] = sistema.p_0

    for t in ms.linrange(sistema.t_0, sistema.t_end):
        resultado[t+1] = func_act(resultado[t], t, sistema)

    return resultado
```

Esta nueva función recibe por argumento la variable `sistema` para obtener los parámetros, y la función de actualización que debe considerar para generar nuevos valores de población.

Así, desacoplamos la simulación de la forma en la que se generan nuevos valores, y podemos *jugar* con distintas funciones de *step*.

Por ejemplo, ahora podríamos correr la simulación de esta forma:

```
res = run_simulation(sistema, func_actualizacion_proporcional)
```

3.6 Utilizando el parámetro α

Ahora vamos a simplificar un poco el código haciendo uso de un nuevo parámetro: α (α).

Si consideramos que la nueva población, en el modelo de crecimiento proporcional, se calcula como la población anterior, mas la cantidad de nacimientos, menos la cantidad de muertes, es decir:

```
nacimientos = tasa_natalidad * poblacion  
muertes = tasa_mortalidad * poblacion  
poblacion = poblacion + nacimientos - muertes
```

Podemos reordenar de esta forma:

```
poblacion = poblacion + tasa_natalidad * poblacion - tasa_mortalidad  
* poblacion
```

Lo cual nos permite sacar factor común:

```
poblacion = poblacion + (tasa_natalidad - tasa_mortalidad) *  
poblacion
```

Ahora, podemos llamar α al término que dejamos entre paréntesis, es decir:

```
alpha = tasa_natalidad - tasa_mortalidad  
poblacion = poblacion + alpha * poblacion
```

Por lo tanto, como α es un parámetro constante también, vamos a agregarlo a nuestro objeto `sistema` de esta forma:

```
sistema.alpha = sistema.tas_nat - sistema.tas_mor
```

A partir de acá podemos escribir una nueva versión de la función de actualización proporcional:

```

def func_actualizacion_alpha(pob, t, sistema):
    crecimiento_neto = pob * sistema.alpha
    return pob + crecimiento_neto

```

Y podemos calcular nuevamente la simulación, de manera sencilla, invocando a esta nueva función de step:

```
res = run_simulation(sistema, func_actualizacion_alpha)
```

3.7 Modelando el crecimiento cuadrático

En el capítulo anterior vimos un modelo de crecimiento proporcional, donde la nueva población se calculaba según una tasa `alpha` respecto de la población anterior. Este modelo resultó más realista que el modelo lineal, pero no se ajustaba tan bien a los datos como éste.

Esta falla en el ajuste tal vez se deba a que las tasas de natalidad y mortalidad, y por consiguiente, el `alpha`, no se ajustan correctamente a los datos, o tal vez la población se calcule de manera no lineal.

Teniendo en cuenta esto, vamos a generar un nuevo modelo, un modelo cuadrático, para intentar ajustarlo mejor a los datos históricos.

Primero vamos a introducir un parámetro `beta` en la ecuación, multiplicado por el cuadrado de la población, para obtener una función cuadrática.

```
crecimiento_neto = sistema.alpha * pob + sistema.beta * pob**2
```

Así, podríamos crear una nueva función de actualización:

```

def func_actualizacion_cuadratica(pob, t, sistema):
    crecimiento_neto = sistema.alpha * pob + sistema.beta * pob**2
    return pob + crecimiento_neto

```

Ahora debemos agregar un nuevo parámetro en nuestra variable `sistema`. Supongamos estos valores:

```

sistema.alpha = 0.025
sistema.beta = -0.0018

```

Podemos calcular la simulación cuadrática invocando a esta nueva función de step:

```
res = run_simulation(sistema, func_actualizacion_cuadratica)
plot_results(census, un, res, "Modelo de crecimiento cuadrático")
```

Esta nueva simulación generara un gráfico como el que se muestra en la Fig. 10. Como puede verse, se ajusta mucho mejor a los datos que el modelo proporcional.

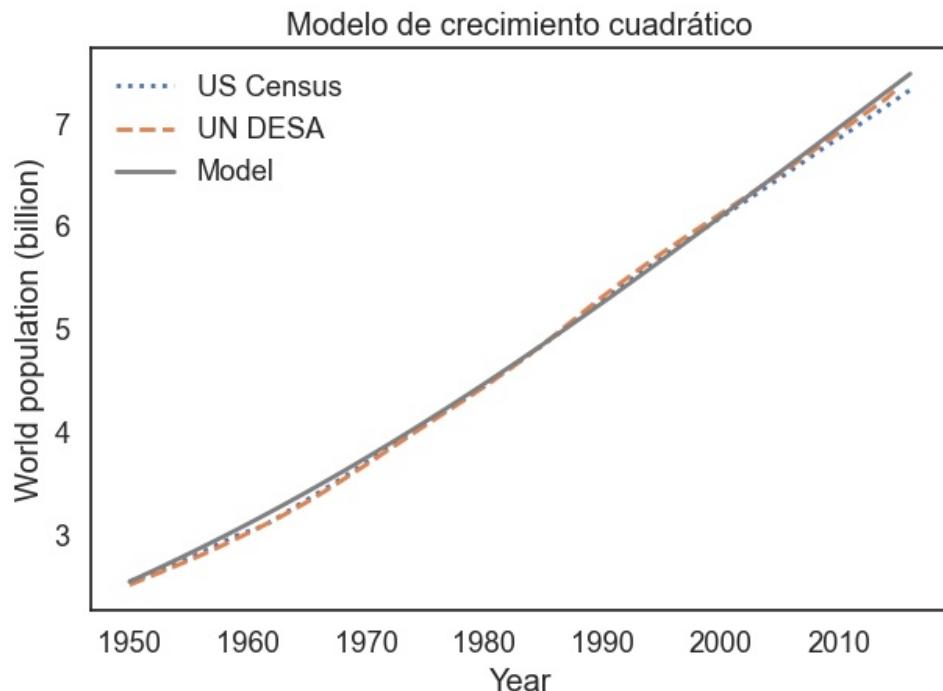


Figura 10: **Modelo de crecimiento cuadrático**

Era de esperarse que este modelo se ajustara mejor a los datos debido a que tenemos dos parámetros, en vez de uno, para modificar y adaptar. En general una mayor cantidad de parámetros facilita la adaptación y permite ajustar los modelos mejor a los sistemas reales. No obstante, y como puede verse en este ejemplo, incorporar más parámetros también añade un grado de complejidad en el modelado matemático.

3.8 Función cuadrática de población

El modelo cuadrático se adapta mejor a los valores reales, y aunque no es obvio que la población se comporte de esta manera, analicemos el crecimiento neto de la misma como función de la población, con la intención de añadir información adicional para analizar.

Primero vamos a crear un arreglo de 100 valores equidistantes de 0 a 15, dado que es un rango, en miles de millones de habitantes, con el que una población mundial podría estar representado, y para cada valor, veamos el crecimiento neto que generaría en un período de tiempo.

Para ello vamos a utilizar la función `linspace` de esta forma:

```
pob_array = ms.linspace(0,15,100)
```

Ahora, calculemos un arreglo de crecimientos netos de población para cada una de las poblaciones de `pob_array`:

```
crecimiento_neto_array = sistema.alpha * pob_array + sistema.beta *  
    pob_array**2
```

Esto almacenará en `crecimiento_neto_array` los valores de crecimiento neto de un período considerando cada una de las poblaciones iniciales de `pob_array`.

Si graficamos estos resultados obtendremos el gráfico de la Fig. 11.

```
pyplot.clf()  
ms.plot(pob_array, crecimiento_neto_array)  
ms.decorate(xlabel="poblacion (mil millones)", ylabel="crecimiento  
neto")  
ms.savefig("/tmp/crecimiento_neto_array.jpg")
```

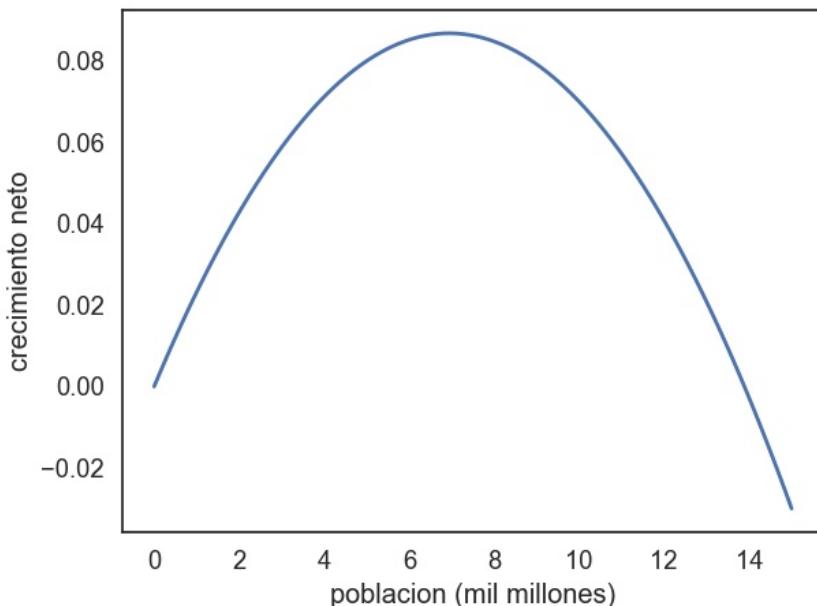


Figura 11: Crecimiento Neto para valores de población

Esta figura muestra en el eje de abscisas la población, y en el eje de ordenadas el valor del crecimiento neto para dicha población. Podemos dividir esta curva en cuatro comportamientos distintos:

1. Cuando la población es menor que 4 mil millones aproximadamente, el crecimiento neto es proporcional, tal y como lo muestra el modelo de crecimiento proporcional analizado anteriormente. La población crece lento porque es pequeña.
2. Entre los 4 y los 10 mil millones de habitantes la población crece más rápido porque es mayor.
3. Sobre los 10 mil millones, la población crece más lentamente. Esto podría explicarse como un efecto de la limitación de los recursos, que puede o reducir la cantidad de nacimientos, o aumentar la cantidad de muertes.
4. Aproximadamente en los 14 mil millones de habitantes el crecimiento neto es 0 (cero), por lo que la población no varía de un período al siguiente. Este punto se denomina **población de equilibrio**.
5. Más allá de los 14 mil millones de habitantes aproximadamente (luego calcularemos valores exactos) el crecimiento neto es negativo, lo que implica que la población se va reduciendo.

La población de equilibrio podríamos encontrarla analíticamente buscando las raíces de la ecuación cuadrática [3.1](#).

$$\Delta p = \alpha p + \beta p^2 \quad (3.1)$$

Si hacemos que la variación Δp sea cero, y sacamos a p como factor común, tenemos la ecuación [3.2](#):

$$0 = p(\alpha + \beta p) \quad (3.2)$$

Esta ecuación será cierta si ocurre una de dos condiciones:

- La población es nula: $p = 0$

- La población es $p = -\alpha/\beta$

La primera de estas condiciones no va a cumplirse dado que estamos analizando una población que tiene, por contexto, alrededor de 14 mil millones de habitantes, por lo que la solución es hallar el valor de p de la segunda condición. Dado que $\alpha = 0,025$ y $\beta = -0,0018$, tenemos que $p = 13,9$

Hagamos un par de sustituciones en la ecuación para llevarla a una forma más convencional:

$$r = \alpha$$

$$K = -\alpha/\beta$$

La ecuación 3.1 queda:

$$\Delta p = rp + (-r/K)p^2$$

Reordenando, tenemos la Ecuación 3.3:

$$\Delta p = rp(1 - p/K) \quad (3.3)$$

Con esta nueva ecuación es más sencillo interpretar al parámetro r como la mayor tasa de crecimiento, observada cuando la población es pequeña, y K como el punto de equilibrio. K se denomina también **capacidad de carga** del modelo, ya que indica la población máxima que puede subsistir sin reducirse/extinguirse.

3.9 Predicciones y proyecciones

Vimos en la sección anterior un modelo cuadrático que aproximaba muy bien los valores históricos de población desde 1950 hasta 2016. Ahora utilizaremos dicho modelo para realizar proyecciones futuras sobre el comportamiento de la población, y lo compararemos con las proyecciones previstas por expertos, también extraídas desde la Wikipedia.

Recordemos primero la función de actualización cuadrática que habíamos definido en la sección anterior:

Ejer-
cicio
Ejer-
cicio

```

def func_actualizacion_cuadratica(pob , t , sistema):
    crecimiento_neto = sistema.alpha * pob + sistema.beta * pob**2
    return pob + crecimiento_neto

```

Las simulaciones que realizamos se basaron en los datos y fechas de inicio y fin del censo de los Estados Unidos, es decir:

```

t_0 = ms.get_first_label(census)
t_end = ms.get_last_label(census)
p_0 = ms.get_first_value(census)

```

Y la variable `sistema` almacenaba todos estos parámetros:

```

sistema = ms.System(t_0=t_0 , t_end=t_end , p_0=p_0 , alpha=0.025 , beta
=-0.0018)

```

Ejecutemos una simulación, pero proyectando el tiempo final al año, por ejemplo, 2250, y grafiquemos los resultados (Fig. 12).

```

sistema.t_end = 2250
resultado = run_simulation(sistema,func_cuadratica1)
plot_results(census, un, resultado , "Modelo cuadratico con r y K")

```

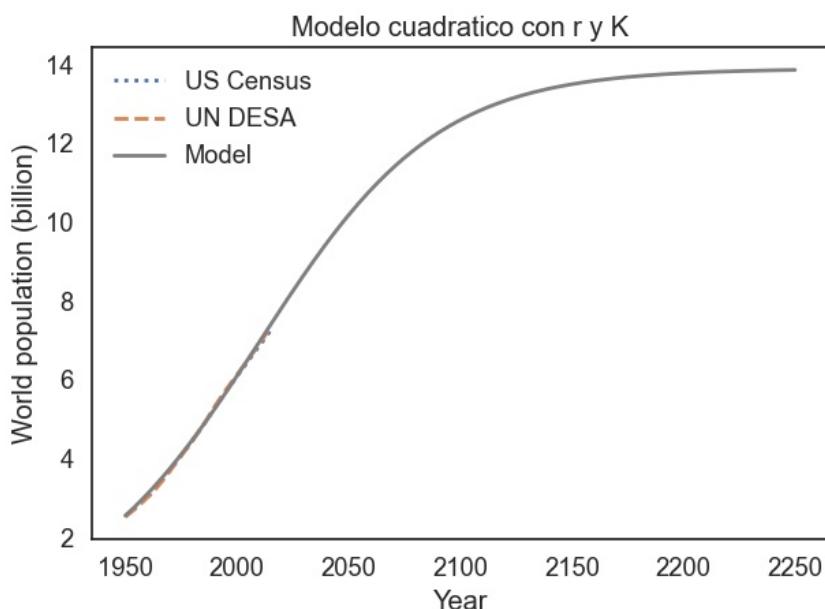


Figura 12: Proyecciones para el 2250 (modelo cuadrático)

La gráfica muestra los datos históricos modelados entre 1950 y 2016, y luego la proyección de valores poblacionales hasta el año 2250. De acuerdo con estos datos, la población mundial continuará creciendo casi linealmente por alrededor de 100 años, y luego crecerá más lentamente los siguientes 100 años, acercándose al valor de 13.9 mil millones de habitantes cerca del año 2250. Recordemos, este era nuestro valor de equilibrio, o capacidad de carga del modelo.

Este modelo asume que el crecimiento de la población está limitado por la disponibilidad de los recursos. En este escenario, cuando la población se acerca a la capacidad de carga del sistema, se producirá un equilibrio entre la cantidad de nacimientos y de muertes, y la población se mantendrá constante. Esto nos lleva a pensar que el modelo podría servir para estimar la capacidad de carga mediante simulaciones.

No obstante, en la realidad no todo esto es cierto. Los últimos 50 años de datos demuestran que el crecimiento poblacional no comienza a caer, al contrario, los primeros años tienen un crecimiento relativamente grande, por lo que no podrían servir los datos históricos para estimar esta caída de crecimiento neto. Además, la limitación de los recursos podría no ser tampoco la principal razón de esta disminución. Consideremos lo siguiente:

- La tasa de mortalidad no se está incrementando, de hecho, se ha reducido de un 1.8 % en 1950 a sólo un 0.8 % en la actualidad, de modo que la disminución en el crecimiento neto debería estar asociada a una reducción de la tasa de natalidad.
- En la medida en que las naciones se van desarrollando, y la calidad de vida de los habitantes mejora, la tasa de natalidad tiende a disminuir.

Por esto, no deberíamos tomar demasiado en serio las estimaciones de capacidad de carga del modelo. Sin embargo, analicemos ahora las estimaciones reales realizadas por expertos en demografía y realicemos la comparación con nuestro modelo cuadrático.

Consideremos el sitio de Wikipedia que venimos analizando:

https://en.wikipedia.org/w/index.php?title=Estimates_of_historical_world_population&oldid=938127092

La tabla 3 de este sitio contiene las proyecciones del censo de Estados Unidos y de las Naciones Unidas, por lo que, podríamos definir una función que lea dicha tabla:

```
def read_table3(filename = 'https://en.wikipedia.org/w/index.php?title=Estimates_of_historical_world_population&oldid=938127092'):
```

```

tables = read_html(filename, header=0, index_col=0, decimal='M')
table3 = tables[3]
table3.columns = ['census', 'prb', 'un']
return table3

tabla3 = read_table3()

```

Esta tabla en alguna de las proyecciones, y para alguno de los años, no posee proyecciones, por lo que en la lectura de Python obtendremos un valor "NaN"(Not a Number - No es un número). Pandas provee una función para eliminar los elementos NaN de tablas de datos: dropna. Podemos usarla en una función de graficación de proyecciones:

```

def plot_projections(table):
    census_proj = table.census / 1e9
    un_proj = table.un / 1e9

    ms.plot(census_proj.dropna(), 'b:', color='C0', label='US Census')
    ms.plot(un_proj.dropna(), 'g--', color='C1', label='UN DESA')
    ms.savefig('/tmp/grafico.jpg')

```

Como las proyecciones se extienden hasta el año 2100, calculemos las proyecciones hasta ese año con nuestro modelo cuadrático, y luego grafiquemos las proyecciones extraídas desde la tabla de la Wikipedia.

```

t_0 = ms.get_first_label(census)
t_end = 2100
p_0 = ms.get_first_value(census)
p_end = ms.get_last_value(census)

sistema = ms.System(t_0=t_0, t_end=t_end, p_0=p_0,
                     alpha=0.025, beta=-0.0018)
resultado = run_simulation(sistema, func_cuadratica1)

plot_results(census, un, resultado, "Proyecciones del modelo")
plot_projections(tabla3)

```

Esta gráfica se ve en la Fig. 13. Aquí se aprecian las gráficas del censo de las Naciones Unidas y nuestras proyecciones, ambos hasta el año 2100, y las proyecciones del censo de los Estados Unidos hasta el año 2050. Las proyecciones de los expertos demógrafos

tienen en cuenta muchos factores, como el nivel de vida de los diferentes países y regiones, la riqueza, etc., aspectos que no considera nuestro modelo. No obstante, ambas proyecciones son similares, por lo que podríamos decir que las proyecciones de nuestro modelo cuadrático no son descabelladas!

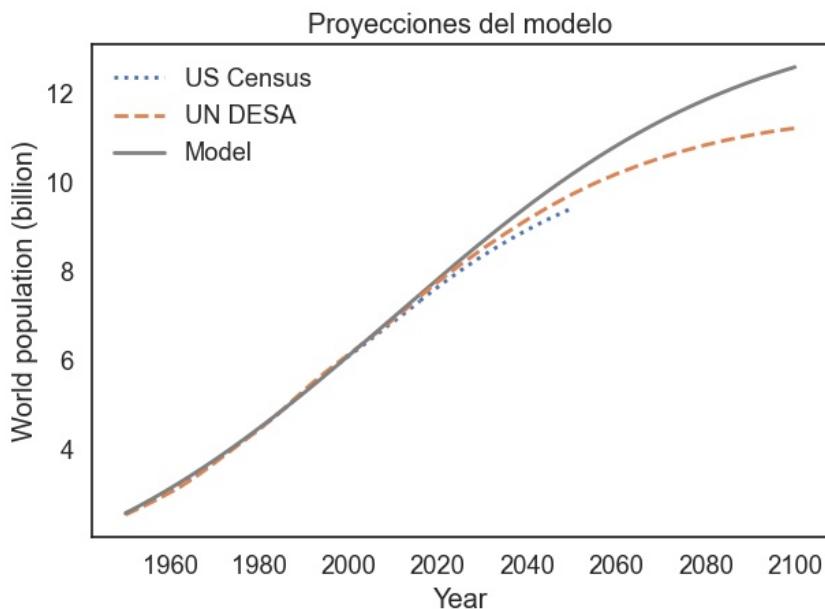


Figura 13: Proyecciones para el 2100 (comparación)

3.9.1 Calculando proyecciones: otra aproximación

Realicemos algunas pruebas adicionales. Sabemos que el crecimiento neto de año a año ha ido disminuyendo por décadas, por lo que podríamos también calcular proyecciones futuras basadas en este punto de vista.

Para ello vamos a usar una función del módulo ModSimPy denominada `compute_rel_diff`. Esta función calcula las diferencias relativas entre los elementos de un iterable de Python. Por ejemplo, si consideramos la siguiente lista:

```
lista = [1, 2, 1, 2, 1, 2, 4, 5, 4, 5, 8]
```

Podríamos generar el arreglo de diferencias relativas de esta forma:

```
ms.compute_rel_diff(lista)
```

Obteniendo:

```
array([ 1. , -0.5 ,  1. , -0.5 ,  1. ,  1. ,  0.25, -0.2 ,  0.25,  0.6 ,
       0. ])
```

Esto quiere decir que, considerando los cambios en los valores de la lista, las diferencias relativas son:

- de 1 a 2: aumenta el 100 % (1)
- de 2 a 1: se reduce el 50 % (-0.5)
- de 1 a 2: aumenta el 100 % (1)
- de 2 a 1: se reduce el 50 % (-0.5)
- de 1 a 2: aumenta el 100 % (1)
- de 2 a 4: aumenta el 100 % (1)
- de 4 a 5: aumenta el 25 % (0.25)
- de 5 a 4: reduce el 20 % (-0.2)
- de 4 a 5: aumenta el 25 % (0.25)
- de 5 a 8: aumenta el 60 % (0.6)
- de 8 a -: 0 % de cambio relativo

A estos valores se los denomina `alpha`. Calculemos las diferencias relativas `alpha` entre los valores poblacionales del censo de las Naciones Unidas y del de Estados Unidos, y realicemos el gráfico correspondiente.

```
pyplot.clf()
census_alpha = ms.compute_rel_diff(census)
ms.plot(census_alpha, label="Censo reldiff")
un_alpha = ms.compute_rel_diff(un)
ms.plot(un_alpha, label="UN reldiff")
ms.savefig("/tmp/reldiff.jpg")
```

Esto nos generará la gráfica de la Fig. 14.

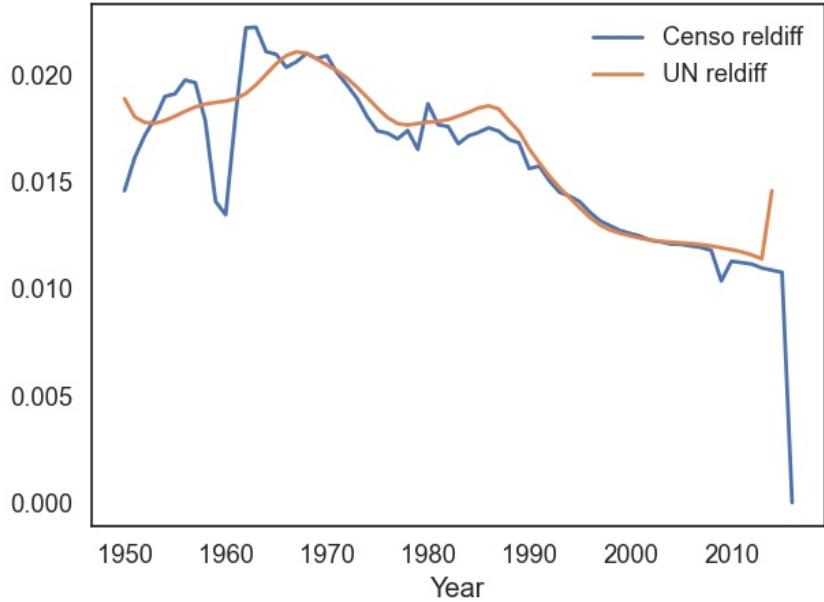


Figura 14: Diferencias relativas en cada censo.

Esta gráfica muestra grandes diferencias en los valore de alpha de ambas proyecciones en los cálculos previos a 1970. Luego, mas o menos las dos curvas coinciden y van reduciendo sus diferencias relativas hasta la actualidad.

Esto nos da pie a modelar de manera lineal, mediante una recta, este segmento de la curva. Creemos una función lineal, una recta, que aproxime lo mejor posible estas dos curvas desde 1970 en adelante. Supongamos la siguiente función alpha:

```
def alpha_func(t):
    b = 0.02
    m = -0.00021
    return b + m * (t - 1970)
```

Si ahora calculamos los valores de la recta para cada uno de los años desde 1970 hasta el 2020, y graficamos dicha recta, obtendremos la gráfica de la Fig. 15

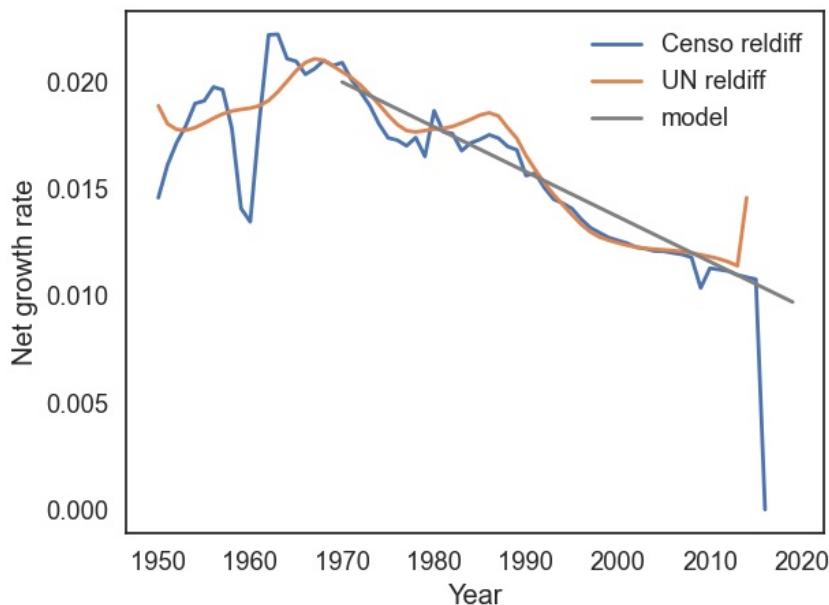


Figura 15: Diferencias relativas en cada censo. Función Alpha

El valor de alpha calculado en el modelo proporcional consideraba las tasas de natalidad y mortalidad. Este nuevo alpha también nos devuelve una tasa de cambio, pero esta vez calculada en base a las diferencias relativas de la población.

```
def func_actualizacion_alpha(pob, t, sistema):
    crecimiento_neto = sistema.alpha_func(t)*pob
    return pob + crecimiento_neto
```

Esto nos lleva a definir una nueva función de actualización, o step, teniendo en cuenta esta nueva forma de cálculo:

```
def func_actualizacion_alpha(pob, t, sistema):
    crecimiento_neto = sistema.alpha_func(t)*pob
    return pob + crecimiento_neto
```

Como se ve, la función de cambio alpha se ha cargado como parámetro en la variable sistema. Asignemos los valores iniciales de una nueva proyección, hasta el año 2100, basada en esta nueva función de actualización:

```
t_0 = 1970
t_end = 2100
p_0 = census[t_0]
```

```

sistema = ms.System(t_0=t_0, t_end=t_end, p_0=p_0,
                     alpha=0.025, beta=-0.0018,
                     alpha_func = alpha_func)

```

Finalmente, calculemos nuestra proyección con esta nueva función de actualización, y comparémosla con la proyección realizada por expertos en demografía:

```

resultado = run_simulation(sistema, func_actualizacion_alpha)
plot_results(census, un, resultado, "Proyecciones del modelo (
    alpha_func)")
plot_projections(tabla3)

```

El resultado puede apreciarse en la Fig. 16. En esta nueva proyección comienza a alejarse de las proyecciones oficiales, por debajo esta vez, y tendiendo a los últimos años, desde 2060 en adelante aproximadamente, pero en los años previos se acerca mejor a los valores oficiales que la proyección anterior.

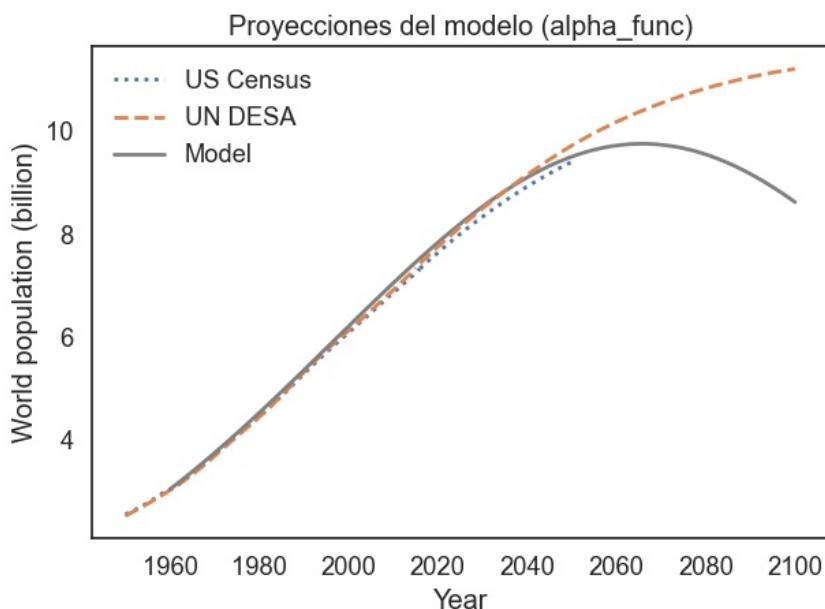


Figura 16: Proyección basado en la función Alpha

3.10 Análisis matemático

3.10.1 Relaciones recurrentes

El modelo poblacional que hemos visto es tan simple que ni siquiera hace falta simular, puede resolverse analíticamente. Puede resolverse analíticamente. Por ejemplo, podemos escribir el modelo de crecimiento constante como:

$$model[t + 1] = model[t] + annual_growth$$

En matemática la notación sería algo así:

$$x_{n+1} = x_n + c \quad (3.4)$$

Donde x_n es la población en el año n , x_0 es la población inicial, y c es la constante de crecimiento anual. Algunas veces es posible resolver relaciones recurrentes escribiendo una ecuación que calcule x_n para un valor dado de n directamente, sin calcular todos los valores como hace la simulación:

$$x_1 = x_0 + c$$

$$x_2 = x_1 + c$$

...

$$x_2 = x_0 + 2.c$$

$$x_3 = x_0 + 3.c$$

$$x_n = x_0 + n.c$$

Así, si queremos saber el valor de la población x_{100} podemos calcularlo directamente.

También podemos escribir el modelo proporcional con una relación de este tipo:

$$x_{n+1} = x_n + t_n.x_n - t_m.x_n$$

$$x_{n+1} = x_n + (t_n - t_m) \cdot x_n$$

Con $\alpha = (t_n - t_m)$:

$$x_{n+1} = x_n + \alpha x_n$$

O, más convenientemente aún:

$$x_{n+1} = x_n(1 + \alpha)$$

Y ahora podemos ver que:

$$x_1 = x_0(1 + \alpha)$$

$$x_2 = x_1(1 + \alpha)$$

$$x_3 = x_2(1 + \alpha)$$

...

O su equivalente:

$$x_1 = x_0(1 + \alpha)$$

$$x_2 = x_0(1 + \alpha)^2$$

$$x_3 = x_0(1 + \alpha)^3$$

...

$$x_n = x_0(1 + \alpha)^n$$

Este resultado es una progresión geométrica, es decir, cuando α es positivo, $(1 + \alpha)$ crece sin límites.

Finalmente podemos escribir un modelo cuadrático así:

$$x_{n+1} = x_n + \alpha x_n + \beta x_n^2 \quad (3.5)$$

También parametrizado como vimos:

$$x_{n+1} = x_n + rx_n \left(1 - \frac{x_n}{K}\right) \quad (3.6)$$

No hay solución analítica para esta ecuación, pero podemos aproximarla con una ecuación diferencial.

3.10.2 Ecuaciones Diferenciales

Partamos de esta ecuación nuevamente:

$$x_{n+1} = x_n + c$$

Podemos definir un Δx que sea el cambio en x desde un instante al siguiente:

$$\Delta x = x_{n+1} - x_n = c$$

Ahora si definimos un Δt como el *step* de tiempo, que es un año en el ejemplo, podemos escribir la tasa de cambio como:

$$\frac{\Delta x}{\Delta t} = c$$

Este modelo es discreto, es decir, solo está definido para valores enteros de n .

Pero en realidad la gente nace y muere todo el tiempo, no solo una vez al año, de modo que el modelo continuo más realista debería ser usando δ en vez de Δ :

$$\frac{dx}{dt} = c$$

Esta representación ya es una ecuación diferencial simple.

Podemos resolverla multiplicando m.a.m. por dt :

$$dx = c \cdot dt$$

Y entonces integrar m.a.m.:

$$\int dx = c \cdot \int dt$$

$$x(t) = ct + x_0$$

Donde x_0 es la constante de integración (no, no usamos c :P)

Similarmente podemos escribir el modelo de crecimiento proporcional de la siguiente manera. Partiendo de la ecuación de la función de step:

$$x_{n+1} = x_n + \alpha x_n$$

$$x_{n+1} - x_n = \alpha x_n$$

$$\Delta x = \alpha x_n$$

Ahora, si definimos el Δt como el intervalo de tiempo transcurrido entre dos instantes de tiempo determinados, la ecuación queda:

$$\frac{\Delta x}{\Delta t} = \alpha x_n$$

Como una ecuación diferencial, continua, puede escribirse de la siguiente manera:

$$\frac{dx}{dt} = \alpha x$$

Multiplicando m.a.m. por dt :

$$dx = \alpha x \cdot dt$$

Dividiendo m.a.m. por x :

$$\frac{dx}{x} = \alpha dt$$

Integramos m.a.m.:

$$\int \frac{dx}{x} = \int \alpha dt$$

$$\ln(x) = \alpha t + K$$

Donde K es la constante de integración.

Usamos exponenciación en ambos lados:

$$e^{\ln x} = e^{\alpha t + K}$$

Que puede reescribirse como sigue:

$$x = e^{\alpha t} \cdot e^K$$

Y dado que K es una constante arbitraria, e^K también lo es... llamémosle C :

$$x = e^{\alpha t} \cdot C$$

Hay muchas soluciones a esta ecuación diferencial con diferentes valores de C .

Como condiciones iniciales sabemos que, para un tiempo inicial, t_0 , la población será la población inicial, x_0 . De esta manera, si sustituimos $t = 0$ en la ecuación, obtendremos la población inicial.

Cuando $t = 0$, $x(0) = e^{\alpha \cdot 0} C = x_0$, por lo que $C = x_0$. También puede verse como $C = x(t)|_{t=0}$.

Así, nuestra solución particular es:

$$x(t) = x_0 \cdot e^{\alpha t}$$

Por último, podemos escribir la ecuación 3.6 del modelo cuadrático también como una ecuación diferencial.

$$x_{n+1} - x_n = x_n + rx_n\left(1 - \frac{x_n}{K}\right) - x_n$$

Si hacemos Δx como $x_{n+1} - x_n$ y sustituimos, obtenemos el Δx :

$$\Delta x = x_n + rx_n\left(1 - \frac{x_n}{K}\right) - x_n$$

Cancelando los valores x_n :

$$\Delta x = rx_n\left(1 - \frac{x_n}{K}\right)$$

Definiendo Δt como el intervalo de tiempo a analizar, la ecuación queda:

$$\frac{\Delta x}{\Delta t} = rx_n\left(1 - \frac{x_n}{K}\right)$$

Si lo llevamos a variable continua, la ecuación diferencial es:

$$\frac{dx(t)}{dt} = rx(t)\left(1 - \frac{x(t)}{K}\right) \quad (3.7)$$

De la misma forma puede aplicarse este razonamiento a la Ecuación 3.5, obteniendo lo siguiente:

$$\frac{dx(t)}{dt} = \alpha x_n + \beta x_n^2 \quad (3.8)$$

Ejer-
cicio

3.10.3 Análisis vs simulación

Una vez que diseñamos el modelo, tenemos dos formas de proceder: simulación o análisis (cálculo).

La simulación a menudo se hace como un programa de computadora que **modela los cambios** en el sistema **a lo largo del tiempo**, como los nacimientos y las muertes, o los movimientos de bicicletas entre lugares.

El análisis hace uso del álgebra y el cálculo, manipulación simbólica de notaciones matemáticas.

Ambos tienen diferentes ventajas y limitaciones. La **simulación** generalmente es más versátil, es fácil agregar y sacar partes de un programa, testear varias versiones del modelo, como hemos venido haciendo. Pero hay varias cosas que se pueden hacer con **análisis** que sería imposible hacer con simulación:

- Con análisis podemos calcular **exacta** y eficientemente un valor que con simulación podríamos solamente aproximar de forma menos eficiente.
- El análisis a menudo provee **atajos computacionales**, es decir, la posibilidad de saltar hacia adelante en el tiempo para calcular el estado del sistema muchos pasos adelante en el futuro, sin necesidad de mantener estados del modelo de forma simulada.
- Con análisis podemos crear **generalizaciones del modelo**, por ejemplo, podríamos probar que ciertos resultados siempre o nunca ocurrirán. Con simulación podemos mostrar ejemplos y contra-ejemplos de esto, pero es muy difícil escribir pruebas.
- El análisis pueden proveer visiones dentro de los modelos y sistemas que ellos describen, por ejemplo, a veces podemos identificar diversos comportamientos, y parámetros clave que los causan.

La matemática es un lenguaje diseñado por humanos para facilitar la manipulación de símbolos, como el álgebra. De forma similar, los lenguajes de programación están diseñados para representar ideas computacionales y ejecutar programas.

Si bien las matemáticas son poderosas, no todos los análisis son realizable desde el ese punto de vista, así como tampoco pueden analizarse todos los sistemas físicos únicamente desde el punto de vista de la simulación computarizada, o al menos, no en profundidad. Cada lenguaje es bueno para el propósito para el que fue diseñado, y menos bueno para el otro, pero generalmente son complementarios, así que vamos a trabajarlos en conjunto.

3.10.4 Ecuaciones diferenciales con sympy

En clase...

3.10.5 Ecuación logística

Otra forma de representar sistemas biológicos naturales simples, como una población humana, o de cualquier especie, aves, insectos, elefantes, etc.

La población se expresa como una magnitud P que depende del tiempo, por lo que la denotaremos $P(t)$. La idea es estudiar cómo varía la población para la siguiente generación, es decir, para el tiempo $t + 1$, o sea, cómo obtener el valor de $P(t + 1)$.

La población dependerá de los nacimientos y las muertes, por lo que podría expresarse la población con esta ecuación diferencial:

$$P(t + 1) = P(t).[b - c.P(t)] \quad (3.9)$$

Donde:

- b es la cantidad de nacimientos (población * tasa de natalidad)
- c es la tasa de mortalidad

Si ahora hacemos una sustitución como esta: $X = c.P(t)/b$. Si se despeja $P(t)$ se obtiene:

$$P(t) = (b/c).X(t) \quad (3.10)$$

Así, sustituyendo 3.10 en 3.9 queda:

$$(b/c).X(t + 1) = (b/c).X(t).[b - c.(b/c).X(t)]$$

$$X(t + 1) = X(t).[b - b.X(t)]$$

$$X(t + 1) = b.X(t).[1 - X(t)]$$

Y esto nos permite modelar la ecuación logística normalizada de esta manera:

$$X(t+1) = b \cdot X(t) \cdot [1 - X(t)] \quad (3.11)$$

$$X(t+1) = -b \cdot X^2(t) + b \cdot X(t) \quad (3.12)$$

Esta es la forma normalizada de la ecuación, y tiene la ventaja que nos permite estudiar poblaciones en forma relativa entre una generación t y la siguiente $t + 1$, ya que X varía entre 0 y 1 ($0 < X < 1$).

Además, el coeficiente b sólo puede tomar valores entre 1 y 4 ($1 < b < 4$), ya que para menores de 1 y mayores de 4 la población se extingue.

Por último, por su condición de **no linealidad**, el resultado de esta ecuación depende fuertemente de las condiciones iniciales. No es lo mismo una población inicial de 100 habitantes que una de 100000.

Esta ecuación además tiene otras características importantes. Suponiendo, por ejemplo, una población inicial de 0.2 ($x_0 = 0.2$), es interesante analizar cómo se comporta el modelo para diferentes valores del parámetro b .

En la Figura 17 se puede ver el desarrollo temporal y su diagrama de fase con estos valores.

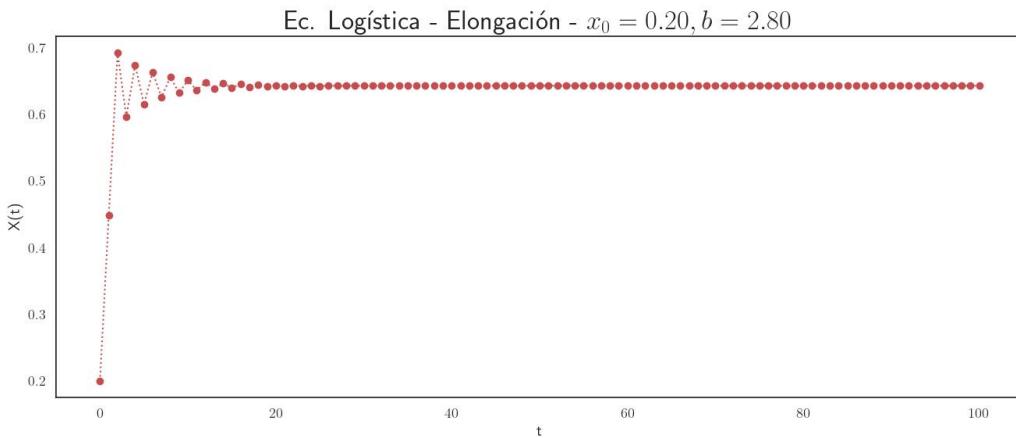


Figura 17: Ecuación Logística - Elongación - $b=2.8$

Realicemos un diagrama de fase comparando los valores de la población $X(t)$ con los de la siguiente población, $X(t + 1)$. A nivel gráfico resulta útil, además de graficar los puntos correspondientes, graficar también la curva de $X(t)$, y una recta a 45°.

La gráfica final puede verse en la Fig. 18.

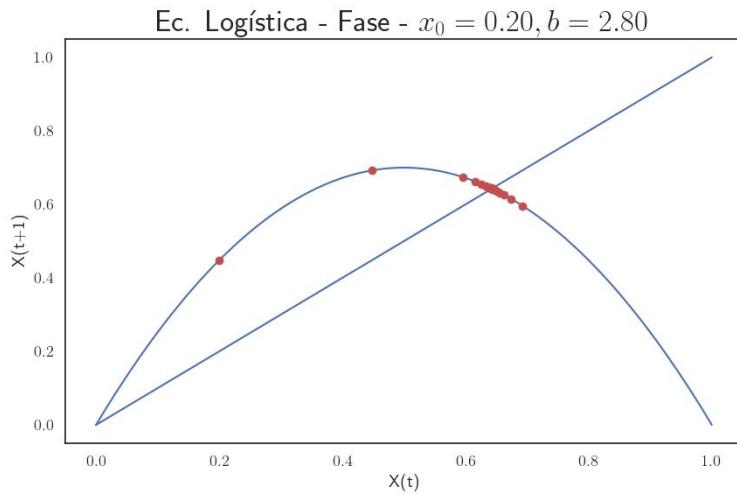


Figura 18: Ecuación Logística - Fase - $b=2.8$

La intersección de la recta de 45° con la curva de $X(t)$ muestra el punto de estabilidad en 0.6429. Es decir, cuando la población va evolucionando y alcanza dicho valor, se estabiliza, la siguiente población será igual que la anterior.

Veamos ahora qué ocurre si modificamos el coeficiente b a $b = 3.1$ y comenzamos con el mismo valor inicial de 0.2. El sistema presenta una bifurcación y se estabiliza ahora en dos valores: 0.764 y 0.558. Las Figuras 19 y 20 muestran este comportamiento en elongación y fase respectivamente.

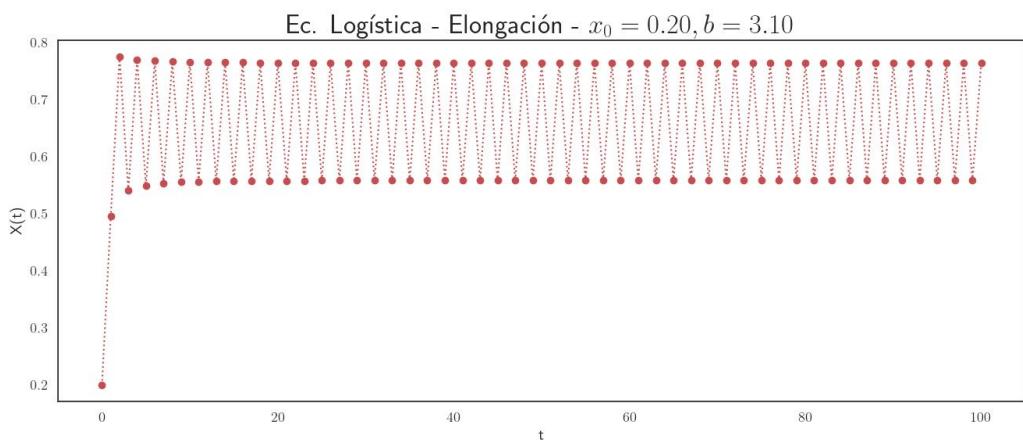


Figura 19: Ecuación Logística - Elongación - $b=3.1$

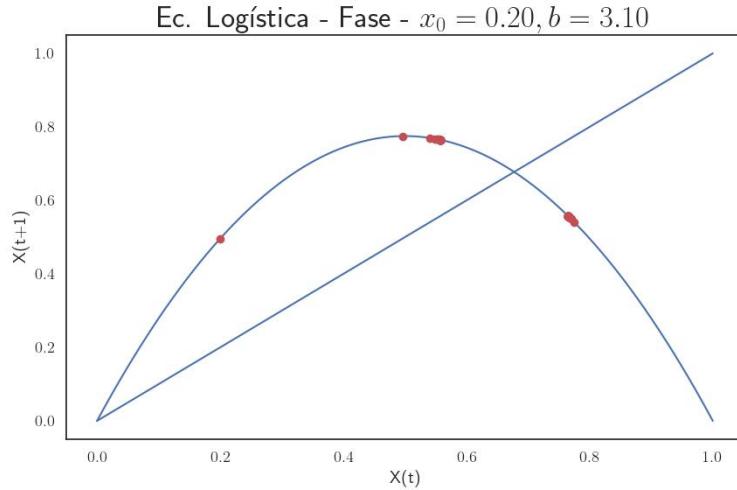


Figura 20: Ecuación Logística - Fase - $b=3.10$

Ahora si volvemos a modificar a b por $b = 3.49$ el sistema se estabiliza en 4 valores: 0.8723, 0.3885, 0.8291 y 0.4944, en ese orden de aparición.

Las Figuras 21 y 22 muestran este comportamiento en elongación y fase respectivamente.

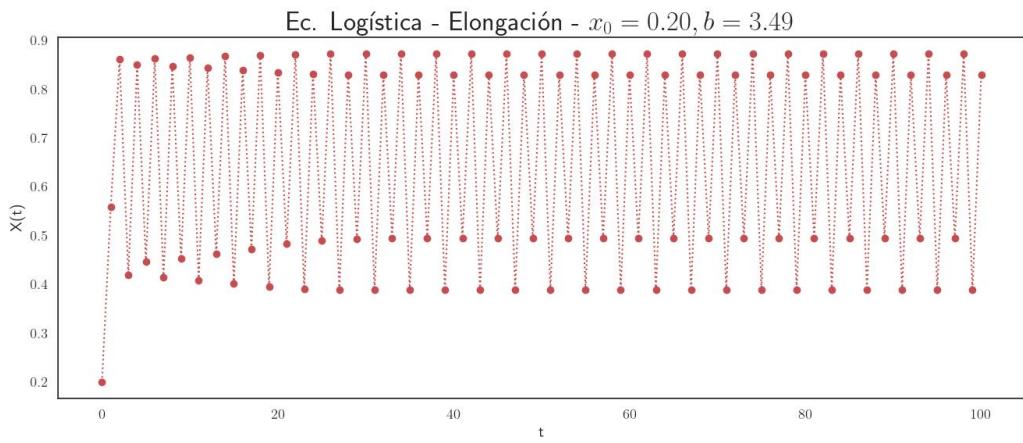


Figura 21: Ecuación Logística - Elongación - $b=3.49$

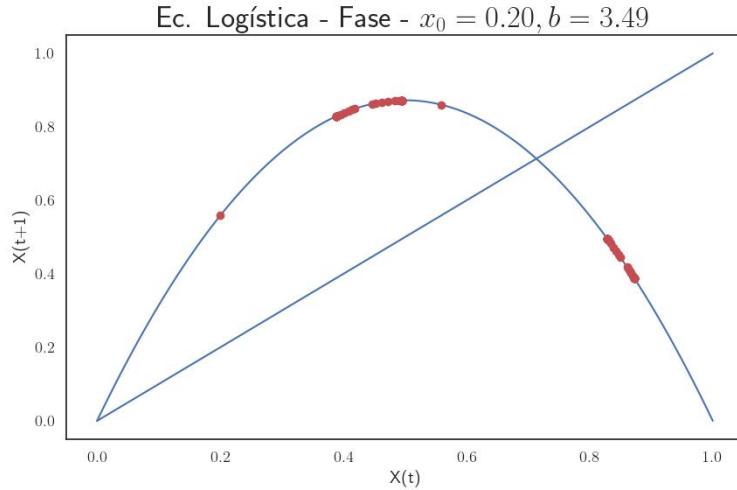


Figura 22: Ecuación Logística - Fase - $b=3.49$

Finalmente para un valor $b = 3.8$ el sistema entra en lo que se denomina régimen caótico, no se estabiliza en ningún valor particular. Las Figuras 23 y 24 muestran este resultado en elongación y fase respectivamente.

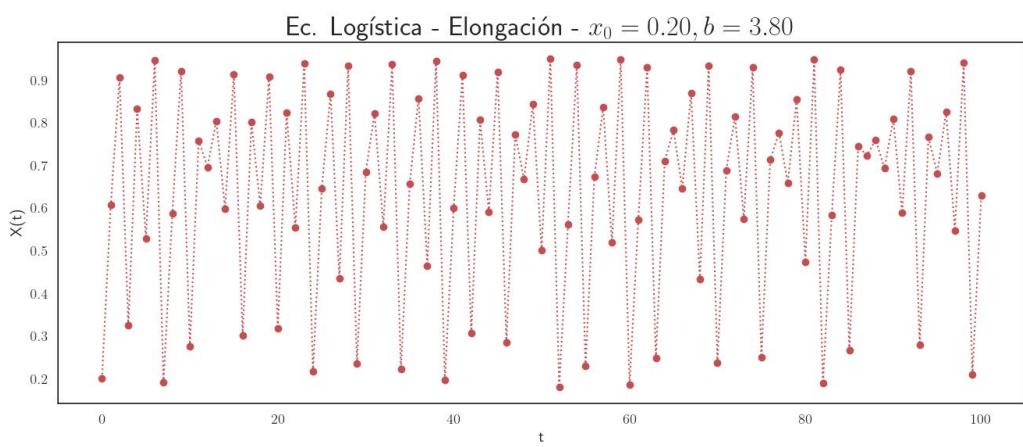


Figura 23: Ecuación Logística - Elongación - $b=3.80$

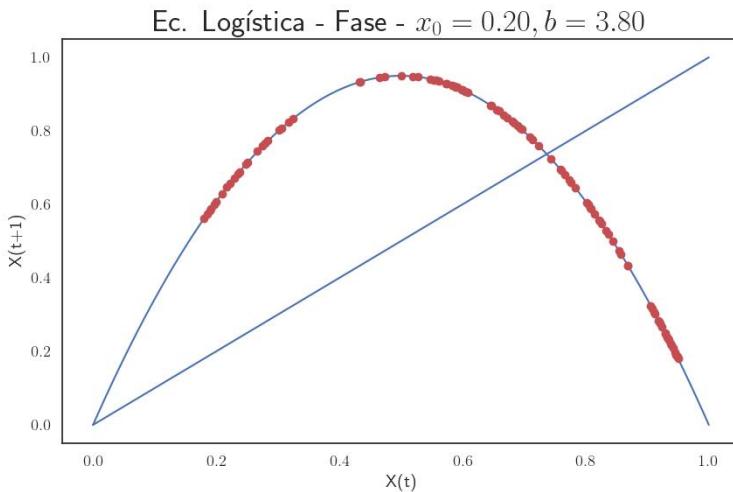


Figura 24: Ecuación Logística - Fase - $b=3.80$

De esto podemos sacar algunas conclusiones:

1. Para valores de $b < 2,98$ la población final se estabiliza en un único valor.
2. Para valores entre 2,98 y 3,45 existen dos posibles soluciones, se estabiliza en dos valores.
3. Para valores de b entre 3,45 y 3,57 se estabiliza en 4 valores.
4. Para valores de b entre 3,57 y 3,8284 no se estabiliza (sistema caótico / zona caótica).
5. Para valores de b entre 3,83 y 3,84 se estabiliza en 3 valores.
6. Para valores de b mayores a 3.84 no se estabiliza (sistema caótico / zona caótica)

Si graficamos el diagrama de puntos de estabilidad para valores de b , obtenemos un diagrama de Feigenbaum , como se ve en la Figura 25:

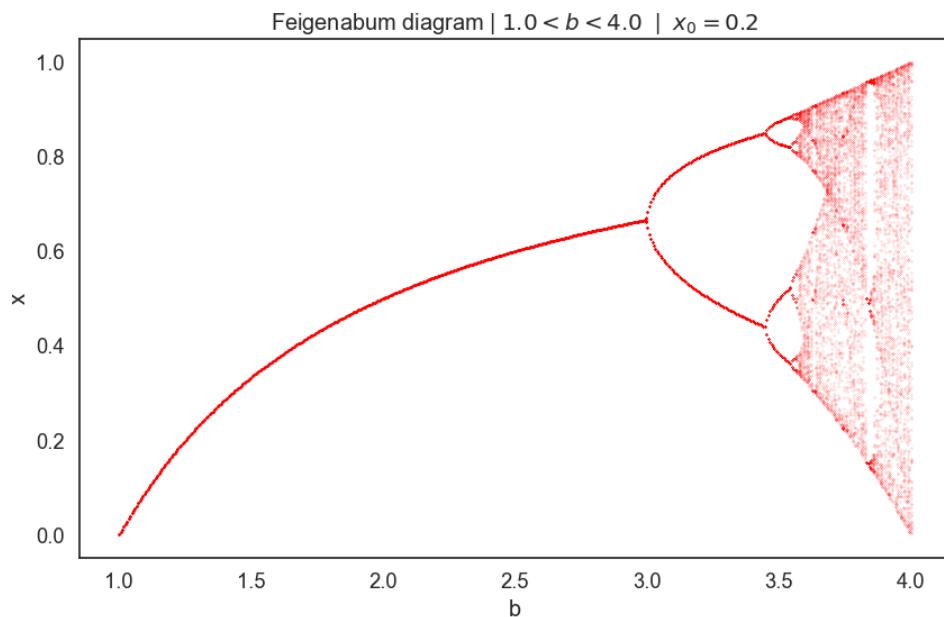


Figura 25: Diagrama de Feigenbaum

Si ampliamos la zona caótica, por ejemplo, entre 3.5 (cuatro valores de estabilidad) y 3.66 (zona caótica) podemos ver esto (Figura 26)

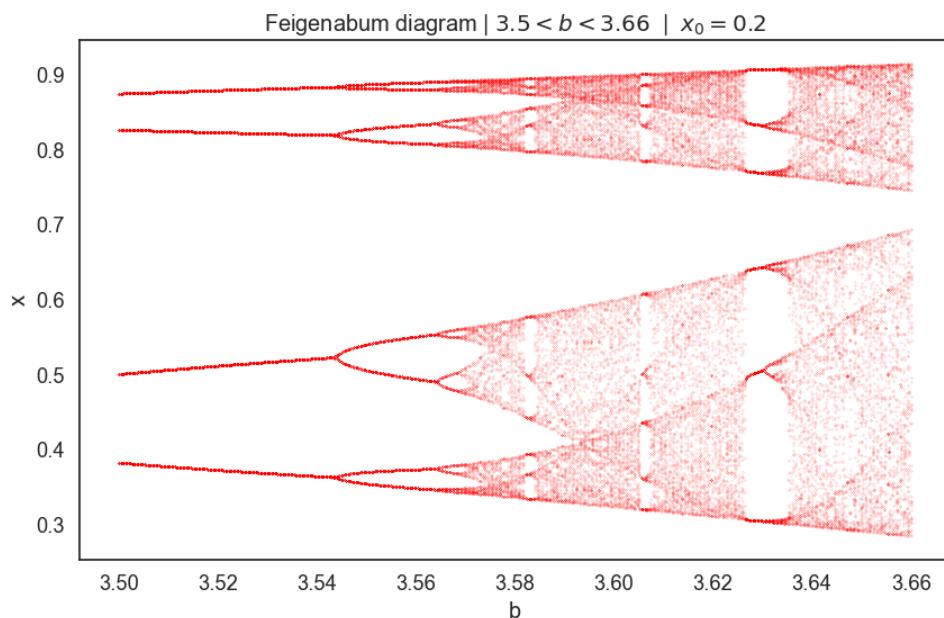


Figura 26: Diagrama de Feigenbaum

Si hacemos otra ampliación entre 3.6 y 3.66 (zona caótica marcada entre las rectas rojas) y acotamos los valores de X entre 0.4 y 0.6, veremos algo similar a lo que muestra la Figura 27:

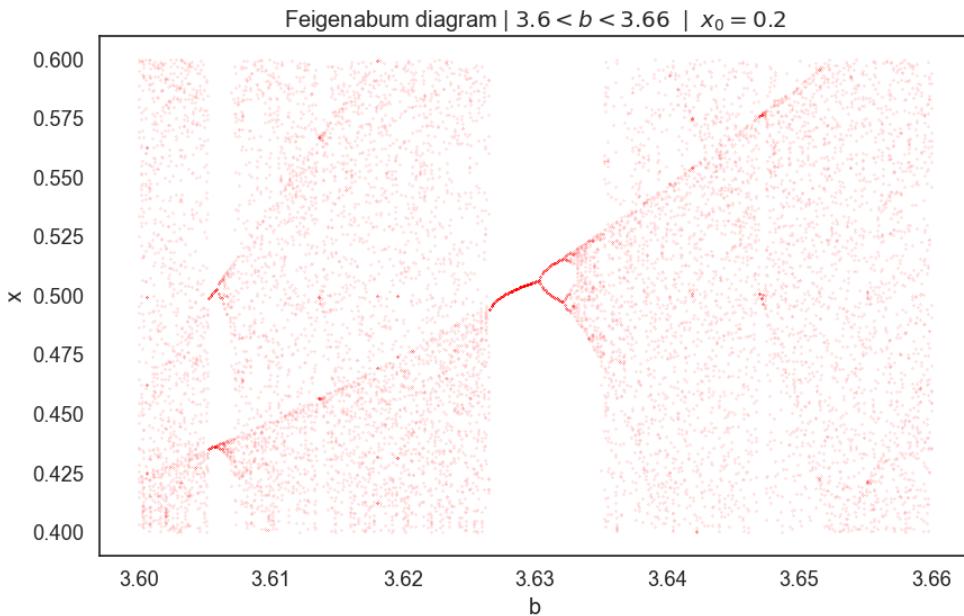


Figura 27: Diagrama de Feigenbaum

Como se ve, tenemos el mismo comportamiento del diagrama original, pero cada vez en menor escala.

A esto se lo denomina **comportamiento fractal**, la repetición del mismo patrón grafico en diferentes escalas. Más adelante en la materia veremos que esto ocurre con otros modelos caóticos también.

CAPÍTULO 4

Modelo Epidemiológico

Desarrollaremos ahora un modelo de un modelo epidemiológico en la medida en que se propaga en la población. El caso particular será el de la plaga Freshman.

Cada año en el colegio Olin cerca de 90 nuevos estudiantes llegan al campus desde varias partes del país y del mundo.

La mayoría llegan felices y saludables, pero usualmente al menos uno llega con alguna clase de enfermedad infecciosa.

Unas semanas después y en forma predecible una fracción de la clase entera se enferma con lo que llamamos la plaga Freshman.

Veremos ahora un bien conocido modelo de enfermedades infecciosas, el modelo KermackMcKendrick, y lo usaremos para explicar la progresión de la enfermedad sobre el transcurso del semestre, prediciendo el efecto de las posibles intervenciones (como vacunas) y diseñando la campaña de intervención más efectiva.

Hasta ahora hemos creado nuestro propio modelo, definido los parámetros, variables importantes, etc.

Ahora veremos un modelo ya creado y funcional, lo analizaremos, y haremos ingeniería inversa sobre él, identificando algunas decisiones importantes del modelado, ventajas y desventajas.

4.1 El modelo SIR

El modelo de Kermack - McKendrick es un modelo simplificado de un modelo SIR, sigla que identifica tres clases de personas:

- S: personas susceptibles, capaces de contraer la enfermedad si están en contacto con alguien infectado.
- I: personas infectadas, capaces de pasar la enfermedad a otros con los que tengan contacto.
- R: personas recuperadas, consideradas inmunes a la infección para este modelo.

Pensemos en cómo el número de personas en cada categoría cambia con el tiempo. Supongamos que sabemos que las personas con la enfermedad están infectadas por un período de 4 días en promedio. Si 100 personas son infectadas en un punto determinado del tiempo, e ignoramos el tiempo particular que cada una va a estar infectada, esperamos que se recuperen de 1 a 4 días.

De otra manera, si el tiempo entre recuperaciones es de 4 días, la tasa de recuperación es de 0.25 personas recuperadas por día, lo que bien podemos denotar como la **tasa de recuperados por día** con la letra gamma γ .

Si el total de personas en la población es de N , y la fracción de los infectados es i , el total de recuperados por día podría considerarse: $\gamma i N$.

Ahora pensemos en el número de infectados. Supongamos que sabemos que cada persona susceptible se pone en contacto con una persona cada 3 días en promedio, de modo que la otra persona podría infectarla si está infectada. Denotemos a esa **tasa de contacto** con la letra β .

En general no se puede asumir que sabemos β por adelantado, pero mas adelante veremos cómo estimarlo en base a datos históricos.

Si s es una fracción de la población que es susceptible, $s.N$ es el número de personas susceptibles, βsN es el número de contactos por día, y βsiN es el número de esos contactos en los que la otra persona es infecciosa.

En resumen:

- El número de recuperados por día es $\gamma i N$. Dividiéndolo por N nos da la fracción de la población que se recupera en un día, γi .
- El número de nuevas infecciones que podemos esperar por día es βsiN . Dividiéndolo por N nos da la fracción de la población que es infectada por día, βsi .

Este modelo asume que la población es cerrada, es decir, no llega ni sale nadie, de modo que el tamaño de la población, N , es constante.

4.2 Las ecuaciones SIR

Si consideramos al tiempo como una cantidad continua, podemos escribir ecuaciones diferenciales que describan las tasas de cambio para s , i y r , donde r es la fracción de la población que se ha recuperado:

$$\frac{ds}{dt} = -\beta si \quad (4.1)$$

$$\frac{di}{dt} = \beta si - \gamma i \quad (4.2)$$

$$\frac{dr}{dt} = \gamma i \quad (4.3)$$

Para evitar complicar mucho las ecuaciones hemos puesto s , i y r pero en realidad son funciones del tiempo, $s(t)$, $i(t)$ y $r(t)$ respectivamente (sistemas dinámicos, sistemas que cambian con el tiempo).

Los modelos SIR son ejemplos de modelos de compartimiento, porque dividen al mundo en categorías discretas, o compartimientos, y describen también las transiciones de un compartimiento a otro. También podemos encontrar en la bibliografía que los compartimientos se denominan existencias, y las transiciones flujos.

En este ejemplo tenemos tres categorías, susceptibles, infectados y recuperados, y dos flujos, nuevas infecciones y nuevos recuperados. Podría graficarse como se ve en la Figura 28



Figura 28: Modelo SIR

Los compartimientos y los flujos se grafican generalmente usando diagramas de flujo o de bloques.

Los compartimientos o categorías con rectángulos, los flujos o cambios de estado con flechas.

El widget de las flechas representan las válvulas que controlan las tasas de flujo en el cambio de estado, ahí se grafican los parámetros que controlan dichas válvulas.

4.3 Implementación

Para un sistema físico hay muchos modelos posibles, y para uno dado, hay muchas formas de representarlos en código. Por ejemplo, podemos representar un modelo SIR como un diagrama de bloques y de flujo, como un conjunto de ecuaciones diferenciales, o como un programa en Python. El proceso de representar un modelo en estas formas se denomina implementación. Ahora veremos cómo implementar un SIR en python.

Representaremos el estado inicial del sistema usando un objeto State con las variables S, I y R, que representan la población en cada compartimiento del gráfico:

```
init = State(S=89, I=1, R=0)
```

Ahora convertimos los números en fracciones dividiéndolos por el total:

```
init /= sum(init) # init = init / sum(init)
N = sum(init)
```

Por ahora asumiremos que sabemos la tasa de contactos por día y la tasa de recuperados por día:

```
tc = 3      # time between contacts in days  
tr = 4      # recovery time in days
```

Podemos ahora calcular el beta y el gamma:

```
beta = 1 / tc    # contact rate in per day  
gamma = 1 / tr   # recovery rate in per day
```

Ahora necesitamos un objeto System para almacenar los parámetros y las condiciones iniciales. La siguiente función toma los parámetros del sistema como parámetros de la función y retorna un objeto System con todo:

```
def make_system(beta, gamma):  
    init = State(S=89, I=1, R=0)  
    init /= sum(init)  
    t0 = 0  
    t_end = 7 * 14  
    return System(init=init, t0=t0, t_end=t_end, beta=beta, gamma=gamma)
```

El valor por defecto para t_end es de **14 semanas (7*14 días)**.

4.4 La función de actualización (step)

En cualquier punto del tiempo el estado del sistema es representado por un objeto State con tres variables, S, I y R, de modo que definiremos una función que tome el System como parámetro y el tiempo actual, y devuelva un estado nuevo:

```
def update_func(state, t, system):  
    s, i, r = state  
    ##### variable = state \rightarrow variable.S, variable.I, variable.R  
    infected = system.beta * i * s  
    recovered = system.gamma * i  
    s -= infected
```

```

i += infected - recovered
r += recovered
return State(S=s, I=i, R=r)

```

La primera línea usa una asignación múltiple, el valor de state es un objeto State que contiene los tres valores. A la izquierda todas las variables donde van a caer esos valores en orden.

Las variables locales, s, i y r, son minúsculas para distinguirlas de S, I y R.

La función de actualización calcula los infectados y los recuperados como fracciones de la población, y entonces actualiza *s*, *i* y *r*. Esos valores están contenidos en el objeto State que retorna.

Así, podemos llamar a esta función de esta manera:

```
state = update_func(init, 0, system)
```

El resultado podría ser este:

	Valor
S	0.985388
I	0.011865
R	0.002747

Figura 29: SIR

Además se puede ver que la función `update_func` no usa el parámetro *t*, se incluye porque las funciones de update en general sí lo usan o dependen del tiempo, y es conveniente si ponerlo en todas, se use o no.

4.5 Corriendo la simulación

Ahora simularemos el modelo sobre una secuencia de pasos:

```

def run_simulation(system, update_func):
    state = system.init
    for t in linrange(system.t0, system.t_end):

```

```

state = update_func(state, t, system)
return state

```

Los parámetros de `run_simulation` son el objeto `System` y la función `update`. El objeto `System` contiene los parámetros del modelo, condiciones iniciales y valores de `t0` y `t_end`.

Esto es similar a lo que ya hemos visto en el modelo poblacional:

```

system = make_system(beta, gamma)
final_state = run_simulation(system, update_func)

```

El resultado del estado final será este:

	Valor
S	0.520819
I	0.000676
R	0.478505

Figura 30: SIR

Este resultado indica que después de 14 semanas (98 días) cerca del 52 % (aprox 47 personas) de la población sigue siendo susceptible de ser contagiada, lo que significa que nunca han sido infectados, menos del 1 % fueron infectados, y el 48 % (aprox 43 personas) fue recuperado, es decir, en algún punto se infectaron y se curaron.

4.6 Recolectando los resultados

La versión previa del `run_simulation` solo retorna el estado final, pero podríamos ver cómo los estados van cambiando durante el tiempo.

Hay varias formas de hacer esto, pero una conveniente es la de usar tres objetos `TimeSeries`, y otra es usar un nuevo objeto llamado `TimeFrame`.

Veamos la resolución usando `TimeSeries`:

```

def run_simulation(system, update_func):
    S = TimeSeries()
    I = TimeSeries()

```

```

R = TimeSeries()
state = system.init
t0 = system.t0
S[t0], I[t0], R[t0] = state
for t in linrange(system.t0, system.t_end):
    state = update_func(state, t, system)
    S[t+1], I[t+1], R[t+1] = state
return S, I, R

```

Primero creamos los objetos TimeSeries para almacenar los resultados. Notese que las variables S, I y R son objetos TimeSeries ahora.

Luego inicializamos el state y el t0, y el primer elemento de S, I y R con el state.

Dentro del bucle usamos la función update_func para calcular el estado del sistema en el siguiente step, entonces usamos asignaciones múltiples para almacenar cada elemento del state por separado en los TimeSeries.

Al final retornamos los elementos S, I y R.

```

system = make_system(beta, gamma)
S, I, R = run_simulation(system, update_func)

```

Podemos usar esta función para graficar:

```

def plot_results(S, I, R):
    plot(S, '--', label= 'Susceptible')
    plot(I, ' - ', label= 'Infected')
    plot(R, ':', label= 'Resistant')
    decorate(xlabel= 'Time (days)', ylabel= 'Fraction of population')

```

Y finalmente graficar. Ver Figura 31

```
plot_results(S, I, R)
```

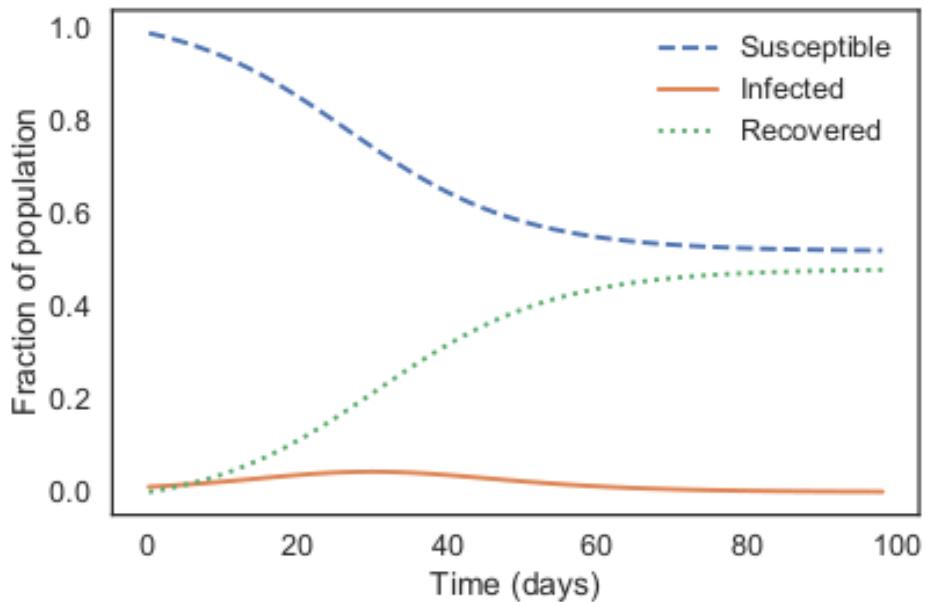


Figura 31: SIR - Resultado inicial

Notar en la gráfica que alrededor del día 21 comienza el brote, y hasta el día 42 crece hasta su punto máximo. La fracción de la población que está infectada nunca es muy alta, pero se suma y la gente comienza a recuperarse. En total casi la mitad de la población se enferma.

Veamos ahora la versión del `run_simulation` con `TimeFrame`.

Si el numero de variables de estado es pequeño, como este caso, podemos almacenarlas separadas como `TimeSeries` como hemos hecho. Pero si es más grande una buena alternativa es usar `TimeFrame`, otro objeto definido en `ModSim`.

Un `TimeFrame` es casi igual al `DataFrame` que usamos antes. La función nueva quedaría algo así:

```
def run_simulation(system, update_func):
    frame = TimeFrame(columns=system.init.index)
    frame.row[system.t0] = system.init
    for t in linrange(system.t0, system.t_end):
        frame.row[t+1] = update_func(frame.row[t], system)
    return frame
```

La primera línea crea un objeto TimeFrame vacía con una columna por variable de estado. Entonces, antes del bucle, almacenamos las condiciones iniciales en el timeFrame en la posición t0.

Basados en la forma en la que antes almacenamos los TimeSeries, sería tentador hacer esto:

```
frame[system.t0] = system.init
```

Pero cuando usamos los corchetes con TimeFrame o DataFrame seleccionamos una columna, no una fila, por ejemplo, podemos seleccionar la columna de los susceptibles con:

```
frame['S']
```

Para seleccionar una fila debemos usar .row de esta forma:

```
frame.row[system.t0] = system.init
```

Dado que el valor system.init es un objeto State que tiene tres elementos, esos tres elementos se cargarán en cada una de las columnas de la fila seleccionada en el TimeFrame, esto es, asigna el valor S del system.init a la columna S, e igual con los otros dos.

Podemos usar la misma característica para escribir el bucle más conciso, asignando el State que obtenemos desde update_func directamente a la siguiente fila del frame.

Finalmente retornamos el frame completo:

```
results = run_simulation(system, update_func)
```

Ahora podemos utilizar la misma función de graficación anterior pasando los tres argumentos por separado:

```
plot_results(results.S, results.I, results.R)
```

Ver el código sir1.py

4.7 Optimizando

Antes hemos visto el modelo SIR para enfermedades infecciosas y lo usamos para modelar la plaga Freshman en Olin. Ahora veremos algunas métricas para cuantificar los efectos de la enfermedad y algunas intervenciones tendientes a reducir los efectos.

4.7.1 Métricas

Cuando tenemos un grafico de timeseries vemos todo lo que ha ocurrido cuando el modelo corrió, pero a menudo queremos reducirlo a unos pocos números que sumaricen el resultado.

En el modelo SIR podríamos querer saber el tiempo hasta el pico de la enfermedad, en número de personas que se han enfermado en el pico, el número de estudiantes que seguirán enfermos al final del semestre, o el numero total de estudiantes que se han enfermado en cualquier instante de tiempo.

Como un ejemplo, podríamos centrarnos en este último punto, el total de estudiantes enfermos, y podríamos considerar algunas intervenciones para minimizarlo.

Cuando una persona se enferma, se mueve de S a I, así que podemos obtener el número total de infectados calculando la diferencia entre S al principio y al final:

```
def calc_total_infected(results, system):  
    return results.S[system.t0] - results.S[system.t_end]
```

Si tenemos una TimeSeries llamada S, podríamos calcular el valor mayor de la serie de esta forma:

```
mayor = S.max()
```

A su vez, el índice de la serie es el momento o tiempo en el que ocurre cada evento, por lo que el tiempo en el que ocurre el mayor numero de infectados es:

```
tiempo_mayor = S.idxmax()
```

4.7.2 Inmunización

Modelos como este son especiales para responder preguntas del tipo ¿qué pasaría si?. Consideremos por ejemplo la inmunización.

Supongamos que hay una vacuna que causa que un estudiante se haga inmune a la Freshman plague sin verse infectado. ¿Cómo podría modificar el modelo este efecto?

Una opción es tratar a la inmunización como una forma rápida de pasar un estudiante de S a R sin pasar por los I.

Podríamos implementar esta función:

```

def add_immunization(system, fraction):
    system.init.S -= fraction
    system.init.R += fraction

```

Cada vez que ejecutemos esta función estaremos "vacunando" a una persona.

Si asumimos que el 10 % de los estudiantes están vacunados al principio del semestre, y la vacuna es 100 % efectiva, podemos simular el efecto con estas líneas:

```

system2 = make_system(beta, gamma)
add_immunization(system2, 0.1)
results2 = run_simulation(system2, update_func)

```

Podríamos graficar ambos modelos, el anterior que no tenía vacunas, y el nuevo, a ver qué ocurre.

En este caso vamos a trabajar únicamente con S, recordemos que entre el S inicial y el final tenemos la cantidad total de personas que han sido infectadas. Figura 32.

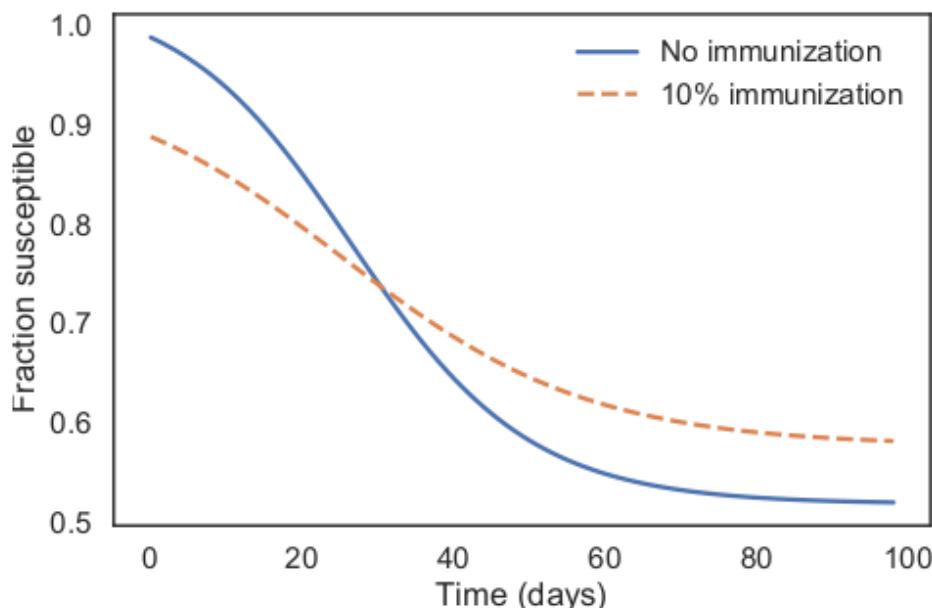


Figura 32: SIR - Immunize

Sin la vacuna casi el 47 % de la población se infectará en algún momento, mientras que con la vacuna sólo el 31 % se infectará.

Ahora veamos qué pasa si administramos más vacunas. Esta función barre un rango de porcentajes de personas vacunadas y grafica los resultados:

```

def sweep_immunity(immunize_array):
    sweep = SweepSeries()
    for fraction in immunize_array:
        sir = make_system(beta, gamma)
        add_immunization(sir, fraction)
        results = run_simulation(sir, update_func)
        sweep[fraction] = calc_total_infected(results, sir)
    return sweep

```

El parámetro de `sweep_immunity` es un array de tasas de vacunación. El resultado es un objeto `SweepSeries` que mapea para cada tasa de vacunación a la fracción resultante de estudiantes infectados. Veamos la gráfica en la Figura 33

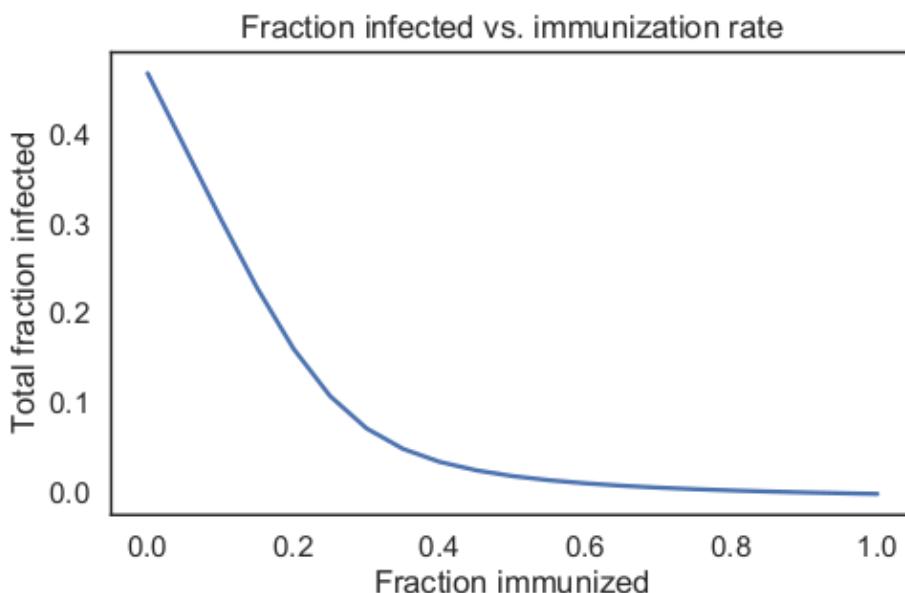


Figura 33: SIR - Immunize vs Infected

Se puede ver que a medida que la tasa de estudiantes vacunados aumenta, el número de infecciones disminuye abruptamente.

Si el 40 % de los estudiantes están vacunados, menos del 4 % del total se enfermará. Esto es porque la vacuna tiene dos efectos: protege a las personas que se han vacunado de enfermarse, y además protege al resto de la población de contagiarse.

Reduciendo el número de casos susceptibles e incrementando el numero de casos "resistentes" hacemos más difícil la propagación de la enfermedad porque una fracción de los contactos entre personas no transmite la enfermedad.

Este fenómeno se llama **inmunidad del grupo** ([Herd Immunity](#)) y es un elemento importante para la salud pública.

Ahora bien, la disminución abrupta de la curva anterior es muy buena, pero también tiene lo malo. Es buena porque significa que no tenemos que vacunarlos a todos, y las vacunas pueden proteger a todo el grupo sin siquiera ser el 100 % efectivas.

Pero tiene su punto malo porque una pequeña disminución de la vacunación puede causar un crecimiento abrupto de las infecciones. En este ejemplo si reducimos la fracción de los vacunados del 80 % al 60 % podría no ser tan malo, pero si reducimos el porcentaje de vacunados del 40 % al 20 % (un 20 % igual que antes) se disparará el brote infeccioso afectando a más del 15 % de la población. Para una enfermedad grave como el sarampión (sólo por nombrar una) esto podría causar una catástrofe a la salud pública.

Uno de los modelos como este se usan para demostrar el fenómeno de la inmunidad de grupo y para predecir el efecto de intervenciones como la vacunación. Otro uso es para evaluar determinadas guías en la toma de decisiones.

4.8 ¿Nos lavamos las manos?

Supongamos que estamos encargados de mantener la salud de nuestros estudiantes, y que tenemos sólo un presupuesto de \$1200 para combatir la Freshman Plague. Tenemos dos opciones con esta cantidad de dinero:

1. Podemos pagar vacunas, con un promedio de \$100 cada dosis.
2. Podemos pagar una campaña para recordarles a los estudiantes lavarse las manos.

Ya vimos cómo el modelo puede ser afectado por la vacunación. Ahora veamos cómo podemos modelar una campaña para lavarse las manos.

Tenemos que responder dos preguntas:

1. ¿Cómo debería introducirse el efecto de lavarse las manos en el modelo?

2. ¿Cómo podría cuantificarse el efecto de gastar el dinero en una campaña para lavarse las manos?

Para mantener la simplicidad, asumamos que tenemos datos históricos de campañas similares en otros colegios que muestran que una buena campaña puede cambiar el comportamiento de los estudiantes y reducir la tasa de infección en un 20 %.

En términos del modelo, lavarse las manos tiene un efecto en la reducción de beta. Esta no es la única manera en la que podríamos incorporar el efecto, pero parece razonable y fácil de implementar.

Ahora tenemos que modelar la relación la cantidad gastada en la campaña y su efectividad. De nuevo, supongamos que tenemos datos de escuelas que sugieren:

- Si gastamos \$500 en posters, materiales de lectura, tiempo del personal, podemos cambiar el comportamiento de los estudiantes de modo que se reduzca la efectividad del valor beta un 10 %.
- Si gastamos \$1000 el total de la disminución de beta es del 20 %.
- Por encima de \$1000 el beneficio adicional es muy pequeño.

Luego veremos en el código cómo definir una función logística para este caso, pero sí podemos definir por ahora una función que calcule el factor en el que el parámetro beta se reducirá:

```
def compute_factor(spending):  
    return logistic(spending, M=500, K=0.2, B=0.01)
```

La función logística utilizada es una función denominada Función de Richards, o [Función Logística Generalizada](#). Esta función permite modelar crecimiento, y es una extensión o generalización de funciones logísticas o sigmoideas que dan la posibilidad de crear curvas tipo "S" de manera más flexible.

Los parámetros elegidos inicialmente son:

- **M=\$500** puesto que si el crecimiento fuera lineal, y con \$1000 se alcanza el 20 % de disminución de Beta, entonces con \$500 se alcanza la mitad. En la curva tipo

S marcaría el punto de inflexión, el valor en el que la tasa de crecimiento pasa de acelerar a desacelerar.

- **K=0.2** porque vamos a limitar el porcentaje de reducción de Beta en 20 % (relacionado con el valor anterior).
- **B=0.01** para definir la suavidad de la curva (tasa de crecimiento media)

Usemos ahora el factor retornado para escribir una función `add_hand_washing` que tome el objeto System y una determinada cantidad de dinero y modifique el parámetro beta para modelar el efecto de lavarse las manos:

```
def add_hand_washing(system, spending):  
    factor = compute_factor(spending)  
    system.beta *= (1 - factor)
```

Ahora probemos barrer un rango de valores para gastos y usar la simulación para determinar el efecto:

```
def sweep_hand_washing(spending_array):  
    sweep = SweepSeries()  
    for spending in spending_array:  
        sir = make_system(beta, gamma)  
        add_hand_washing(sir, spending)  
        results, run_simulation(sir, update_func)  
        sweep[spending] = calc_total_infected(results, sir)  
    return sweep
```

Ahora veamos cómo esto se ejecuta:

```
spending_array = linspace(0, 1200, 20)  
infected_sweep = sweep_hand_washing(spending_array)
```

Si graficamos estos valores (Figura 34) veremos el porcentaje de reducción de la tasa de infecciones respecto del dinero invertido en la campaña:

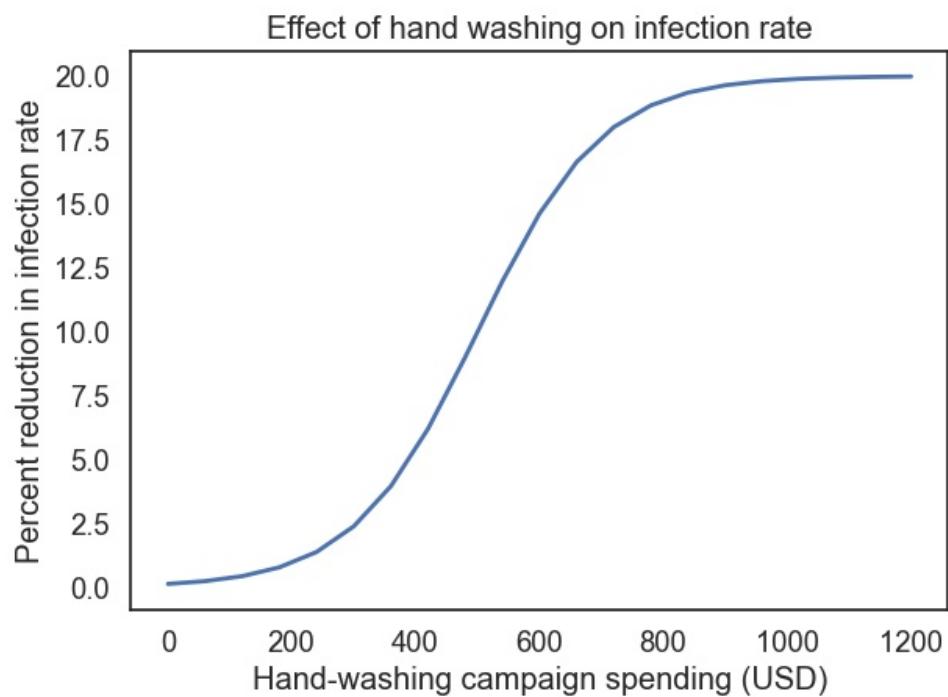


Figura 34: SIR - Campaña de lavado de manos

Ahora veamos la Figura 35, el total de infectados respecto de la cantidad de plata invertida en la campaña de lavarse las manos:

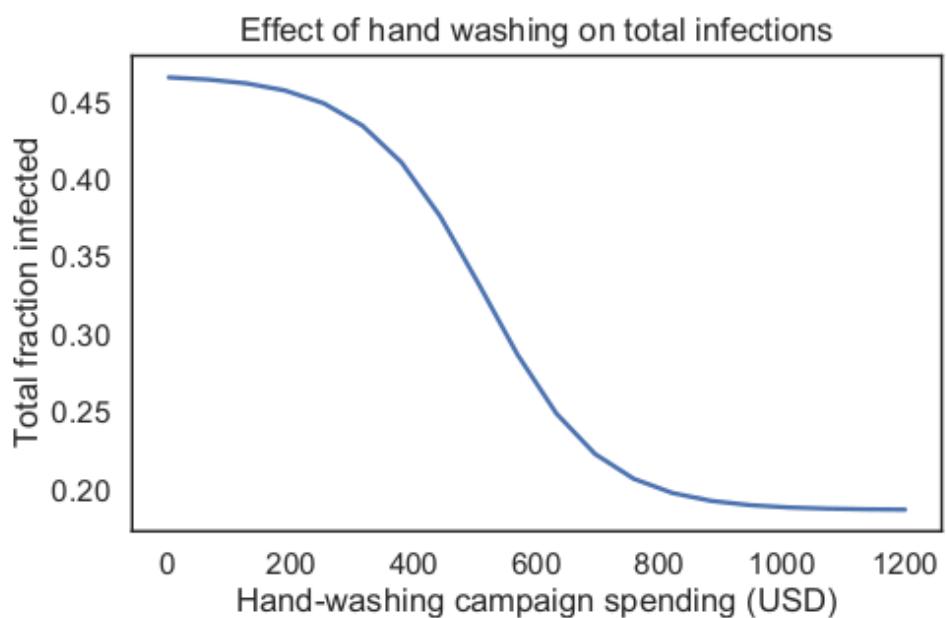


Figura 35: SIR - Campaña de lavado de manos vs infectados

Para gastos de campaña menores a \$200 el efecto es mínimo, pero para gastos de \$800 el efecto es considerable. De ahí en adelante prácticamente no hay diferencia.

4.9 Optimización

Ahora pongamos todos los parámetros juntos. Con un presupuesto fijo de \$1200 podemos decidir cuánto gastar en campaña de lavado de manos y cuánto gastar en vacunación.

Veamos algunos parámetros:

```
num_students = 90
budget = 1200
price_per_dose = 100
max_doses = int(budget / price_per_dose)
```

La fracción de presupuesto/precio_por_dosis no va a ser un valor entero, por eso usamos `int()` para convertirlos a enteros redondeando (no hay "media dosis").

Barreremos ahora el rango de posibles dosis:

```
dose_array = linrange(max_doses, endpoint=True)
```

En este ejemplo llamaremos `linrange` con solo un argumento. Esto retorna un arreglo **NumPy** con los enteros desde 0 hasta la cantidad máxima de dosis. Con un argumento `endpoint=True`, el resultado incluirá ambos extremos del intervalo.

Entonces podemos correr la simulación para cada elemento del arreglo de dosis de esta manera:

```
def sweep_doses(dose_array):
    sweep = SweepSeries()
    for doses in dose_array:
        fraction = doses / num_students
        spending = budget - doses * price_per_dose
        sir = make_system(beta, gamma)
        add_immunization(sir, fraction)
        add_hand_washing(sir, spending)
        run_simulation(sir, update_func)
```

```

sweep[doses] = calc_total_infected(sir)

return sweep

```

Por cada numero de dosis calcularemos la fracción de los estudiantes que podemos vacunar, fraction, y la cantidad de dinero que nos sobraría para invertir en una campaña de lavado de manos, spending.

Ahora correremos la simulación y veremos estas cantidades, y almacenaremos el número de infecciones. Veamos el efecto en la Figura 36

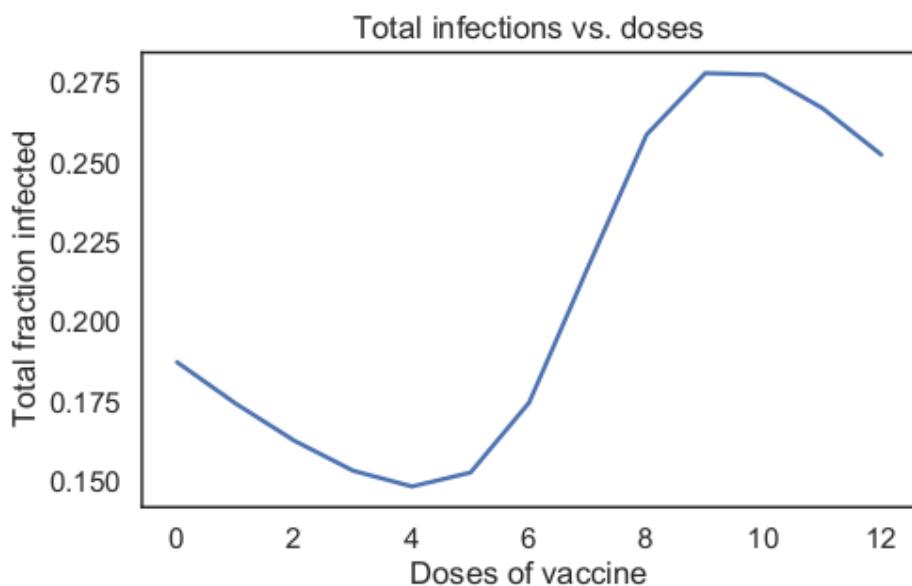


Figura 36: SIR - Vacunación

Esta figura nos muestra el resultado. Si no compramos dosis de vacunas y gastamos todo en campaña de lavado de manos, la cantidad de infectados será del 19 % aproximadamente. Con 4 dosis compradas tendremos el mínimo de infectados, y estaremos gastando \$800 en campaña de lavado de manos. Este es el punto óptimo de inversión con mayor efectividad.

También podemos incrementar el número de dosis, reduciendo el dinero restante en campaña de lavado de manos, pero nos da peores resultados, aunque los resultados son interesantes. Con 10 dosis compradas el efecto de la inmunidad de grupo comienza a estabilizarse, y a partir de ahí empieza a reducirse la cantidad de infectados. Veamos el código sir2.py

4.10 Barriendo dos parámetros

Ya hemos presentado un modelo SIR de enfermedad infeccional, específicamente el modelo Kermack-McKendrick. Extendimos el modelo para incluir vacunación y el efecto de una campaña de lavado de manos, y usamos este modelo extendido para administrar un presupuesto de manera óptima, esto es, minimizar el número de infecciones.

Pero hemos asumido que los parámetros del modelo, como la tasa de contacto y la tasa de recuperación eran conocidos. Ahora veremos cómo podemos explorar el comportamiento del modelo variando estos parámetros, usando análisis para comprender cómo se relacionan, y proponer un método para usar los parámetros estimados.

4.10.1 Barriendo Beta

Recordemos que Beta es la tasa de contacto, que captura la frecuencia de interacción entre personas y la porción de estas que resultan en una nueva infección.

Si N es el tamaño de la población, y s es la fracción de la misma que es susceptible a contagios, sN es el número de personas susceptibles, y βsN es el número de esos contactos que resultan en una nueva infección.

Si incrementamos β se espera que el total de infectados se incremente también. Para cuantificar esta relación, crearemos un rango de valores para β :

```
beta_array = linspace(0.1, 1.1, 11)
```

Entonces podemos correr la simulación para cada valor de beta:

```
for beta in beta_array:
    sir = make_system(beta, gamma)
    run_simulation(sir, update1)
    print(sir.beta, calc_total_infected(sir))
```

Podemos encapsular el código en una función que almacene los resultados en un objeto SweepSeries de esta forma:

```
def sweep_beta(beta_array, gamma):
    sweep = SweepSeries()
    for beta in beta_array:
```

```

system = make_system(beta, gamma)
run_simulation(system, update1)
sweep[system.beta] = calc_total_infected(system)
return sweep

```

Ahora podemos correr esta función de esta manera:

```
infected_sweep = sweep_beta(beta_array, gamma)
```

Y graficar los resultados: (Figura 37).

```

label = ' gamma = ' + str(gamma)
plot(infected_sweep, label=label)

```

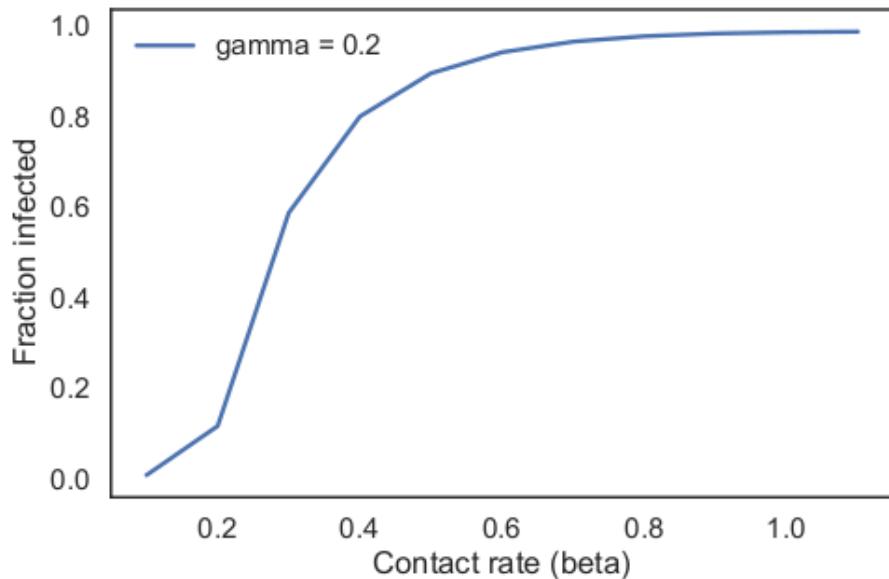


Figura 37: SIR - Tasa de contacto β

Esta figura muestra los resultados. Recordemos que representa un barrido de parámetros, no un time series, de modo que el eje x es un parámetro, beta, no el tiempo.

Cuando beta es pequeño, la tasa de contacto es baja y el brote nunca se alcanza. El total de infectados es cercano a cero en este caso. Cuando beta se incrementa, se alcanza un umbral de 0.3 donde la fracción de los infectados se incrementa rápidamente. Cuando beta excede el 0.5, mas del 80 % de la población ya está infectada.

4.10.2 Barriendo gamma

Veamos qué pasa cuando barremos el parámetro gamma. De nuevo, usaremos un linspace:

```
gamma_array = linspace(0.1, 0.7, 4)
```

Y corremos el sweep_beta para cada valor de gamma:

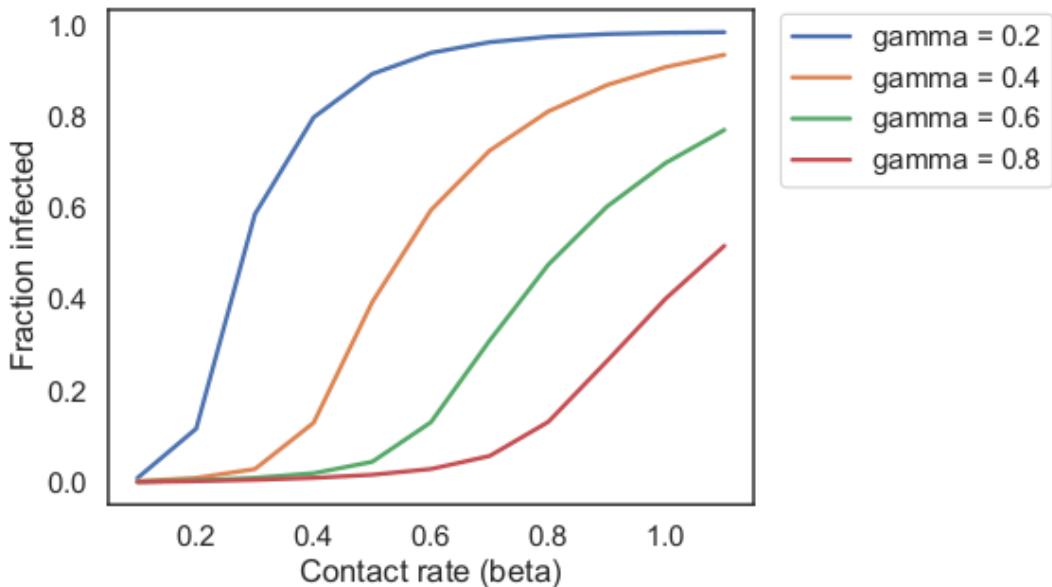


Figura 38: SIR - Barriendo Gamma

Cuando gamma es pequeño, la tasa de recuperación es baja también, esto significa que las personas se recuperan más lentamente, y permanecen infectadas por más tiempo en promedio. En este caso, incluso una baja tasa de contacto resulta en una epidemia.

Cuando gamma es alto, beta tiene que ser también grande para que todo funcione, es decir, si tenemos una alta tasa de recuperación, la tasa de contacto también será elevada.

4.10.3 Usando SweepFrame

Ya vimos que podíamos barrer un rango de gamma para cada uno de los valores. Esto se llama barrido en dos dimensiones.

Si queremos almacenar los resultados y luego graficarlos es más cómodos usar un SweepFrame, que es un tipo de DataFrame en el que las filas representan un parámetro y las columnas el otro, y los valores contienen métricas de cada simulación.

Una función de ejemplo sería esta:

```
def sweep_parameters(beta_array, gamma_array):  
    frame = SweepFrame(columns=gamma_array)  
    for gamma in gamma_array:  
        frame[gamma] = sweep_beta(beta_array, gamma)  
    return frame
```

La función `sweep_parameters` toma como parámetros un arreglo de beta y uno de gamma, y los combina.

Crea un SweepFrame para almacenar los resultados, uno por columna para cada valor de gamma y uno por fila para cada valor de beta.

Cada vez que cicla simula todos los valores de beta para un valor fijo de gamma, y almacena el resultado en cada columna nueva.

Al final el SweepFrame almacena la fracción de los estudiantes infectados para cada par de valores, beta y gamma.

Podemos ejecutarla así:

```
frame = sweep_parameters(beta_array, gamma_array)
```

Y podemos graficar así:

```
for gamma in gamma_array:  
    label = ' gamma = ' + str(gamma)  
    plot(frame[gamma], label=label)
```

Alternativamente podemos hacer esto:

```
for beta in [1.1, 0.9, 0.7, 0.5, 0.3]:  
    label = ' = ' + str(beta)  
    plot(frame.row[beta], label=label)
```

Y esto nos genera la gráfica de la Figura 39

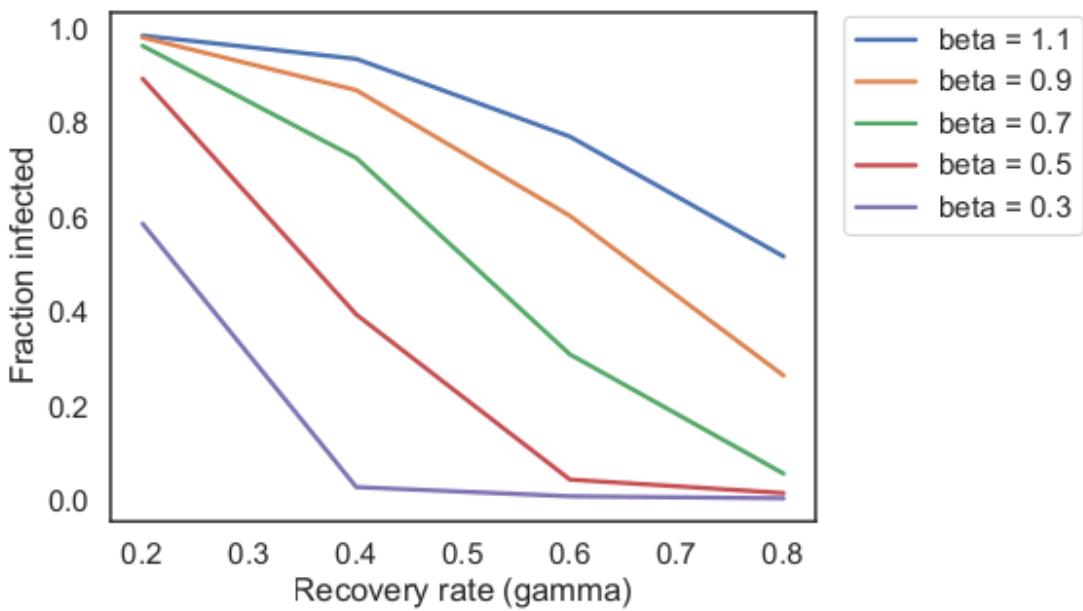


Figura 39: SIR - Barriendo Gamma

En esta gráfica se puede ver la fracción de los infectados respecto del parámetro gamma (tasa de recuperación) para diferentes valores de beta (tasa de contacto). Cuando la tasa de recuperación gamma es baja, la fracción de los infectados es alta, y más alta es cuando mayor sea la tasa de contacto beta. Cuando la tasa de recuperación es alta, cantidad de gente infectada será baja, más baja cuando menor sea la tasa de contacto beta.

Este ejemplo demuestra un uso de SweepFrame, podemos correr el análisis una vez y almacenar los resultados para generar diferentes gráficas luego.

Otra forma de visualizar los resultados es un barrido de dos dimensiones en un gráfico de contorno, que muestra los parámetros en cada eje y lineas de contorno, o sea, lineas de valores constantes. En este ejemplo el valor es la fracción de los estudiantes infectados.

La librería ModSim provee la función **contour** que toma un SweepFrame como parámetro y genera un diagrama de contorno, como el que se ve en la Figura 40

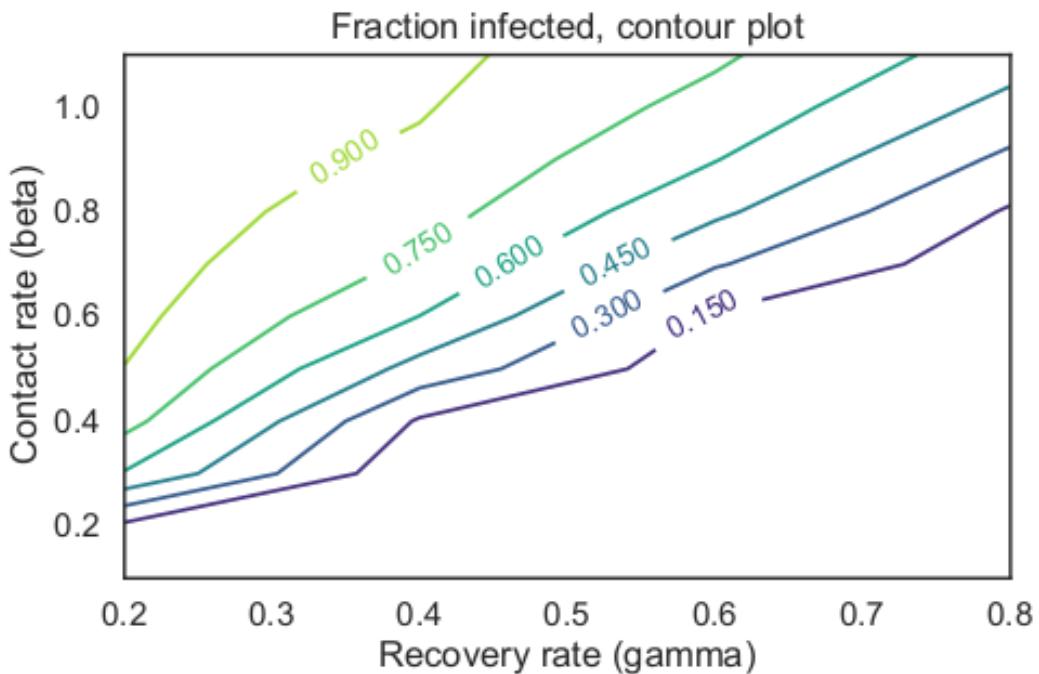


Figura 40: SIR - Diagrama de contorno

Las tasas de infección son bajas abajo a la derecha, donde los ratios de contacto y recuperación son altos. Las líneas se incrementan en la medida en que nos movemos hasta la esquina superior izquierda, donde la velocidad de contacto es alta y la velocidad de recuperación es baja. Los números en las líneas representan la fracción de la población infectada.

Esta figura sugiere que podría haber una relación entre beta y gamma que determina el resultado del modelo. De hecho, la hay, la veremos luego.

4.11 Análisis

Anteriormente usamos simulación para predecir el efecto de una enfermedad infecciosa en una población susceptible y diseñamos algunos métodos para intervenir y reducir esa tasa de infección, a saber, vacunación y campañas de lavado de manos.

Ahora usaremos análisis para investigar la relación entre los parámetros beta y gamma, y los resultados de la simulación.

La última figura nos sugería que hay una relación entre los parámetros del modelo SIR, beta y gamma, que determinaba el resultado de la simulación, la fracción de los estudiantes infectados. Pensemos ahora cómo puede ser esta relación:

- Cuando beta excede a gamma, hay más contactos (y por consiguiente, mayor probabilidad de infecciones) que recuperaciones por unidad de tiempo. La diferencia entre beta y gamma podría ser llamada exceso de tasa de contacto), en unidades de contacto por día.
- Como una alternativa, podríamos considerar a la fracción beta/gamma, que es el número de contactos por recuperación. Esto es porque el numerador y denominador están en la misma unidad, no tenemos unidades, no es una tasa de cambio respecto de algo, simplemente es un número, sin dimensión.

Esto de describir los sistemas en parámetros sin dimensión a veces es útil para moverse entre la simulación y el modelado. Es tan útil, de hecho, que hasta tiene nombre: *nondimensionalization*.

4.11.1 Explorando los resultados

Supongamos que tenemos un SweepFrame con una fila por cada valor de beta, y una columna por cada valor de gamma. Cada elemento en el SweepFrame es una fracción de los estudiantes infectados en una simulación con un beta y un gamma particulares.

Podríamos armar el sweepframe de esta forma: (Figura 41)

```
beta_array = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 ,  
             1.1]  
  
gamma_array = [0.2, 0.4, 0.6, 0.8]  
  
frame = sweep_parameters(beta_array, gamma_array)  
frame.head()
```

	0.2	0.4	0.6	0.8
0.1	0.010756	0.003642	0.002191	0.001567
0.2	0.118984	0.010763	0.005447	0.003644
0.3	0.589095	0.030185	0.010771	0.006526
0.4	0.801339	0.131563	0.020917	0.010780
0.5	0.896577	0.396409	0.046140	0.017640
0.6	0.942929	0.597902	0.132889	0.030292
0.7	0.966299	0.728470	0.311843	0.058824
0.8	0.978152	0.814460	0.478326	0.133589
0.9	0.984057	0.872270	0.605688	0.266890
1.0	0.986882	0.911669	0.701425	0.403751
1.1	0.988148	0.938680	0.773818	0.519583

Figura 41: SIR - SwapFrame

Y también podríamos mostrar todos los valores así:

```
for gamma in frame.columns:
    column = frame[gamma]
    for beta in column.index:
        frac_infected = column[beta]
        print(beta, gamma, frac_infected)
```

El frame que armamos tiene 4 valores de gamma para 11 valores de beta, es decir, el frame.

Armemos ahora una función que grafique para cada valor beta/gamma, nuestro parámetro no-dimensional, la cantidad de infectados:

```
def plot_sweep_frame(frame):
    """Plot the values from a SweepFrame.

    For each (beta, gamma), compute the contact number,
    beta/gamma
```

```

frame: SweepFrame with one row per beta, one column per gamma
"""

for gamma in frame.columns:
    column = frame[gamma]
    for beta in column.index:
        frac_infected = column[beta]
        plot(beta/gamma, frac_infected, 'ro')

```

Y si la usamos y graficamos: (Figura 42)

```

plot_sweep_frame(frame)
decorate(xlabel='Contact number (beta/gamma)', ylabel='Fraction
infected')
savefig('/tmp/grafico.jpg')

```

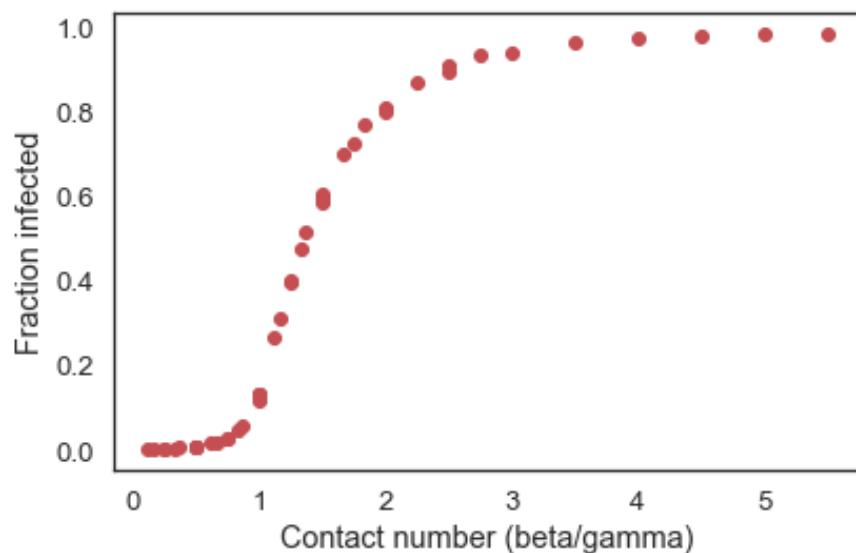


Figura 42: SIR - Contact Number

Esta figura muestra que el resultado se ajusta a una curva aproximadamente. Esto significa que podemos predecir la fracción de estudiantes que serán infectados basados en un simple parámetro como es el valor β/γ . No necesitamos saber los valores de β y γ por separado, de hecho.

4.12 Número de contacto (Beta/Gamma)

Recordemos primero que el número de infecciones en un día determinado es $\beta s i N$ y que el número de recuperados es $\gamma i N$. si dividimos esas cantidades, el resultado será $\beta s / \gamma$, que es el número de nuevos infectados respecto de los recuperados, como una fracción de la población (porcentaje).

Cuando una nueva enfermedad es introducida en una población susceptible, s es aproximadamente 1, es decir, prácticamente el 100 % de la población es susceptible de ser infectada, de modo que el número de personas infectada por cada persona enferma es aproximadamente β / γ .

A este número se lo conoce como "[número de contacto](#)" o "[número básico de reproducción](#)", y por convención se lo denota como R_0 , pero en el modelo SIR esta notación puede ser confusa, por lo que usaremos c en su lugar.

El gráfico realizado sugiere que hay una relación entre c y el número total de infecciones. Podemos derivar esta relación para analizar las ecuaciones diferenciales de este modelo poblacional, ecuaciones [4.1](#), [4.2](#) y [4.3](#)

De la misma forma que dividimos la tasa de contacto por la de infecciones para obtener una cantidad sin dimensiones c , ahora podemos dividir di/dt sobre ds/dt como planteamos al principio de esta sección para obtener la tasa de nuevos infectados respecto de los susceptibles:

$$\frac{\frac{di}{dt}}{\frac{ds}{dt}} = \frac{\beta s i - \gamma i}{-\beta s i}$$

Si comenzamos a operar ambos lados de la igualdad, cancelando términos, podemos llegar a una conclusión:

$$\begin{aligned}\frac{di}{ds} &= \frac{\beta si - \gamma i}{-\beta si} \\ \frac{di}{ds} &= \frac{\beta si}{-\beta si} - \frac{-\gamma i}{-\beta si} \\ \frac{di}{ds} &= -1 + \frac{\gamma}{\beta s}\end{aligned}$$

Multiplicando por $1/\gamma$ numerador y denominador del segundo término de la igualdad:

$$\begin{aligned}\frac{di}{ds} &= -1 + \frac{\frac{\gamma}{\gamma} \cdot \frac{1}{\gamma}}{\frac{\beta s}{\gamma} \cdot \frac{1}{\gamma}} \\ \frac{di}{ds} &= -1 + \frac{1}{\frac{\beta}{\gamma} \cdot s}\end{aligned}$$

Y como $\beta/\gamma = c$, nuestro número de contacto, la ecuación diferencial queda de esta forma:

$$\frac{di}{ds} = -1 + \frac{1}{c.s} \quad (4.4)$$

Dividir una ecuación diferencial por otra no es un movimiento muy obvio, pero en este caso es muy útil porque nos da una relación entre i , s y c que no depende del tiempo.

De esta relación podemos derivar una ecuación que relaciona c con el valor final de s . En teoría, esta ecuación hace posible inferir c , el número de contacto, observando el curso de una epidemia.

Si multiplicamos m.a.m. por ds obtenemos lo siguiente:

$$di = \left(-1 + \frac{1}{c.s} \right) ds$$

Si ahora integramos m.a.m. respecto de s :

$$i = -s + \frac{1}{c} \log s + q$$

Donde q es la constante de integración. Reordenemos los términos:

$$q = i + s - \frac{1}{c} \log s$$

Ahora veamos si podemos determinar qué es q . Al principio de la epidemia, si la fracción de los infectados es pequeña y casi todos los habitantes son susceptibles, podemos usar esta aproximación:

$$\begin{aligned} i(0) &= 0 \\ s(0) &= 1 \end{aligned}$$

Ahora con estos datos intentemos calcular q :

$$q = 0 + s - \frac{1}{c} \log 1$$

Dado que el $\log 1 = 0$, tenemos que $q = 1$.

Ahora bien, al final de la epidemia, en el tiempo infinito, asumiremos que la cantidad de infectados es cero puesto que se recuperaron todos, esto es, $s(\infty) = 0$.

La cantidad de susceptibles no la conocemos ya que la población puede evolucionar: $s(\infty) = s_\infty$.

Ahora tenemos esta ecuación:

$$q = 0 + s_\infty - \frac{1}{c} \log s_\infty$$

Resolviendo para c queda:

$$c = \frac{\log s_\infty}{s_\infty - 1}$$

Esta ecuación relaciona c con s_∞ y hace posible estimar c basado en los datos, y posiblemente predecir el comportamiento de futuras epidemias.

4.13 Análisis y simulación

Comparemos este resultado analítico con el resultado de la simulación. Crearemos un arreglo de valores de s_∞ :

```
s_inf_array = linspace(0.0001, 0.9999, 31)
```

y calcularemos el correspondiente valor de c :

```
c_array = log(s_inf_array) / (s_inf_array - 1)
```

Para obtener el total de infectados calculamos la diferencia entre $s(0)$ y s_∞ . Luego almacenaremos los resultados en un objeto Series (recordemos que $s(0)$ era 1):

```
frac_infected = 1 - s_inf_array
frac_infected_series = Series(frac_infected, index=c_array)
```

Si graficamos estos datos veremos lo siguiente: Figura 43

```
plot_sweep_frame(frame)
plot(frac_infected_series, label='Analysis')
```

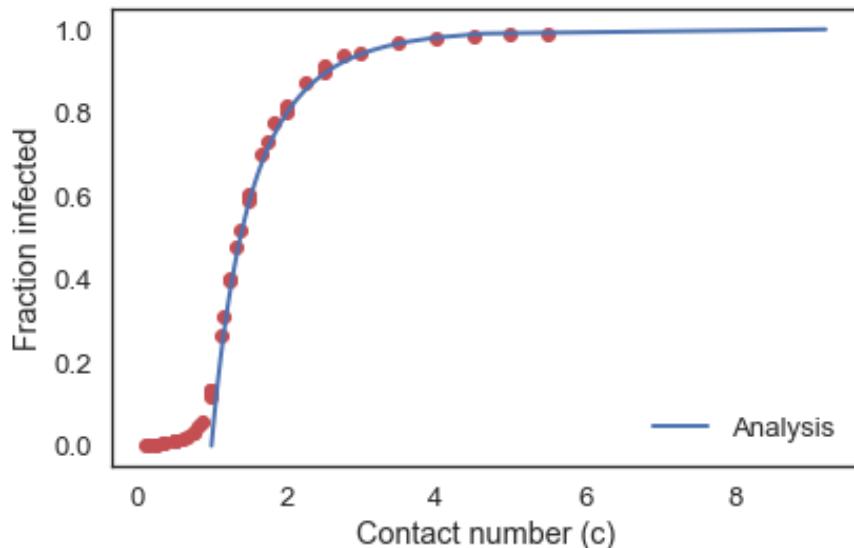


Figura 43: SIR - Simulación vs Cálculo exacto

Este gráfico muestra la fracción de los infectados como una función del número de contacto, resolviendo con simulación (puntos rojos) y análisis (curva azul).

Como sabemos, los objetos Series tienen un index y sus valores asociados, en este caso el index es c , y los valores asociados a cada valor de c son la fracción de infectados.

Analizando la gráfica, cuando el número de contacto es mayor a 1, la simulación y el análisis coinciden, mientras que cuando el número de contacto es menor a 1, no lo hacen. El análisis indica que con un c menor a 1 no deberían haber infecciones, la simulación muestra que sí hay un pequeño número de infecciones con esos valores de c .

La razón de esta discrepancia es que la simulación divide el tiempo en series discretas de días, mientras que el análisis trata al tiempo como una cantidad constante.

Entonces **¿qué modelo es mejor?**

Y la respuesta es: probablemente ninguno :P

Cuando el número de contacto es pequeño, el progreso temprano de la epidemia depende mucho del escenario inicial si hay suerte y se detecta a tiempo, el primer infectado (el paciente cero), no infectará a nadie, y no habrá epidemia. Si no hay suerte, el paciente cero podría tener un amplio grupo de amigos, o podría trabajar en un restaurante o escuela donde tenga contacto con mucha gente, o no, para lo cual el inicio de la epidemia puede variar. Por esto es que para poder saber qué pasa en esos primeros momentos necesitamos un modelo más detallado. Pero para un número de contacto superior a 1, el modelo SIR descripto puede ser una excelente aproximación.

4.13.1 Estimando el número de contacto

El gráfico anterior muestra que si sabemos el número de contacto podemos calcular la fracción de los infectados. Pero también podemos leer esta gráfica de otra manera. Al final de la epidemia, si podemos estimar la fracción de la población que alguna vez estuvo infectada, podemos usarla para estimar el número de contacto.

En teoría podemos, en la práctica puede no funcionar tan bien dependiendo de la forma de la curva.

Cuando el número de contacto es cercano a 2, la curva es bastante escalonada o vertical, lo que significa que pequeños cambios en c producen grandes cambios en el número de infecciones. Si observamos que la fracción de los infectados está entre el 20 % o 30 % y el 80 %, adivinaríamos que c es cercano a 2.

Por otro lado, para un número de contacto mayor, cerca de la totalidad de la población estará infectada, de modo que la curva será casi plana o asintótica. En este caso no seremos capaces de saber qué valor tiene c con precisión, ya que un valor de c mayor que 3 nos dará un porcentaje de la población casi igual. Afortunadamente esto no ocurre en el mundo real, muy pocas epidemias afectan al 90 % de la población.

Así, el modelo SIR tiene sus limitaciones, sin embargo provee una visión dentro del comportamiento de las enfermedades infecciosas, especialmente el fenómeno de la inmunidad de grupo. Como vimos antes, si conocemos los parámetros del modelo, podemos usarlos para evaluar posibles intervenciones, además podríamos ser capaces de usar datos reales de los resultados tempranos de la epidemia para estimar esos parámetros.

CAPÍTULO 5

Osciladores

5.1 Método de Euler

El método de Euler es un método que permite resolver ecuaciones diferenciales ordinarias (EDO), y consiste en encontrar soluciones numéricas a una ecuación diferencial en un intervalo de valores discretos definidos. Supongamos la siguiente derivada:

$$\frac{du}{dt} = f(u, t)$$

Con valores iniciales $u_0 = 0$ e $u_0 = 0$.

Supongamos que los valores de x van desde u_0 hasta u_n .

Estos valores serían $u_1, u_2, u_3 \dots u_n$.

Cada nuevo valor de x se obtiene como $u_i = u_{0+i}.h$, donde h es la longitud de cada intervalo:

$$h = \frac{x_n - x_0}{n}$$

Con las condiciones iniciales podemos calcular la derivada primera al inicio:

$$\dot{y}(x_0) = f(x_0, y_0)$$

Como sabemos, esta derivada representa la pendiente de la recta tangente a la curva $f(x_0, y_0)$ en el punto inicial $A_0 = (x_0, y_0)$

Entonces se hace una predicción aproximada del valor de la función $y(x)$ en el siguiente punto:

$$y(x_1) \approx y_1$$

$$\text{con } y_1 = y_0 + (x_1 - x_0).f(x_0, y_0) = y_0 + h.f(x_0, y_0)$$

Con esto ya tenemos el valor de $A_1 = (x_1, y_1)$

Ahora repetimos el mismo procedimiento para los demás valores A_i comprendidos en el intervalo definido.

El i -ésimo valor puede calcularse como:

$$y_i = y_{i-1} + h.f(x_{i-1}, y_{i-1})$$

Así, despejando la función f :

$$f(x_{i-1}, y_{i-1}) = \frac{dy_i}{dx_i} = \frac{y_i - y_{i-1}}{h}$$

La curva aproximada respecto de la curva original podría ser algo así (Fig. 44):

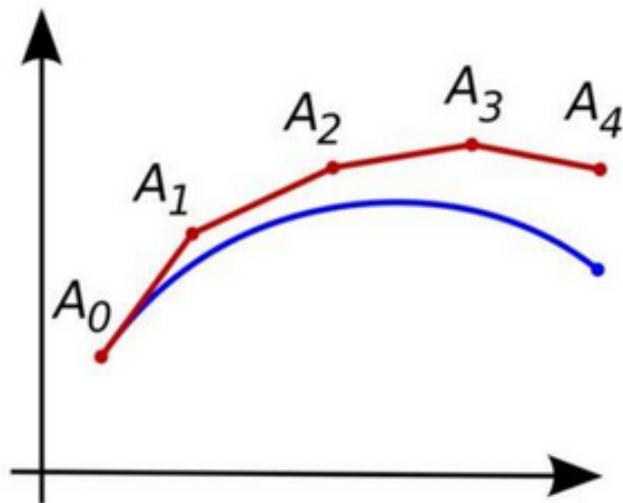


Figura 44: Método de Euler

5.1.1 `odeint`: resolviendo ODE's con Python

Las ecuaciones diferenciales ordinarias, u ODE (Ordinary Differential Equation), pueden resolverse de manera analítica, o gráfica y numérica. La resolución analítica ya se ha estudiado en materias de Cálculo en años anteriores de la carrera, así que nos centraremos en la resolución numérica y gráfica utilizando, particularmente, Python.

Consideremos una ecuación diferencial simple:

$$\frac{dx}{dt} = \frac{x - t}{x + t}$$

Por el método de Euler visto antes, podemos aproximar la derivada de manera discreta de la siguiente manera:

$$\frac{x_{i+1} - x_i}{h} = \frac{x - t}{x + t}$$

Despejando x_{i+1})

$$x_{i+1} = x_i + h \left(\frac{x_i - t_i}{t_i + x_i} \right)$$

Con esta ecuación recurrente podemos iterar para valores $x_0, x_1, x_2, \dots, x_n$. En vez de hacerlo de manera manual con un bucle en nuestro código, vamos a utilizar el método

`odeint` del módulo Scipy. `odeint` usa algunas mejoras y técnicas más avanzadas para resolver este tipo de ecuaciones, es decir, va un poco más allá de un simple método de Euler.

Por ejemplo, este código permite obtener una tabla de valores para x_i y t_i . Como generalmente se considera a t la variable independiente, esta no será la excepción:

```
# dx/dt = (x-t)/(x+t), x(0)=1

from scipy.integrate import odeint
f = lambda x,t: (x - t)/(x + t)    # f(t,x)
x0 = 1.                           # cond. inic.
a,b,h = 0, 1, 0.1                 # intervalo [a,b], delta t = h
tp = arange(a, b+h, h)           # --
xp = odeint(f, x0, tp)           # solve
```

La tabla de datos que nos devuelve, para cada instante de tiempo:

```
tp =  0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

xp =  [1.]  [1.09112676]  [1.16784164]  [1.23348808]  [1.29014358]
      [1.33920916]  [1.38168493]  [1.41831529]  [1.44967216]  [1.47620596]
      [1.49827843]
```

5.2 Sistemas no lineales de segundo orden

Cuando se estudian los sistemas lineales se pretende *predecir* o inferir el comportamiento futuro de un sistema ante diversas condiciones iniciales. Si los sistemas dinámicos son lineales, el uso de las variables de estado permiten describir la salida para cualquier tiempo dado, y de manera adecuada, teniendo en cuenta las variables de entrada.

La mayoría de los sistemas naturales son, sin embargo, no lineales, pero dada la simplicidad y escalabilidad de la matemática lineal, se estudian en general linealizándolos por tramos. Esta linealización únicamente es válida alrededor de las condiciones del sistema planteado, es decir, los valores predichos únicamente serán válidos para el modelo en el tramo linealizado.

Otro punto a tener presente es el comportamiento ante incertidumbre (error) y perturbaciones en los parámetros de entrada. En un sistema pequeñas perturbaciones de las variables de entrada no producen grandes cambios en las salidas, mientras que en un sistema no lineal pequeños cambios en las entradas y los parámetros pueden generar grandes variaciones en las salidas. Por esto es necesario conocer con precisión las variables de entrada y las condiciones de trabajo, y esto a veces no es posible.

5.3 Osciladores

Muchos sistemas físicos se pueden representar como osciladores, tales como el sistema masa resorte amortiguador, un péndulo, o circuitos electrónicos.

La forma más sencilla de modelarlos es mediante matemáticas, en una ecuación diferencial de segundo orden como la siguiente:

($u(t)$ es la elongación, $v(t)$ es la velocidad)

$$\ddot{u}(t) + \omega^2 \cdot u(t) = 0$$

Veamos el ejemplo del M-R-A:

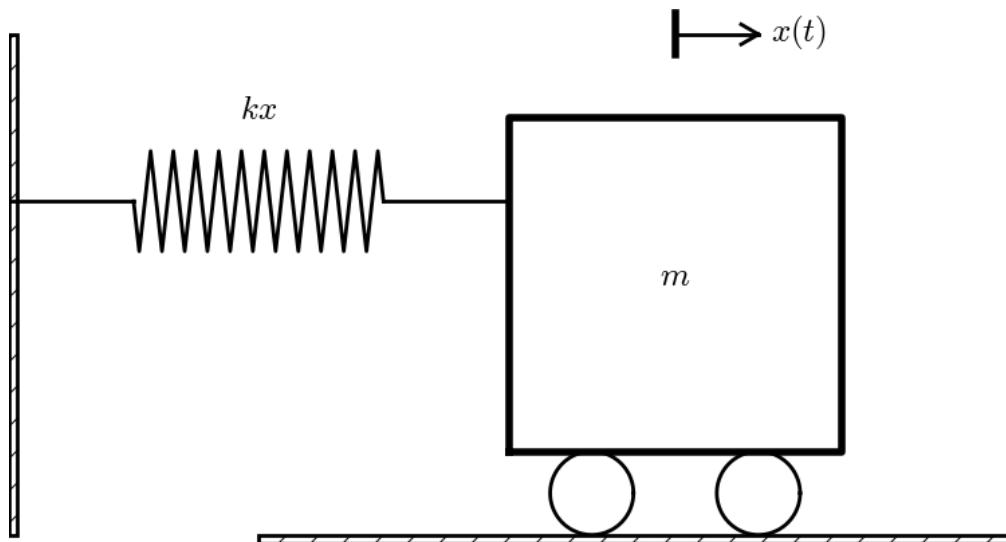


Figura 45: Sistema M-R-A ideal

En muchos sistemas de ingeniería basados en osciladores disponen de ecuaciones diferenciales que permiten comprenderlos, controlarlos y predecir su funcionamiento. Co-

menzaremos con un modelo simple que captura la dinámica esencial de un sistema oscilatorio, como es el caso del sistema m-r-a mostrado en la figura anterior.

Este sistema dispone de un cuerpo con masa m fijado a un resorte cuya constante elástica es k , y vamos a considerar que el cuerpo no tiene fricción sobre la superficie horizontal (los rodillos debajo del cuerpo indican esto). Cuando el resorte se estira o se comprime, el cuerpo comienza a oscilar sobre la superficie. Más precisamente, $x(t)$ va a ir obteniendo diferentes valores de desplazamiento del cuerpo en el eje x . El cuerpo no está en movimiento cuando $x = 0$, por lo que en este punto se lo considera posición de equilibrio.

La fuerza del resorte es $-k.x$, donde k es la constante elástica que debemos medir.

Asumiremos que no hay otras fuerzas como fricción, por lo que por la segunda ley de Newton:

$$F = m.a$$

Teniendo en cuenta que $F = -k.x$ y que la aceleración es $a = \ddot{x}$ tenemos:

$$-k.x = m.\ddot{x}$$

Que puede ser escrito como:

$$\ddot{u}(t) + \omega^2.u(t) = 0$$

$$\text{Con } \omega = \sqrt{\frac{k}{m}}$$

La ecuación anterior es una ecuación diferencial de segundo orden, y por lo tanto necesita dos condiciones iniciales para resolverse:

$$x(0) = X_0 \text{ (desplazamiento inicial)} \quad \dot{x}(0) = 0 \text{ (velocidad inicial)}$$

La solución exacta de esta ecuación es: $x(t) = X_0 \cdot \cos(\omega.t)$

Esto puede ser fácilmente verificable reemplazando este valor en la ecuación anterior, y chequeando las condiciones iniciales. (agregar verificación) !!

Esta ecuación diferencial además aparece en muchos otros contextos. Un ejemplo clásico es el péndulo simple, cuya ecuación es la siguiente:

$$m \cdot L \cdot \ddot{\theta} + m \cdot g \cdot \sin(\theta) = 0$$

Donde m es la masa del péndulo, L la longitud de la cuerda, g la aceleración de la gravedad, y θ el ángulo o desplazamiento.

Considerando ángulos pequeños, $\sin(\theta) \approx \theta$, y tenemos la misma ecuación de antes sustituyendo x por θ y $\omega = \sqrt{\frac{g}{L}}$.

En este caso $x(0) = \theta$ y $\dot{x} = 0$ si θ es el ángulo inicial para $t = 0$.

Para angulos pequeños, el sistema se considera lineal porque el angulo es similar al seno del angulo: (sistema lineal ideal)

$$m \cdot L \cdot \ddot{\theta} + m \cdot g \cdot \theta = 0$$

5.4 Solución numérica

No hemos visto métodos numéricos para resolver derivadas segundas, y tales métodos son una opción en este caso, pero podemos adaptarlo con métodos de resolución de derivadas primeras.

La ecuación de segundo orden podemos expresarla como dos ecuaciones diferenciales de primer orden para facilitar la resolución.

Tenemos las variables de estado:

- Elongación: u
- Velocidad: v

Las ecuaciones de estado son las derivadas de las variables de estado:

- $\dot{u} = v$
- $\dot{v} = \ddot{u} \rightarrow$ despejamos directamente desde la ecuación de 2do orden.

De la ecuación de orden 2:

$$\ddot{u} + \omega^2.u = 0$$

Se obtienen estas dos ecuaciones de orden 1:

$$\dot{u} = v$$

$$\dot{v} = \ddot{u} = -\omega^2.u$$

Imaginando...

$$\ddot{u} + A.\ddot{u} + B.\dot{u} + C.u = 0$$

Las ecuaciones quedarían:

$$\dot{u} = v$$

$$\dot{v} = \ddot{u} = a$$

$$\dot{a} = \ddot{v} = \ddot{u}$$

$$\ddot{u} = -A.a - B.v - C.u$$

Ahora podemos aplicar el método de Euler a estas ecuaciones para expresarlas de manera discreta.

Considerando a $n = 1, 2, \dots, N$, este método ofrece una aproximación de las derivadas primeras.

$$\frac{u_{n+1} - u_n}{\Delta t} = v_n$$

$$\frac{v_{n+1} - v_n}{\Delta t} = -\omega^2.u_n$$

Reordenando:

$$u_{n+1} = u_n + \Delta t \cdot v_n$$

$$v_{n+1} = v_n - \Delta t \cdot \omega^2 \cdot u_n$$

Esto puede programarse en python de una manera similar a esta:

(ver mra.py)

Dado que ya conocemos la solución exacta $u(t) = X_0 \cdot \cos(\omega t)$, tenemos que ver cómo generar una aproximación apropiada para el intervalo de tiempo $[0, T]$. Esta solución tiene un período $P = \frac{2\pi}{\omega} = 1/f$. Simulando tres períodos de la función coseno, o sea, $T = 3P$, y eligiendo un Δt de modo que haya 20 intervalos por período, es decir, $\Delta t = P/20$, y un total de $N_t = T/\Delta t$ intervalos.

El código genera la siguiente gráfica:

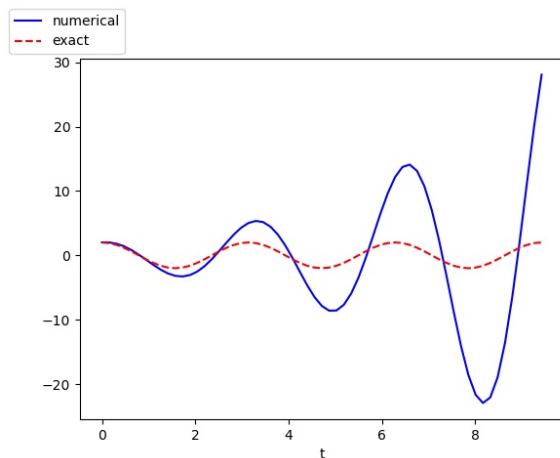


Figura 46: Sistema M-R-A ideal - Elongación

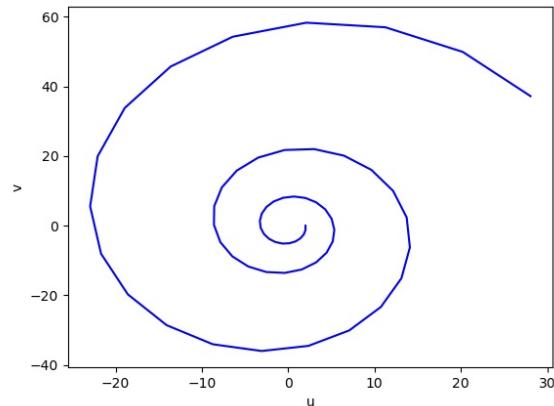


Figura 47: Sistema M-R-A ideal - Fase

Esta figura compara entre la solución numérica y la exacta. Para nuestra sorpresa, la solución numérica parece mal comparada con la otra. ¿A qué se debe este error? ¿Problemas con el código o con el método de Euler?

Ahora, analicemos cómo se comporta el modelo a variaciones de Δt . Veamos la gráfica que genera el sistema para diferentes valores de Δt

Con un $\Delta t = P/40$ vemos que se aproxima más la curva de Euler a la gráfica exacta.

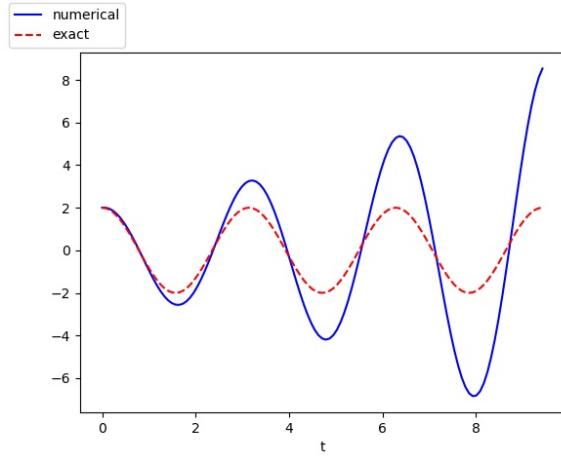


Figura 48: $\Delta t = P/40$

Veamos qué ocurre con valores de $P/160$ y $P/2000$ para Δt :

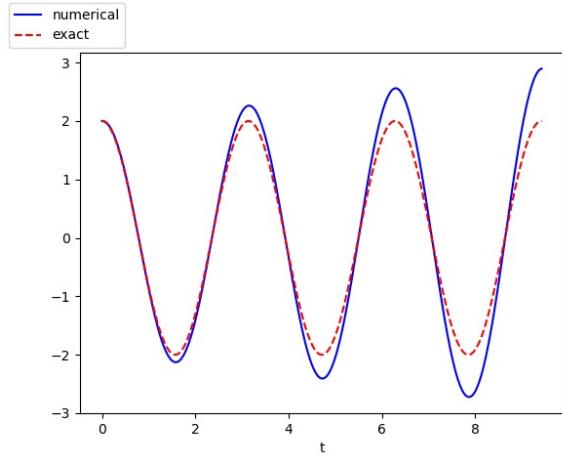


Figura 49: $\Delta t = P/160$

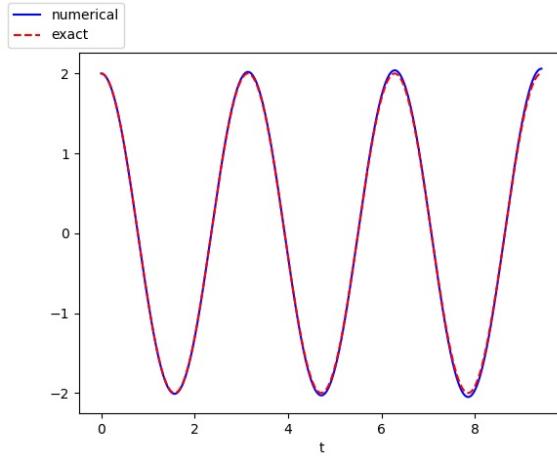


Figura 50: $\Delta t = P/2000$

Mas o menos con 2000 intervalos por período de oscilación parece suficientemente preciso para acercarnos al modelo matemático. Este modelo igual tiene algunas discrepancias, por ejemplo si simulamos 20 periodos en vez de 3, veremos que tenemos un pequeño incremento en la amplitud de la función senoidal.

Como conclusión de esto podemos decir que el método de Euler tiene como punto flojo ese pequeño crecimiento en la amplitud, pero que puede minimizarse disminuyendo el tamaño del Δt para obtener resultados satisfactorios.

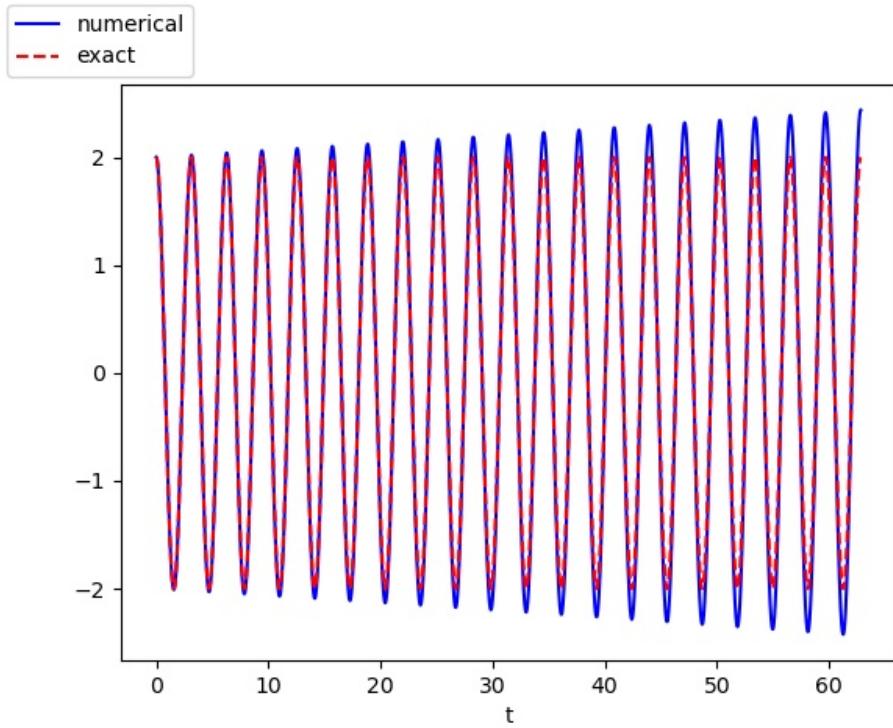


Figura 51: $\Delta t = P/2000$ con 20 períodos

5.5 Una solución mágica: Euler-Cromer

Volvamos a las ecuaciones anteriores (A):

$$u_{n+1} = u_n + \Delta t \cdot v_n$$

$$v_{n+1} = v_n - \Delta t \omega^2 u_n$$

Podemos reemplazar el un de la segunda ecuación por u_{n+1} . Las ecuaciones ahora quedan (B):

$$u_{n+1} = u_n + \Delta t \cdot v_n$$

$$v_{n+1} = v_n - \Delta t \omega^2 u_{n+1}$$

Veamos cómo se comporta el ejemplo de código inicial $\Delta t = P/20$ y para 3 períodos:

Bien no? Veamos ahora para 40 períodos:

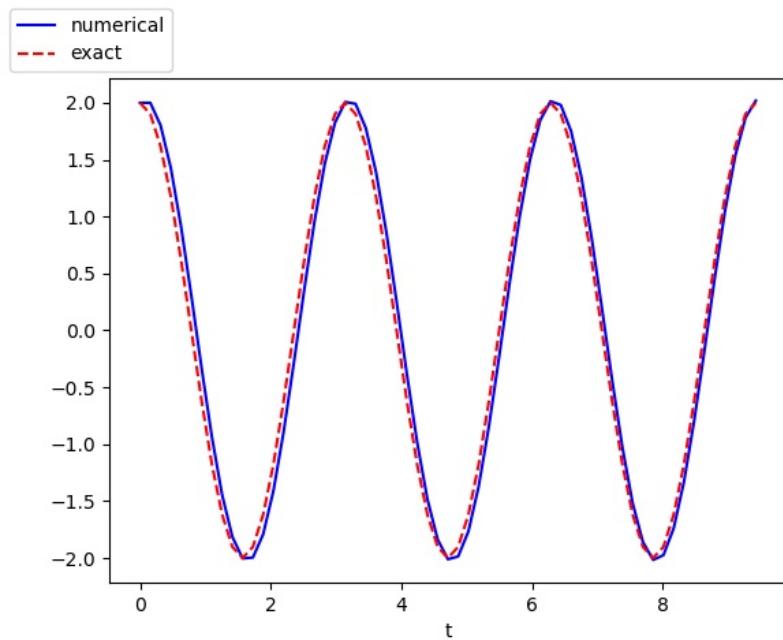


Figura 52: Aproximación Euler-Cromer

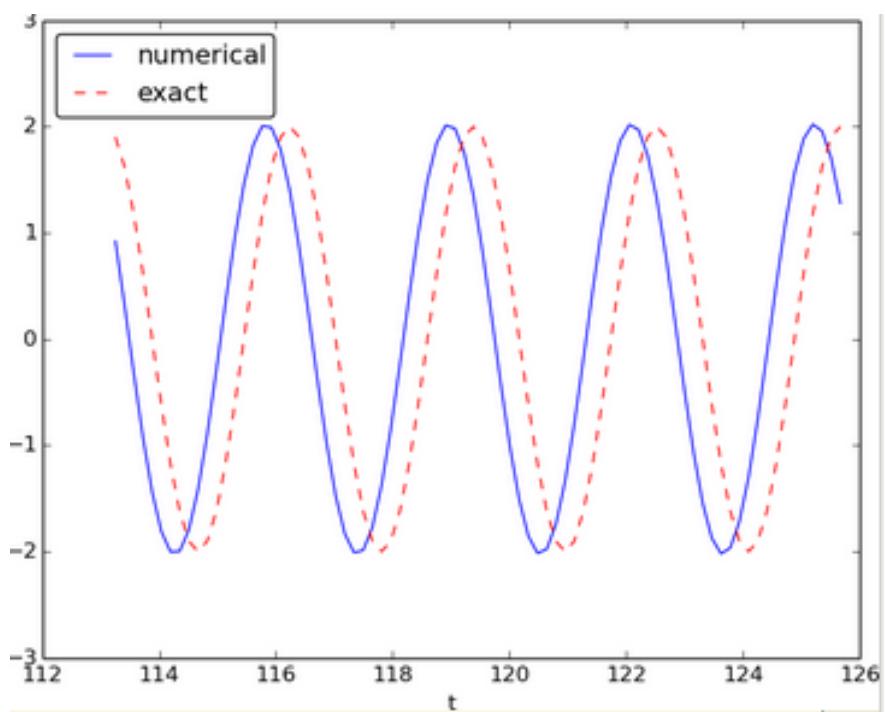


Figura 53: Aproximación Euler-Cromer - 40 periodos

Hemos eliminado la amplitud de la versión aproximada, a costa de un pequeño cambio de fase, que en simulaciones grandes puede ser un problema, pero para pocos períodos no.

Analicemos la justificación matemática. Las ecuaciones eran (transcripción):

$$u_{n+1} = u_n + \Delta t \cdot v_n$$

$$v_{n+1} = v_n - \Delta t \omega^2 u_{n+1}$$

Reordenemos:

$$\frac{u_{n+1} - u_n}{\Delta t} = v_n$$

$$\frac{v_{n+1} - v_n}{\Delta t} = -\omega^2 \cdot u_{n+1}$$

La primera de ellas se interpreta como una ecuación diferencial discretizada muestrada en t_n porque tenemos un v_n en la parte derecha de la ecuación. La parte izquierda de la ecuación se la denomina diferencia hacia adelante de la aproximación de Euler para la derivada \dot{u} .

Las diferencias hacia adelante pueden verse en esta figura:

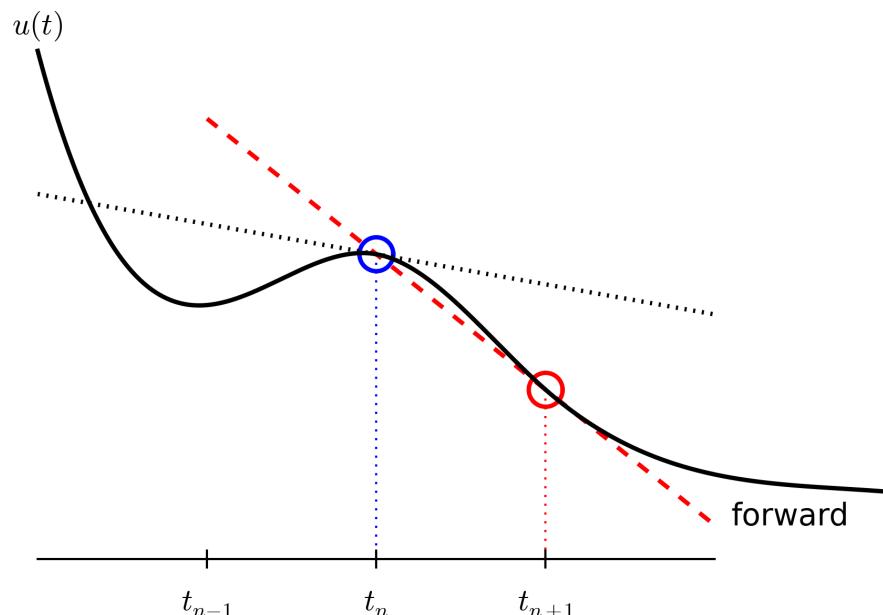


Figura 54: Euler forward difference

Por otro lado, la segunda ecuación se puede interpretar como la ecuación diferencial discretizada muestreada en t_{n+1} ya que en la parte derecha tenemos un t_{n+1} porque tenemos un u_{n+1} en la parte derecha. En este caso, la aproximación de diferencias de la parte izquierda se denomina diferencia hacia atrás.

$$\dot{v}(t_{n+1}) \approx \frac{v_{n+1} - v_n}{\Delta t}$$

ó

$$\dot{v}(t_n) \approx \frac{v_n - v_{n-1}}{\Delta t}$$

Las diferencias hacia atrás pueden verse en esta figura:

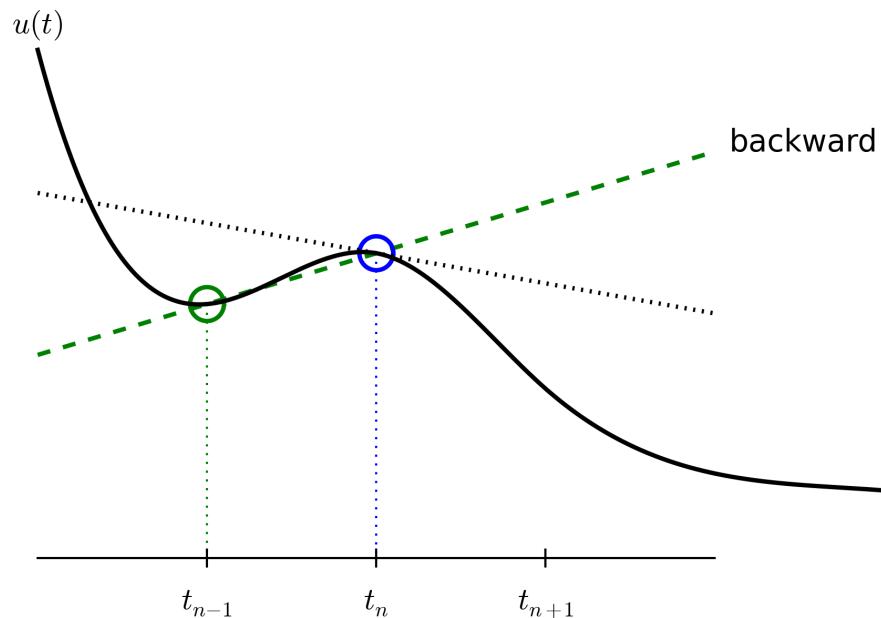


Figura 55: Euler backward difference

Para resumir, usando diferencias hacia adelante para la primer ecuación y diferencias hacia atrás para la segunda resultan en un método mucho mejor de aproximación que solamente usar diferencia hacia adelante en ambas.

La forma estándar de expresar esto es mediante ecuaciones de estado:

$$\dot{v} = \ddot{u} = -\omega^2 u$$

$$\dot{u} = v$$

Y aplicando diferencia hacia adelante en una y hacia atrás en la otra (C) 

$$v_{n+1} = v_n - \Delta t \cdot \omega^2 \cdot u_n$$

$$u_{n+1} = u_n + \Delta t \cdot v_{n+1}$$

Esto es, la velocidad v se actualiza y con ella se calcula la posición u teniendo en cuenta la última velocidad calculada. Si nos fijamos, en las ecuaciones originales B teníamos subíndice $n + 1$ en la ecuación de u , mientras que en estas el subíndice $n + 1$ está en la ecuación de v . A nivel de precisión en los cálculos, no hay diferencias entre las anteriores y estas nuevas ecuaciones, de modo que el orden de las ecuaciones diferenciales originales no importa.

Estas nuevas ecuaciones se conocen como **Euler-Cromer**, o ecuaciones semi-implícitas de Euler.

Podemos implementar las ecuaciones B en Python de esta manera:

```
u = zeros(N_t+1)
v = zeros(N_t+1)
# Initial condition
u[0] = 2
v[0] = 0
# Step equations forward in time
for n in range(N_t):
    v[n+1] = v[n] - dt*omega**2*u[n]
    u[n+1] = u[n] + dt*v[n+1]
```

5.6 Agregando amortiguación y fuerza externa

El modelo $\ddot{u} + \omega^2 \cdot u = 0$ es la forma más simple de un oscilador representado matemáticamente. No obstante, las aplicaciones de la vida real involucran otros efectos físicos que pueden incorporarse a nuestro modelo agregando más términos en nuestra ecuación diferencial.

Típicamente podemos agregar una fuerza de amortiguación y una fuerza del resorte. Ambas fuerzas pueden ser magnitudes no lineales de los argumentos de desplazamiento y velocidad.

Además, la fuerza de entorno $F(t)$ puede actuar en el sistema, por ejemplo, el clásico péndulo tiene una fuerza de restauración o fuerza del resorte $s(u) \approx \sin(u)$, y la resistencia al aire puede actuar como fuerza de amortiguación $f(\dot{u})$. Ejemplos de las fuerzas ambientales incluyen temblores de la tierra como olas, y la fuerza del viento.

Con estas fuerzas en el sistema, F , f y s , la suma de las fuerzas puede escribirse como:

$$F(t) - f(\dot{u}) - s(u)$$

Los signos negativos indican que las fuerzas de amortiguación y del resorte tienden a reducir la fuerza natural del sistema, son fuerzas contra el movimiento del sistema, lo atenúan.

Por ejemplo, un resorte fijado a una masa m que descansa sobre rodillos (sin rozamiento) puede ser fijada también a un amortiguador cuya fuerza será $f(\dot{u})$, como indica la Fig. 56:

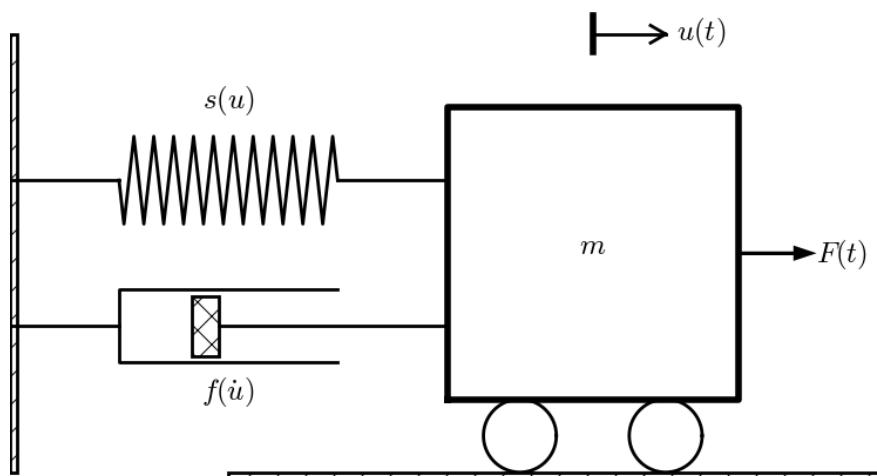


Figura 56: Sistema M-R-A con amortiguación

Podemos considerar también las siguientes asociaciones. Dado que la fuerza de amortiguación es $f(\dot{u})$, la constante amortiguación será b , y la fuerza de amortiguación puede escribirse ahora como:

$$f(\dot{u}) = b \cdot \dot{u}$$

Del mismo modo, la fuerza del resorte es, con k como constante elástica del resorte:

$$s(u) = k \cdot u$$

Esta ecuación no solamente es válida para el modelo m-r-a, sino que también lo es para cualquier oscilador, como por ejemplo, el péndulo. (completar) !!

La segunda ley de Newton para este sistema puede ser escrita así: $F = m \cdot a$

$$m \cdot \ddot{u} = F(t) - f(\dot{u}) - s(u)$$

Reordenando:

$$m \cdot \ddot{u} + f(\dot{u}) + s(u) = F(t)$$

ó

$$m \cdot \ddot{u} + b \cdot \dot{u} + k \cdot u = F(t)$$

Como es una ecuación de segundo orden, requiere dos condiciones iniciales:

$$u(0) = U_0 \wedge \dot{u}(0) = V_0$$

Las ecuaciones de estado son en este caso (D):

$$\dot{v} = \ddot{u} = \frac{1}{m} (F(t) - f(\dot{u}) - s(u))$$

$$\dot{u} = v$$

Una solución atractiva en este caso es volver a aplicar el esquema Euler-Cromer, es preciso y eficiente. Como en el caso anterior, podemos tomar las diferencias hacia adelante en un caso, y las diferencias hacia atrás en el otro:

$$\frac{v_{n+1} - v_n}{\Delta t} = \frac{1}{m}(F(t_n) - s(u_n) - f(v_n))$$

$$\frac{u_{n+1} - u_n}{\Delta t} = v_{n+1}$$

Que pueden reordenarse de esta forma:

$$v_{n+1} = v_n + \frac{\Delta t}{m}(F(t_n) - s(u_n) - f(v_n))$$

$$u_{n+1} = u_n + \Delta t \cdot v_{n+1}$$

Podemos definir ahora una función python que implemente este método:
(ver función EulerComer en mra1.py)

5.7 Amortiguación lineal

Consideremos el sistema amortiguado con $s(u) = k \cdot x$ con k como constante elástica del resorte, y una fuerza de amortiguación $f(\dot{u})$ proporcional a $\dot{u} = v$, $b \cdot \dot{u}$, con algún $b > 0$ (constante de amortiguación).

Elijamos algunos valores comunes de estos parámetros para ilustrar la amortiguación:

$$m = 1; k = 1; b = 0,3; U_0 = 1; V_0 = 0$$

El siguiente código ilustra el caso: (ver función linear_dumping en mra1.py)

Se utilizan las funciones lambda de python como una forma más sencilla de escribir funciones matemáticas en vez de utilizar las funciones tradicionales de python (def...).

La función plot_elong() permite graficar el $u(t)$, o una parte del mismo. En este caso generó este diagrama de elongación (Fig. 57) y fase (Fig. 58):

Vamos a extender ahora el ejemplo anterior para incorporarle una fuerza externa de oscilación, por ejemplo, $F(t) = A \cdot \sin(\omega t)$:

$$m \cdot \ddot{u} + b \cdot \dot{u} + k \cdot u = A \cdot \sin(\omega t)$$

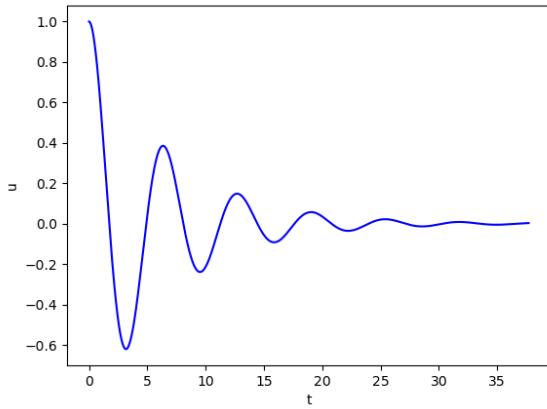


Figura 57: Elongación

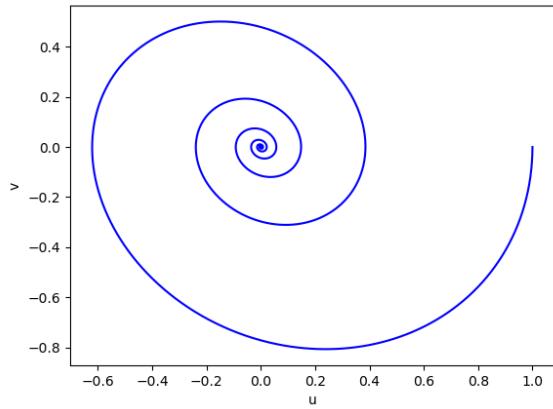


Figura 58: Fase

Supongamos valores: $A = 0,5$ y $\omega = 3$

```
from math import pi, sin
w = 3
A = 0.5
F = lambda t: A*sin(w*t)
```

Esto generará la siguiente figura (elongación Fig. 59 y fase 60).

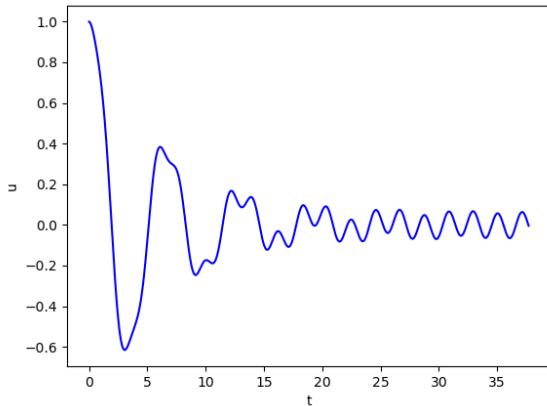


Figura 59: Elongación (forzado)

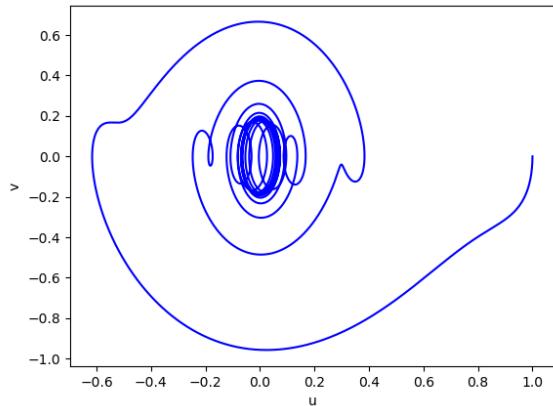


Figura 60: Fase (forzado)

La diferencia con la figura anterior es que las oscilaciones comienzan igual que antes, sin mucha influencia de la fuerza externa, pero las el sistema normal sin oscilación termina y la fuerza externa, $0,5 \sin(3t)$ induce oscilaciones con un periodo más corto, $2\pi/3$.

Veamos qué pasa con una fuerza externa de mayor amplitud... supongamos $A = 1,5$. La Fig. 61 detalla el diagrama de elongación, mientras que la Fig. 62 el de fase:

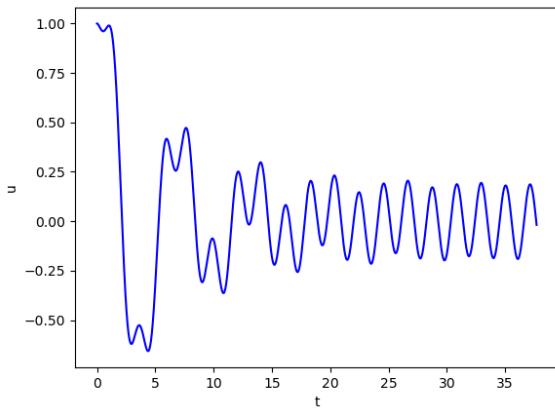


Figura 61: Elongación (forzado)

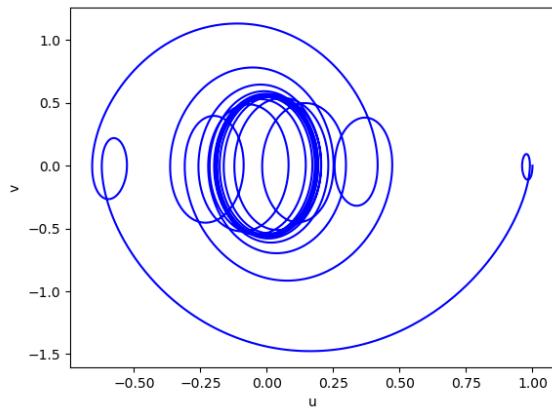


Figura 62: Fase (forzado)

Se puede ver que ahora la fuerza externa tiene mayor influencia en la función original, aún en las primeras oscilaciones.

Aquí se puede seguir probando con diferentes valores, o funciones seno y coseno para la fuerza externa. En los libros de física podemos encontrar la resolución exacta de estas ecuaciones diferenciales, y no las aproximadas que calculamos mediante un modelo simulado.

Un caso particular se da cuando la frecuencia de oscilación natural del sistema amortiguado es igual que la frecuencia de la fuerza externa. Por ejemplo, $F(t) = A \cdot \sin(t)$ con $A = 0,5$, pero una constante de amortiguación menor, $b = 0,1$. Este modelo genera el siguiente diagrama (corrido para 24π): Figuras 63 y 64.

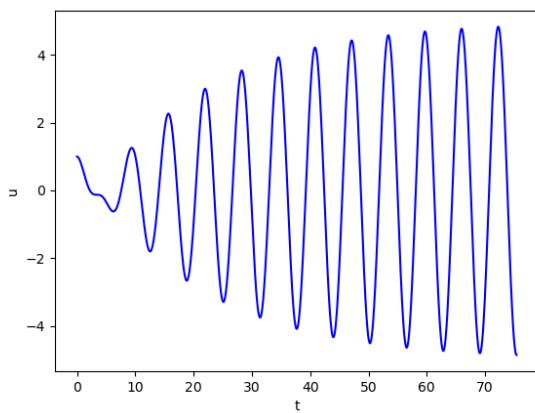


Figura 63: Elongación (forzado)

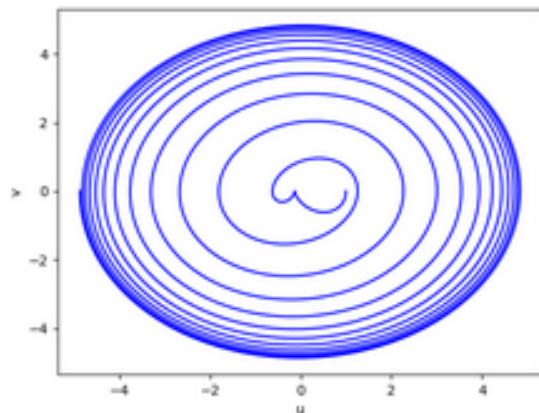


Figura 64: Fase (forzado)

La mayor diferencia es que la amplitud va creciendo significativamente en la medida en que vamos aumentando periodos de oscilación. Este fenómeno se denomina **resonancia**. Reduciendo la amortiguación tenemos un sistema que en resonancia va aumentando su amplitud de manera lineal con el tiempo.

5.8 Usando fricción como amortiguación

Un cuerpo con masa m fijado a un resorte con constante elástica k que se desliza sobre una superficie podría graficarse de la siguiente manera:

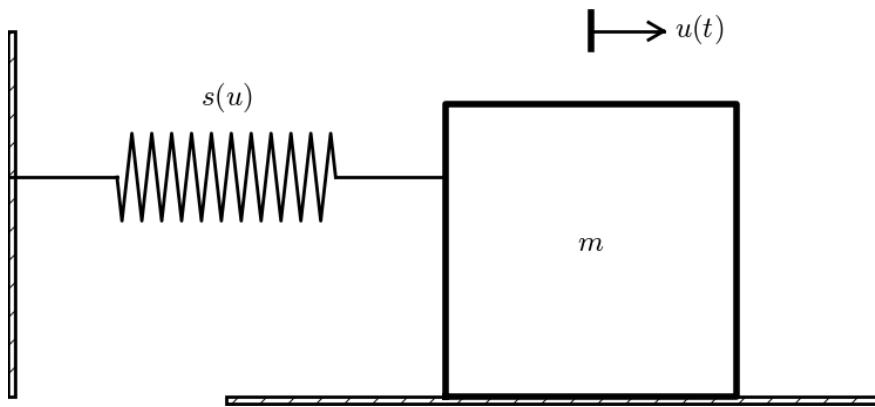


Figura 65: Sistema M-R-A con fricción

El cuerpo con masa m está sujeto a una fuerza de fricción $f(\dot{u})$ que frena su recorrido sobre el plano. La fuerza de fricción $f(\dot{u})$ en este caso quedaría modelada con la **fricción de Coulomb**:

$$f(\dot{u}) = \begin{cases} -\mu mg, & \dot{u} < 0, \\ \mu mg, & \dot{u} > 0, \\ 0, & \dot{u} = 0 \end{cases}$$

Donde μ es el coeficiente de fricción, y $m.g$ es la fuerza normal en la superficie donde el cuerpo se desliza. Esta fórmula puede ser escrita utilizando la función `signo()` como sigue:

$$f(\dot{u}) = \mu \cdot m \cdot g \cdot \text{sign}(\dot{u})$$

La función $\text{sign}(x)$ es cero cuando $x = 0$, 1 cuando x es positivo, y -1 cuando x es negativo. La fuerza de amortiguación no lineal por el rozamiento se define así:

$$s(u) = -k\alpha - 1 \tanh(\alpha u)$$

Que es aproximadamente igual a $-k.u$ para valores pequeños de u (se considera linear en ese caso). Si no hay excitación externa en el sistema, el modelo matemático sería el siguiente:

$$m.\ddot{u} + f(\dot{u}) + s(u) = 0$$

$$m.\ddot{u} + \mu.m.g. \text{sign}(\dot{u}) + k\alpha - 1. \tanh(\alpha u) = 0$$

Simulemos la situación con un cuerpo de $m = 1kg$ que se desliza por una superficie de $\mu = 0,4$ mientras es atraído o repelido por un resorte cuya constante elástica es $k = 1000$.

El desplazamiento inicial del cuerpo será de $10cm$, y el parámetro α en $s(u)$ será de 60. Usando la función `EulerCromer` definida antes, podemos escribir una nueva función que tenga en cuenta la fricción de esta manera:

(ver función `sliding_friction` en `mra1.py`)

En este caso la función $s(u)$ utiliza la forma general no lineal. Esto genera las siguientes gráficas (elongación Fig. 66 y fase 67):

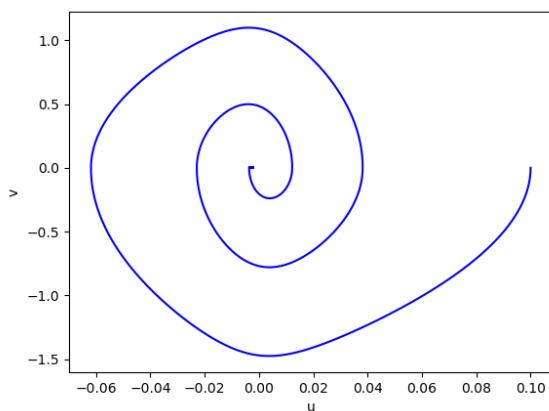


Figura 66: MRA Fricción - Elongación

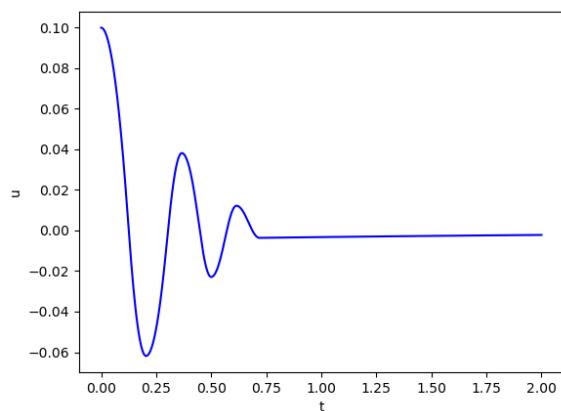


Figura 67: MRA Fricción - Fase

Si consideramos valores de u pequeños, podemos usar la función $s(u) = k.u$, lo que genera el siguiente comportamiento (elongación Fig. 68 y fase 69):

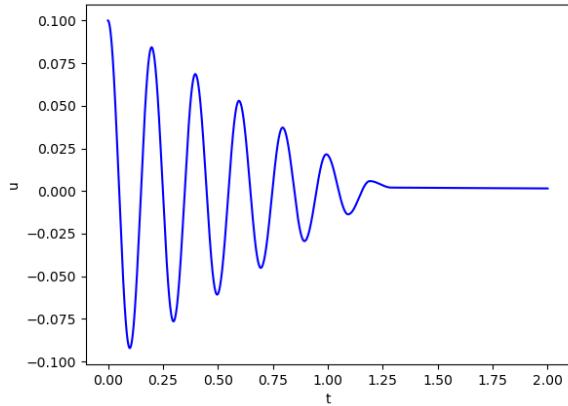


Figura 68: MRA Fricción - Elongación

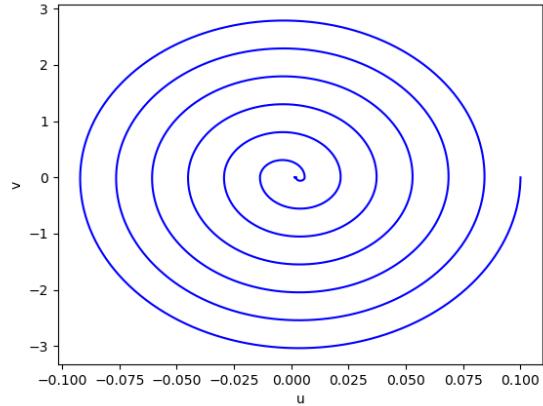


Figura 69: MRA Fricción - Fase

5.9 Oscilador de Van der Pol

El oscilador de Van der Pol intenta compensar la atenuación por amortiguación/rozamiento, pero sin recurrir a una excitación externa, sino mediante la incorporación de una "aceleración negativa" dependiente del valor del desplazamiento o elongación del oscilador.

La ecuación diferencial que lo caracteriza es la siguiente:

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = 0$$

Esta ecuación representa un sistema no lineal amortiguado, y el grado de la no-linealidad está dado por μ . Si $\mu = 0$ el sistema es lineal y no amortiguado, pero si μ se incrementa la fuerza de la no-linealidad también aumenta. Hay que tener en cuenta que μ es un valor positivo.

Consideremos la siguiente asociación: $b = -\mu.(1 - x^2)$

Si reescribimos la ecuación sustituyendo a b tendremos un sistema oscilatorio amortiguado sin fuerza externa. Dependiendo del valor de la elongación y de μ , b podrá ser:

- **Positivo:** el sistema se comportará como un oscilador amortiguado no forzado.
- **Negativo:** el sistema se comportará como un oscilador acelerado.
- **Cero:** el sistema se comportará como un oscilador lineal ideal.

Las ecuaciones de estado permiten representar al sistema de segundo orden como un sistema de ecuaciones diferenciales de primer orden de esta forma:

$$\begin{aligned}\frac{dx}{dt} &= y \\ \frac{dy}{dt} &= \mu(1 - x^2)y - x\end{aligned}$$

Una función en Python que permite resolver Van der Pol podría ser la siguiente:

```

def vdp(t, z):
    x, y = z
    return [y, mu*(1 - x**2)*y - x]

```

El código `vdp0.py` implementa esta función, y permite especificar un argumento por línea de comandos que será el valor de μ a utilizar en la generación de los valores.

Este código realiza los diagramas de secuencia y de fase para el oscilador de Van der Pol. Veamos cómo se comporta el oscilador si utilizamos un valor de $\mu = 0,01$. Elongación en la Fig. 70 y fase en la Fig. 71.

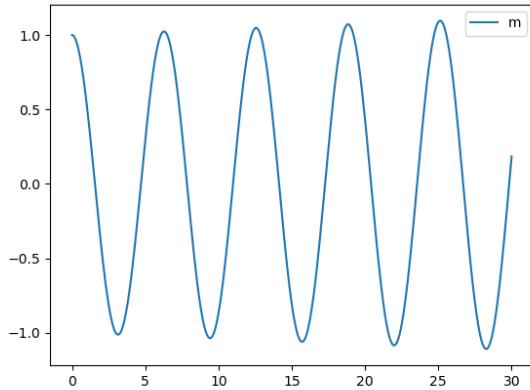


Figura 70: Van der Pol, $\mu = 0,01$ - Elongación

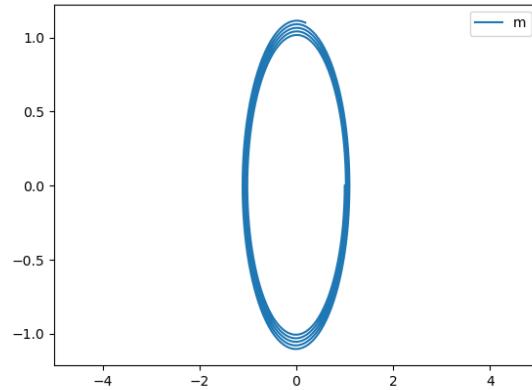


Figura 71: Van der Pol, $\mu = 0,01$ - Fase

Veamos ahora qué ocurre con un pequeño incremento en el valor de μ , ahora de 0.5: Elongación en la Fig. 72 y Fase en la Fig. 73.

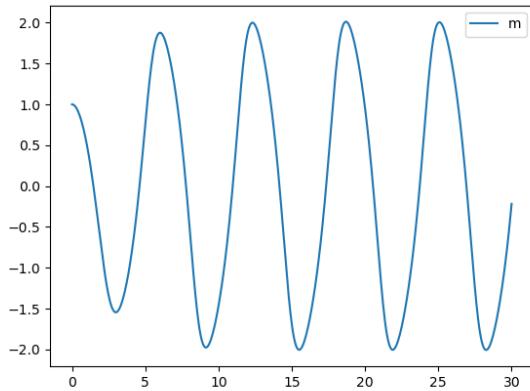


Figura 72: Van der Pol, $\mu = 0,5$ - Elongación

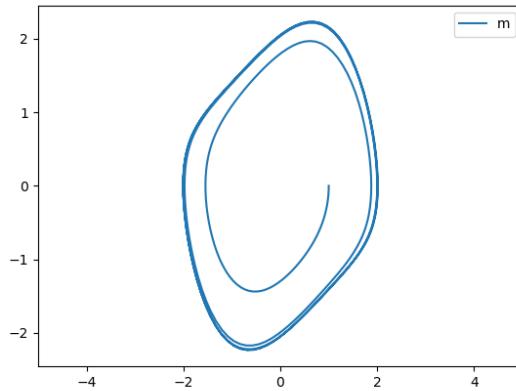


Figura 73: Van der Pol, $\mu = 0,5$ - Fase

Puede apreciarse una deformación de las gráficas. La diferencia es más notable en el diagrama de fase, donde se aprecia que la órbita se torna más romboidal, mientras que con valores más pequeños de μ la órbita se asemejaba más a un oscilador lineal ideal.

El diagrama de elongación muestra que la función senoidal se va deformando pareciéndose cada vez más a una función de diente de sierra.

Incrementemos un poco más el valor de μ : Figuras 74 y 75

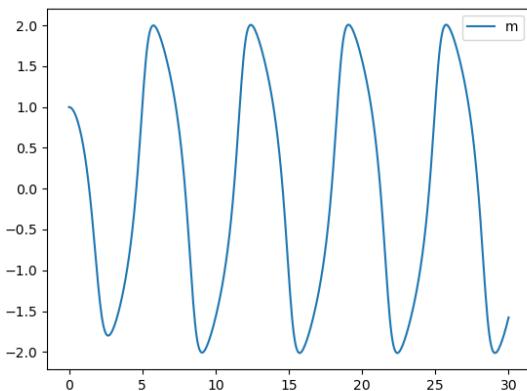


Figura 74: Van der Pol, $\mu = 1,0$ - Elongación

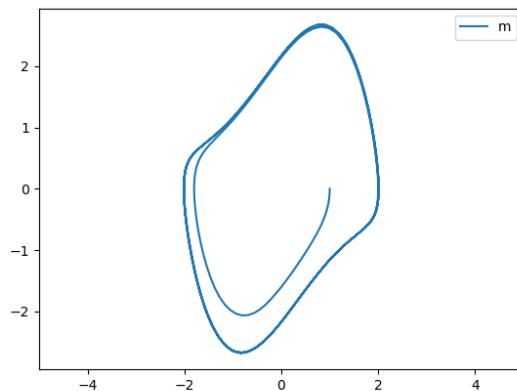


Figura 75: Van der Pol, $\mu = 1,0$ - Fase

Vayamos un poco más allá, aumentemos más el μ , Figuras 76 y 77

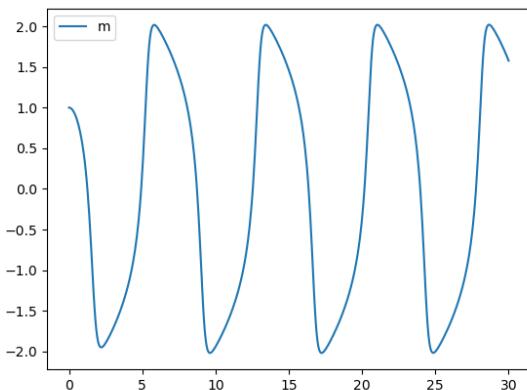


Figura 76: Van der Pol, $\mu = 2,0$ - Elongación

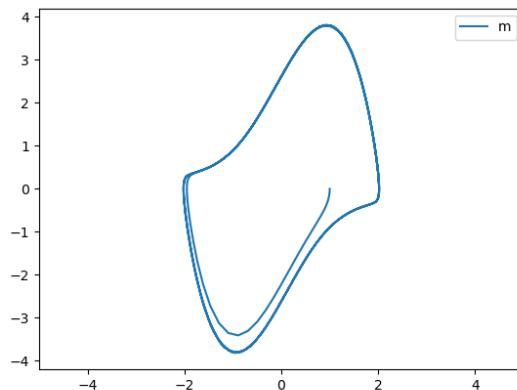


Figura 77: Van der Pol, $\mu = 2,0$ - Fase

Ahora se ve más claramente el diente de sierra en la curva de elongación, y se aprecia mejor la forma que va tomando la órbita en el diagrama de fase.

Aumentemos aún más el parámetro μ , Figuras 78 y 78.

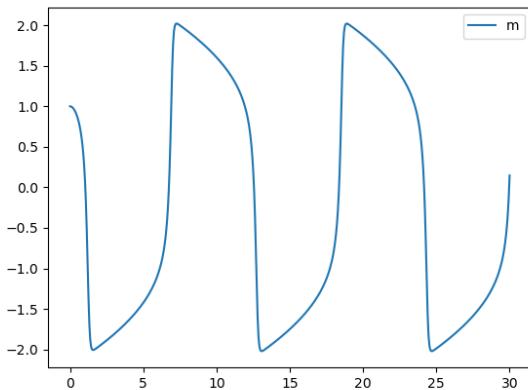


Figura 78: Van der Pol, $\mu = 5,0$ - Elongación

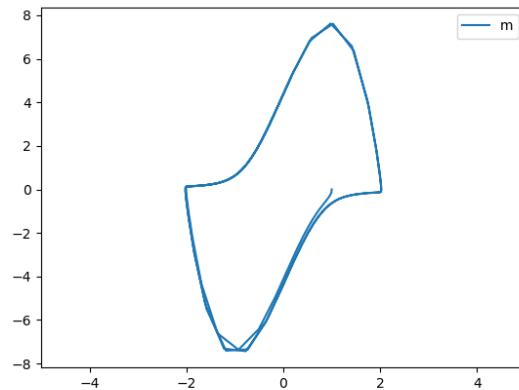


Figura 79: Van der Pol, $\mu = 5,0$ - Fase

A partir de este punto el oscilador tiende a mantener la forma de la curva aumentando las velocidades en los extremos (diagrama de fase) y aumentando el ancho de las oscilaciones en el diagrama de elongación.

El código `vdp1.py` realiza las gráficas superpuestas de para valores de μ de 0, 1, 2 y 3, y permite realizar una comparativa de la evolución de las gráficas de elongación y fase para distintos valores de este parámetro. El resultado puede verse en las Figuras 80 (Elongación) y 81 (Fase).

Como conclusión podemos decir que el oscilador de Van der Pol es un oscilador que, para valores pequeños de μ , tendientes a cero, se comporta como un oscilador lineal ideal, mientras que para valores grandes del mismo parámetro se produce una deformación en la órbita de oscilación en el diagrama de fase (también denominada órbita de relajación), y se profundiza el estilo de *diente de sierra* en el diagrama de elongación.

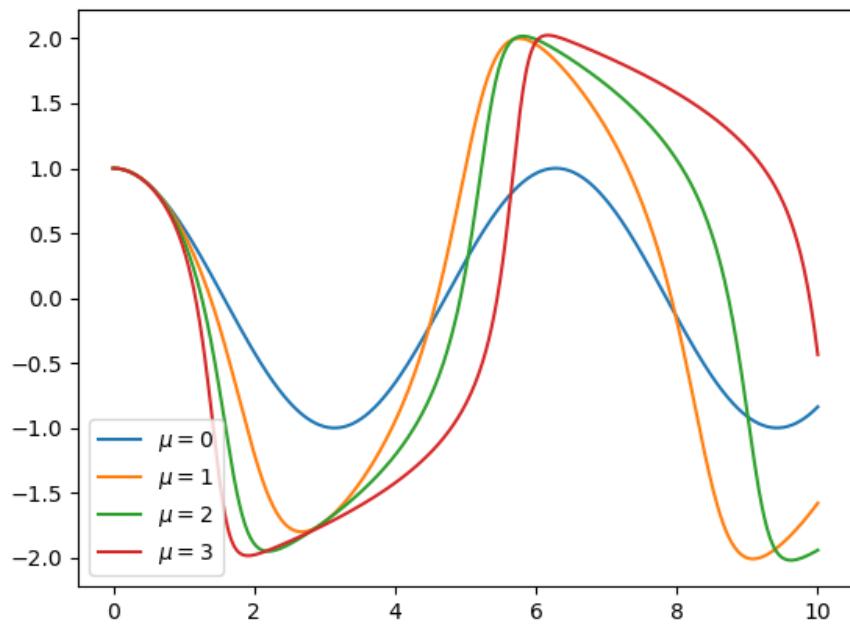


Figura 80: Van der Pol - Elongación

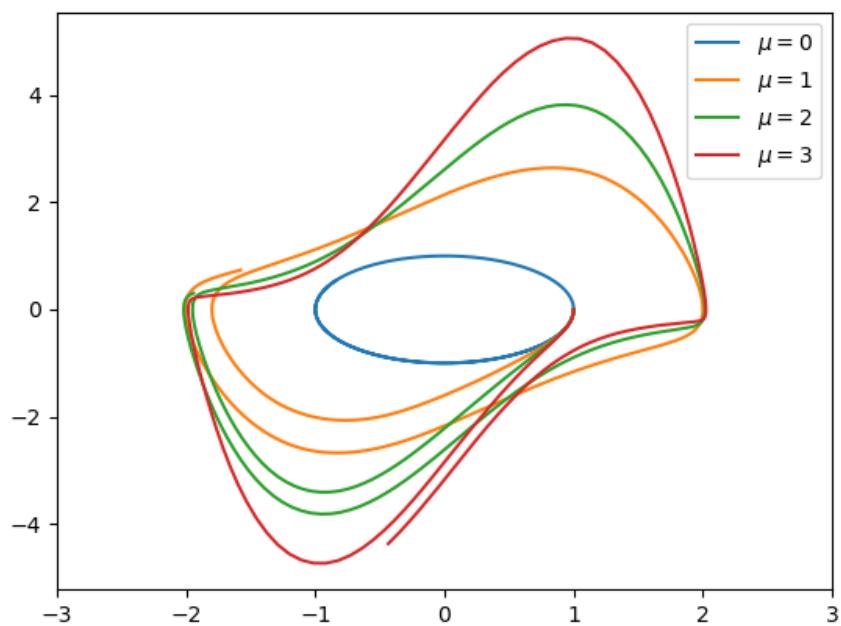


Figura 81: Van der Pol - Fase

5.10 Oscilador caótico de Duffing

Este oscilador tiene el nombre de Georg Duffing (1861-1944), quien lo definió. Provee un ejemplo de oscilador no lineal forzado periódicamente. La ecuación de Duffing es un ejemplo de un sistema dinámico que exhibe un comportamiento caótico. Para este tipo de sistemas hay frecuencias en las que las vibraciones aumentan o disminuyen repentinamente cuando se lo excita con una fuerza externa cuya frecuencia cambia lentamente.

Las frecuencias en las cuales ocurren estos saltos dependen de si la frecuencia aumenta o disminuye y si la no linealidad es mayor o menor (dependiendo de los parámetros del modelo). Entre estas frecuencias, existen múltiples soluciones para una frecuencia de excitación dada, y las condiciones iniciales determinan cuál de estas soluciones representa la respuesta del sistema.

Esta característica da origen a sistemas caóticos como este. Ya hemos visto comportamiento caótico y fractal con la ecuación logística en sistemas poblacionales, y sabemos que este comportamiento, varía dependiendo de valores en los parámetros del modelo. En la práctica los sistemas caóticos suelen estudiarse teniendo en cuenta modelos descritos por ecuaciones diferenciales parciales, tales como el flujo de fluidos agitados, no obstante esto suele ser más complejo, por lo que aquí analizaremos el comportamiento caótico por medio del oscilador de Duffing.

Para lograr el caos en este sistema, necesitamos dos componentes: una fuerza externa que agite el sistema, y una fricción o amortiguación que tienda a detenerlo.

Primero vamos a considerar la ecuación básica sin fuerza externa ni amortiguación, y luego añadiremos estos elementos adicionales.

Para analizar el caos en un sistema simple, consideraremos la ecuación básica de Duffing:

$$\ddot{x} + \delta\dot{x} + \alpha x + \beta x^3 = \gamma \cos(\omega t) \quad (5.1)$$

Donde:

- δ es la constante de amortiguación/rozamiento
- α es la constante del resorte (vamos a considerar $\alpha = 0$ en nuestro modelo simplificado)

- β controla la magnitud de la no linealidad (vamos a considerar $\beta = 1$)
- γ es la amplitud de la fuerza externa (respuesta permanente)

La ecuación desconocida $x = x(t)$ representa el desplazamiento del oscilador en el tiempo t , \dot{x} es la derivada primera con respecto al tiempo, y representa la velocidad, y \ddot{x} es la derivada segunda de x respecto del tiempo, y representa la aceleración.

La ecuación de Duffing simplificada quedaría de esta forma:

$$\ddot{x} + \delta\dot{x} + x^3 = \gamma \cos(\omega t)$$

Las variables de estado de este sistema son el desplazamiento x y la velocidad $v = \dot{x}$. Así, las ecuaciones de estado son:

$$\dot{x} = v \tag{5.2}$$

$$\dot{v} = -x^3 - \delta v + \gamma \cos(\omega t) \tag{5.3}$$

Con valores iniciales de desplazamiento y velocidad de uno: $x(0) = 1; v(0) = 1$, obtenemos el siguiente diagrama de fase de la Figura 82

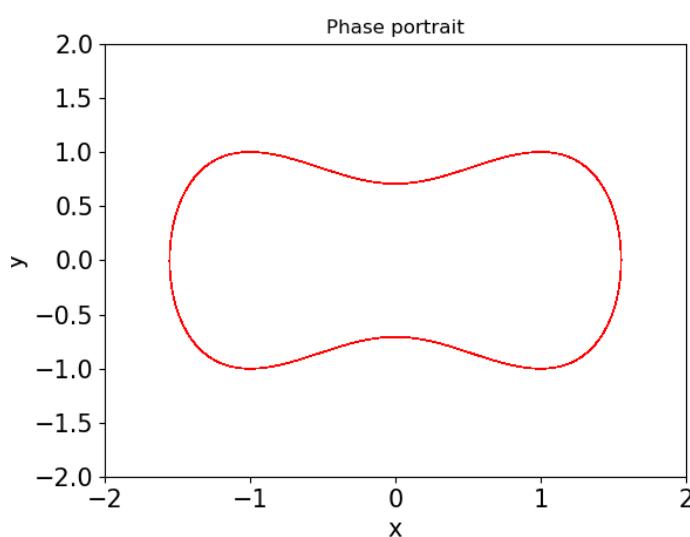


Figura 82: Diagrama de fase de F(x)

Como se observa, la órbita que genera esta función es estable, y los picos máximos de velocidad, tanto positiva como negativa, se dan en los valores de desplazamiento $x = \pm 1$.

5.10.1 Fricción y amortiguación

Como la conservación de la energía nunca puede generar caos con un solo grado de libertad como en este caso, agregaremos una fuerza externa y un coeficiente de amortiguación al sistema. Esto elimina la conservación de energía.

Las ecuaciones de estado ahora nos quedan de esta forma:

$$\begin{aligned}\frac{dx}{dt} &= v \\ \frac{dv}{dt} &= -x^3 - \delta v + \gamma \cos(\omega t)\end{aligned}$$

Donde δ es el coeficiente de fricción, y γ es la amplitud o fortaleza de la excitación externa que oscila con frecuencia ω . Veremos ahora que la transición al caos se produce ahora cuando vamos aumentando la amplitud de esta fuerza externa. Veamos algunos ejemplos.

Vamos a establecer los siguientes valores de parámetros: $\delta = 0,1$, $\gamma = 0,1$, $\omega = 1,4$.

Notemos que γ , la amplitud de la fuerza externa, es muy pequeña. Esto nos genera el siguiente diagrama de elongación: Figura 83

Como puede observarse, el sistema oscila inicialmente en un período de transición, y luego tiende a estabilizarse alrededor del punto $x = 1$ de elongación. Veamos el diagrama de fase de esta situación: Figura 84

Aquí la curva más densa representa el punto de oscilación constante y estable del sistema, mientras que el resto de la traza representa la parte transitoria del oscilador hasta que se estabiliza. Si graficáramos únicamente la parte estable, por ejemplo, para tiempos $t > 200$, el diagrama se vería aproximadamente así (Figura 85)

Aquí se observa una órbita periódica alrededor del desplazamiento $x = 1$. El período de la órbita es de $P = 2\pi/\omega$, que es el período de la fuerza de excitación externa.

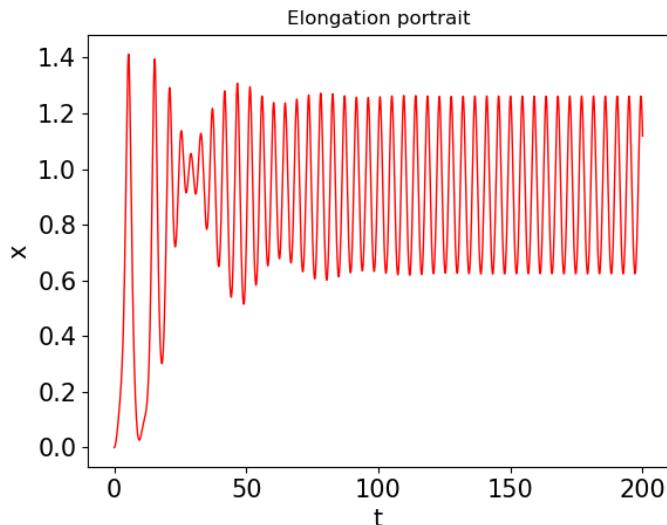


Figura 83: Duffing - Elongación

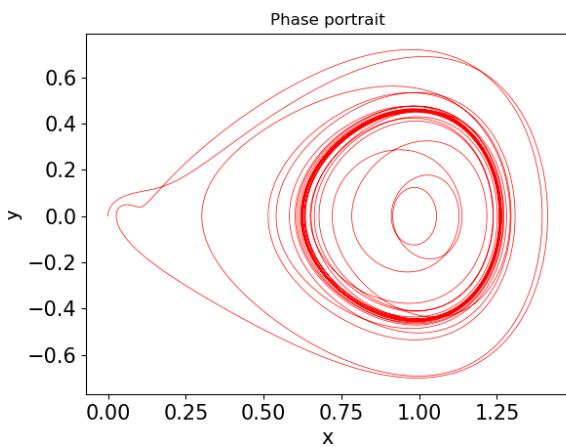


Figura 84: Duffing - Fase

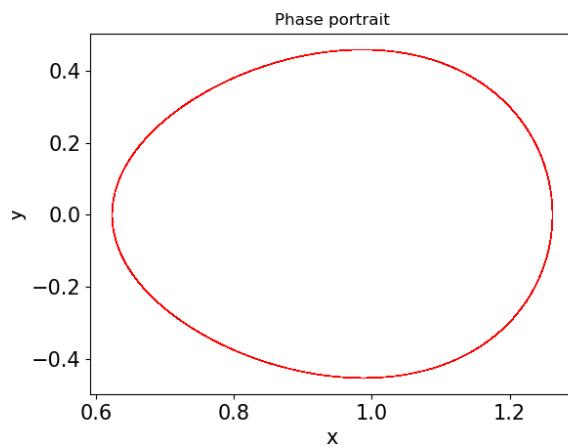


Figura 85: Duffing - Fase estable

5.10.2 Transición al Caos

Ahora investigaremos cómo el diagrama de fase cambia cuando modificamos la amplitud de la fuerza externa. Veamos qué ocurre cuando $\gamma = 0,2$ (Figuras 86 y 87)

En este caso vemos que la sección transitoria varía, pero el sistema, si bien lo hace en $x = -1$, también se estabiliza en una órbita definida. Esto puede verse si graficamos la fase, por ejemplo, para valores de $t > 800$. Figura 88

Realicemos el mismo análisis ahora para $\gamma = 0,32$. Figuras 89 y 90

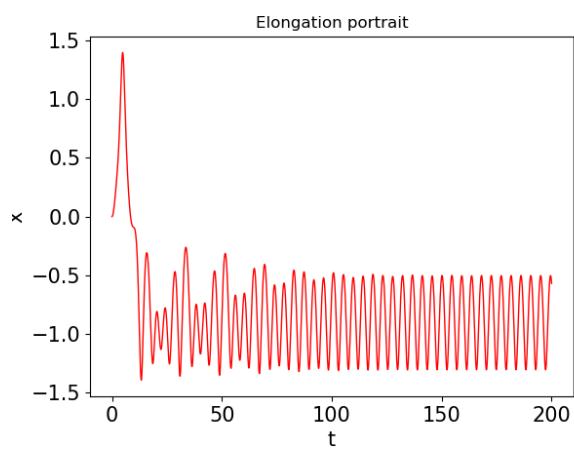


Figura 86: Duffing, $\gamma = 0,2$ - Elongación

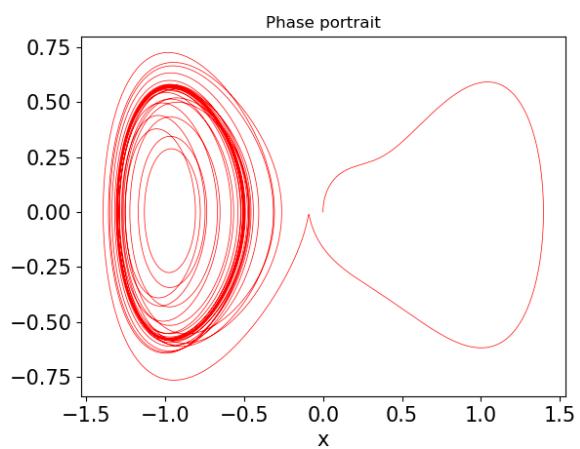


Figura 87: Duffing, $\gamma = 0,2$ - Fase

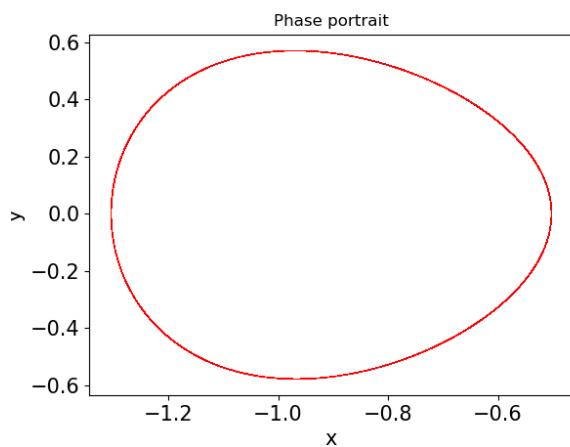


Figura 88: Duffing - Fase

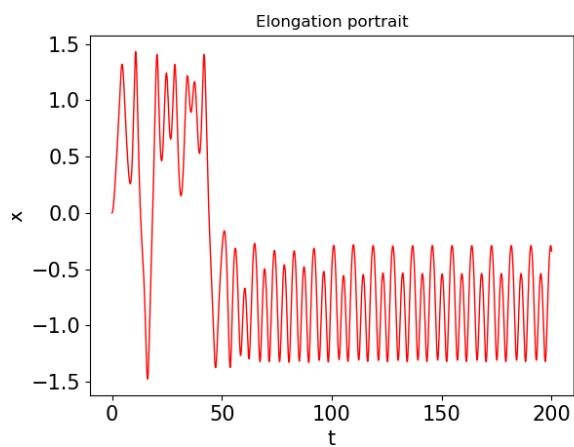


Figura 89: Duffing, $\gamma = 0,32$ - Elongación

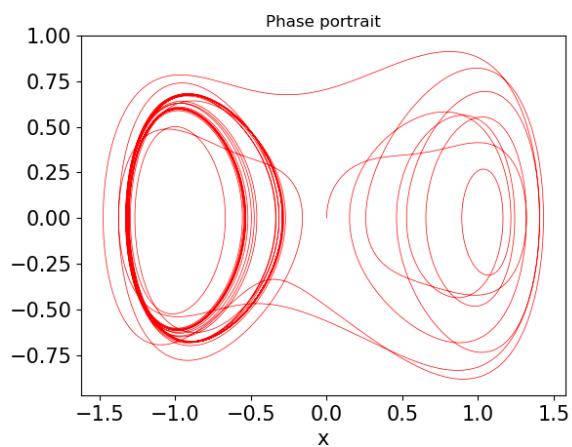


Figura 90: Duffing, $\gamma = 0,32$ - Fase

En este caso vemos que el sistema ya no se estabiliza en una sola órbita, sino que lo hace en dos, saltando de una a otra intermitentemente. Ampliemos el diagrama de fase únicamente para valores de tiempos mayores a 800. Ver Figura 91

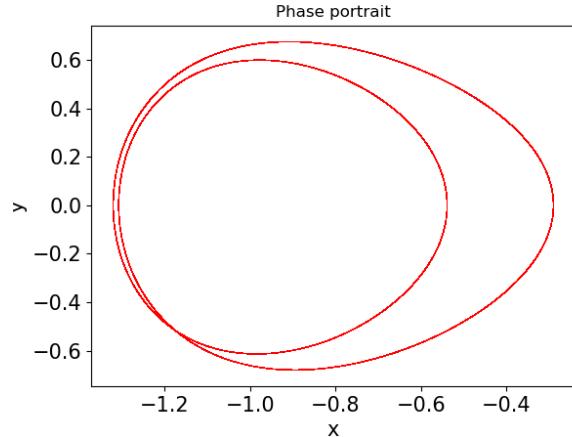


Figura 91: Duffing - Fase

Veamos ahora qué ocurre con una pequeña variación adicional en $\gamma = 0,338$. Figuras 92 y 93.

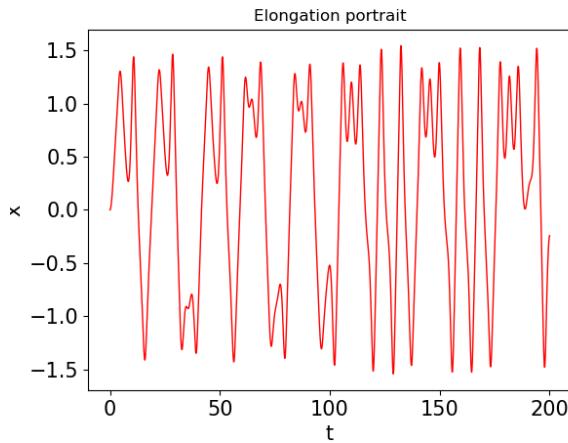


Figura 92: Duffing, $\gamma = 0,338$ - Elongación

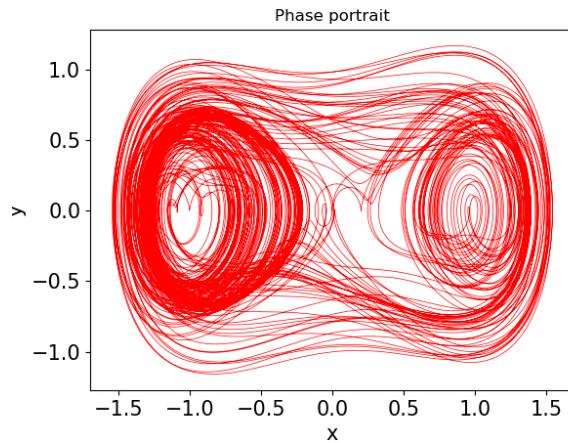


Figura 93: Duffing, $\gamma = 0,338$ - Fase

Y ampliemos la fase para tiempos mayores a 1900: Figuras 94 y 95

Hay que notar que en este caso se estabiliza en 4 órbitas, y por otro lado, lo hace para valores de tiempo mayores, es decir, el sistema demora más tiempo en obtener estabilidad.

Veamos ahora qué ocurre con un valor de $\gamma = 0,35$: (Figuras 96 y 97)

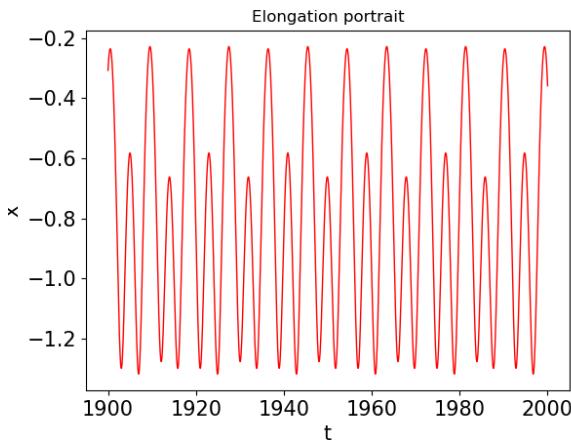


Figura 94: Duffing, $\gamma = 0,338$ - Elongación

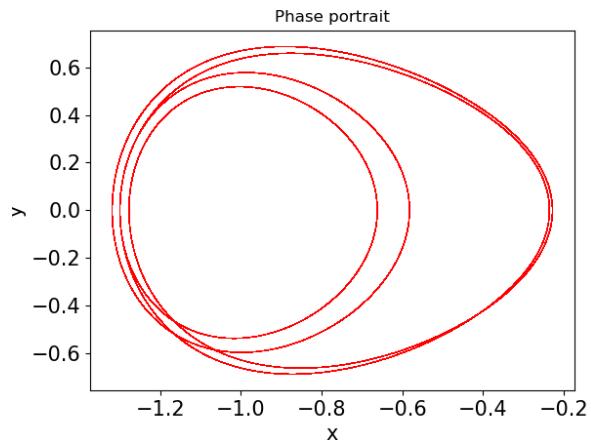


Figura 95: Duffing, $\gamma = 0,338$ - Fase

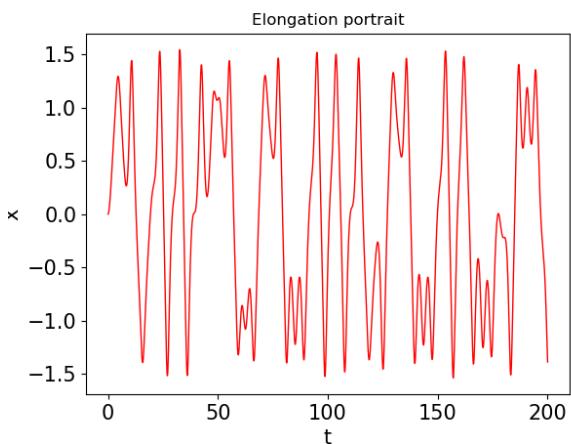


Figura 96: Duffing, $\gamma = 0,35$ - Elongación

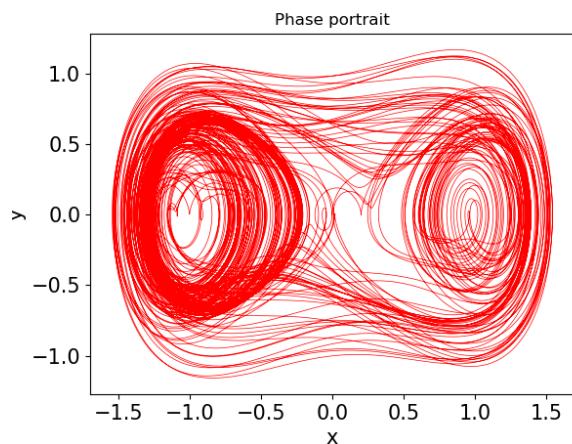


Figura 97: Duffing, $\gamma = 0,35$ - Fase

Realicemos el mismo análisis para valores de tiempo que van desde 2900 a 3000 segundos: (Figuras 98 y 99)

Aquí se ve que no se produce estabilidad en el sistema, el movimiento es caótico.

5.10.3 Pequeños cambios en la entrada

Consideremos los primeros 12 segundos de oscilación del caso anterior, con valores iniciales nulos, $x_0 = 0, v_0 = 0$ (trazo rojo) y simular nuevamente con un pequeño cambio en las entradas, $x_0 = 0,01, v_0 = 0,01$ (trazo azul) (Figura 100)

En este caso puede apreciarse que aunque inicialmente las condiciones son muy similares, el oscilador puede comportarse de manera muy diferente. Esto es una característica intrínseca de los sistemas no lineales.

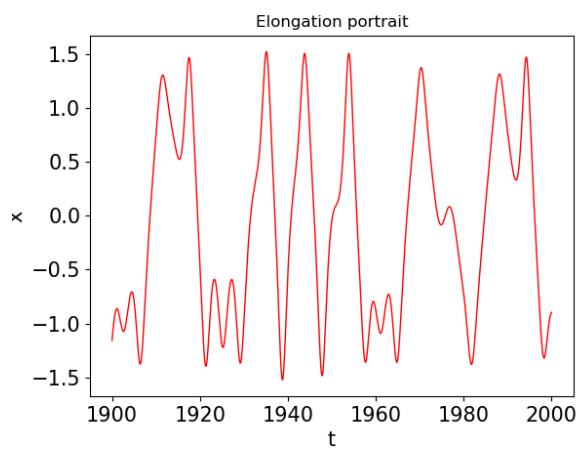


Figura 98: Duffing, $\gamma = 0,35$ - Elongación

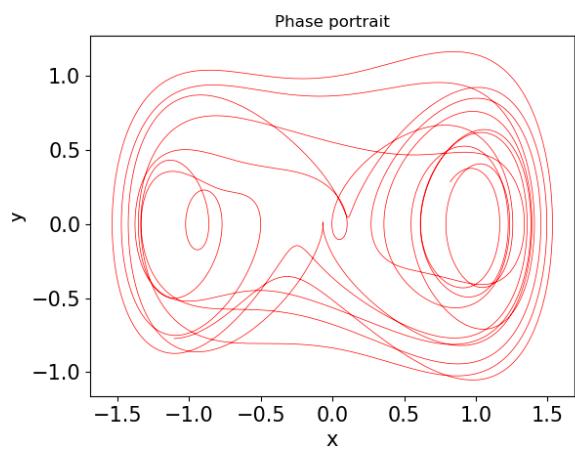


Figura 99: Duffing, $\gamma = 0,35$ - Fase

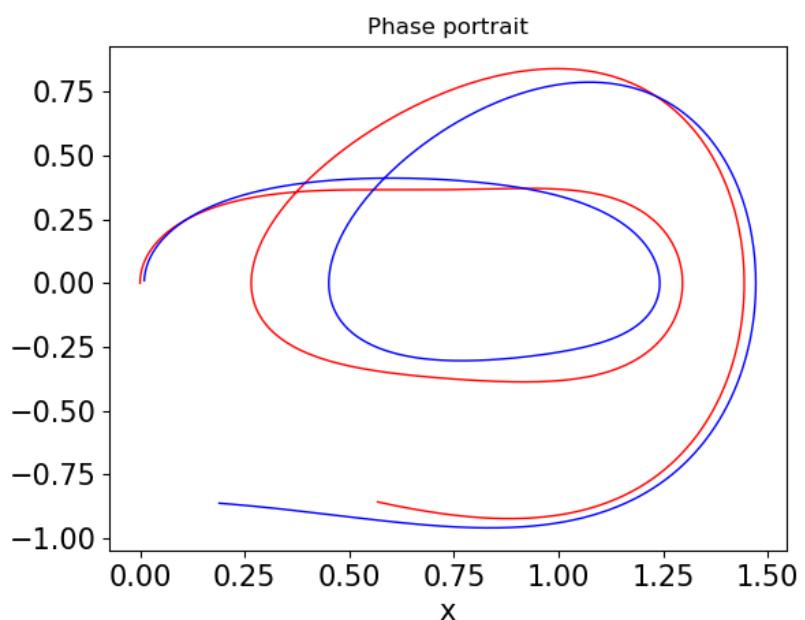


Figura 100: Duffing - Cambio condiciones iniciales

Veamos los diagramas de elongación para los primeros 40 segundos con estos mismos parámetros iniciales (Figura 101)

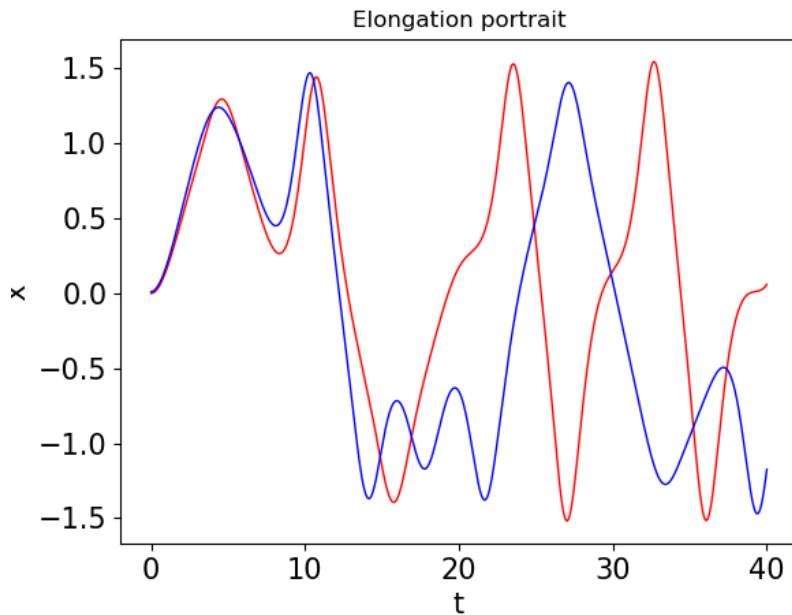


Figura 101: Duffing - Cambio condiciones iniciales

5.10.4 Secciones de Poincaré

Una manera muy útil de analizar el comportamiento caótico de un sistema es mirar las secciones de Poincaré. En este caso se considera el diagrama de fase, pero no toda la curva, toda la trayectoria, sino únicamente en algunos puntos discretos del mismo en diferentes valores de t , por ejemplo, $t = 2\pi/\omega$, $t = 3\pi/\omega$, $t = 4\pi/\omega$, etc.

El diagrama de secciones de Poincaré únicamente rescata el valor instantáneo de x y v para un t_i determinado, por lo que para cada uno de los t_i seleccionados obtendremos un punto en el diagrama de fase.

Si el sistema tuviera un período constante, realizar las secciones de Poincaré para un período igual al del oscilador redundará en un solo punto del diagrama de fase. Lo mismo, si obtenemos valores de x y v dos veces por período, obtendremos dos puntos en el diagrama.

En el caso de sistemas caóticos como el oscilador de Duffing, las secciones de Poincaré producirán formas interesantes que cambiarán respecto de pequeños cambios en las variables de entrada.

El siguiente diagrama fue generado con los datos anteriores, realizando una iteración con `limit = 12000` en el código `duffingX.py`: (Figura 102)

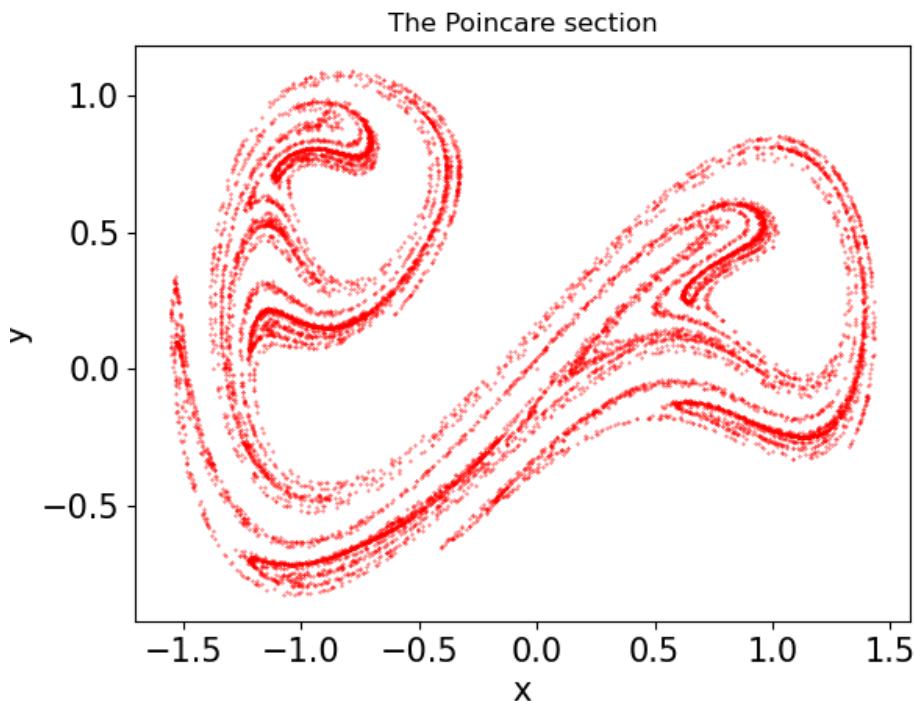


Figura 102: Duffing - Secciones de Poincaré

Este diagrama puede variar dependiendo de cambios en la frecuencia de oscilación de la fuerza externa. La figura 102 fue generada teniendo en cuenta un $\omega = 1,4$. La Figura 103 ha sido generada con un $\omega = 1,5$.

Estos diagramas, a su vez, tienen un comportamiento fractal, lo que indica que si hicieramos zoom en una sección caótica del mismo, veremos que, en menor escala, el comportamiento mantiene la misma forma.

Si realizamos un zoom al área marcada en la Figura 104, ampliando la zona, veremos lo que se aprecia en la Figura 105. Lo mismo ocurre con un zoom en el área marcada en esta figura, cuyo resultado se ve en la Figura 106.

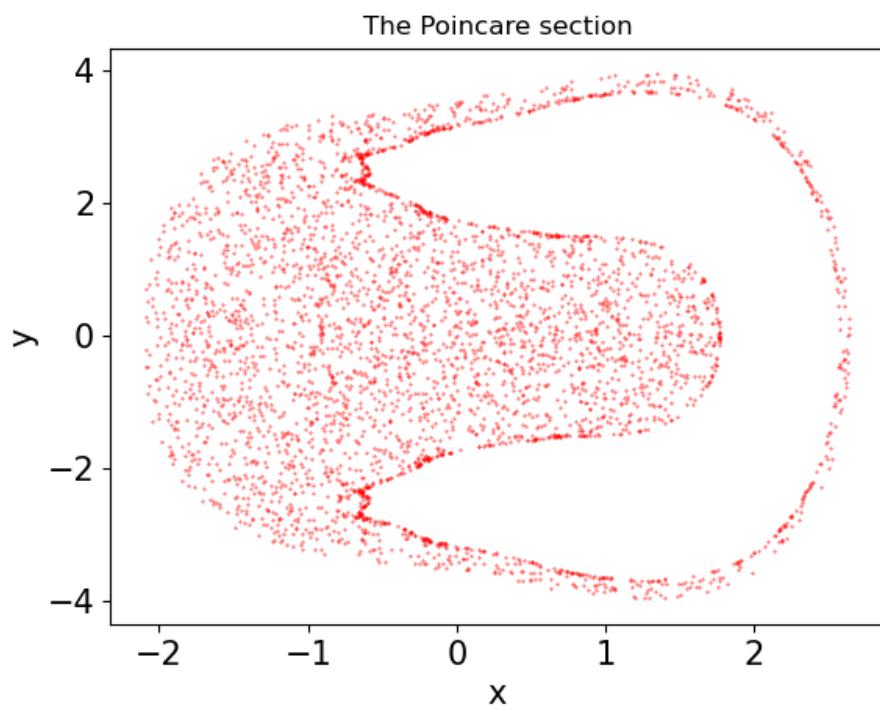


Figura 103: Duffing - Secciones de Poincaré

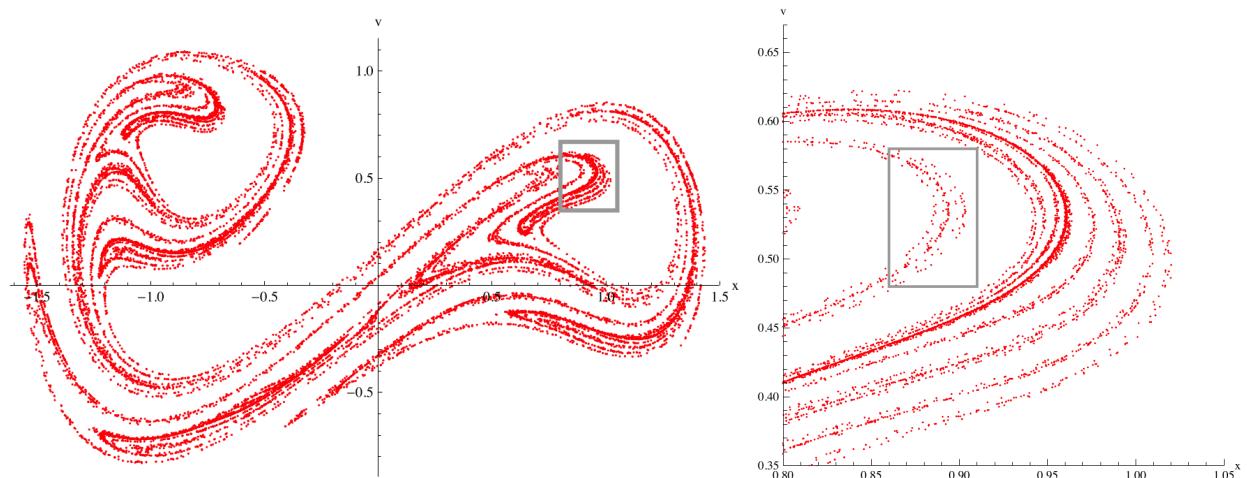


Figura 104: Duffing - Poincaré fractal

Figura 105: Duffing - Poincaré fractal

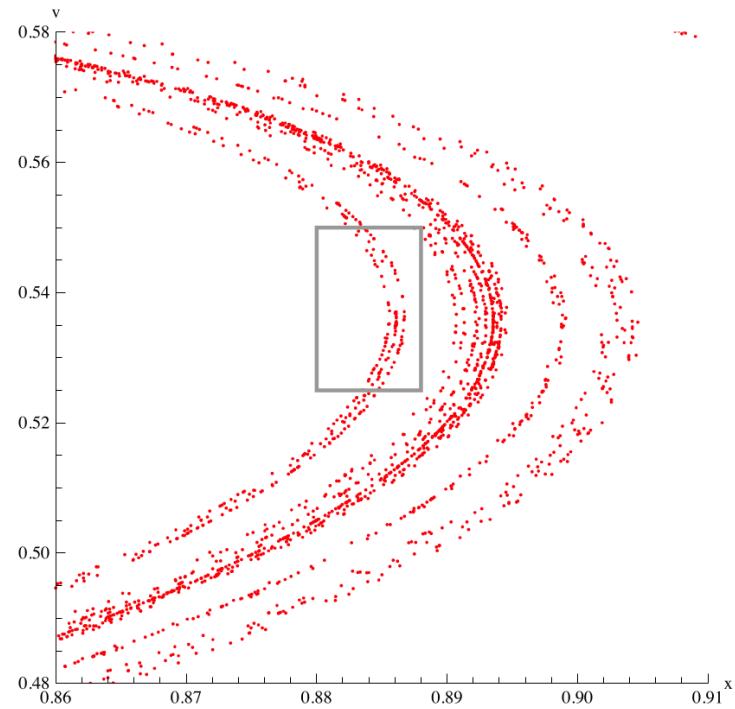


Figura 106: Duffing - Secciones de Poincaré - Zoom

5.10.5 Función potencial

La ecuación de Duffing también puede ser interpretado como el comportamiento de una partícula clásica en una función potencial.

Para ilustrar el caso, elegimos una función que tenga los valores mínimos en ± 1 y una unidad de energía tal que el valor inferior (profundidad) sea $-1/4$, separados por la barra de $x = 0$, punto en el cual $V(0) = 0$. Así, la función potencial está dada por la siguiente ecuación:

$$\dot{x} = V(x) = \frac{x^4}{4} - \frac{x^2}{2}$$

Cuya gráfica es la siguiente:

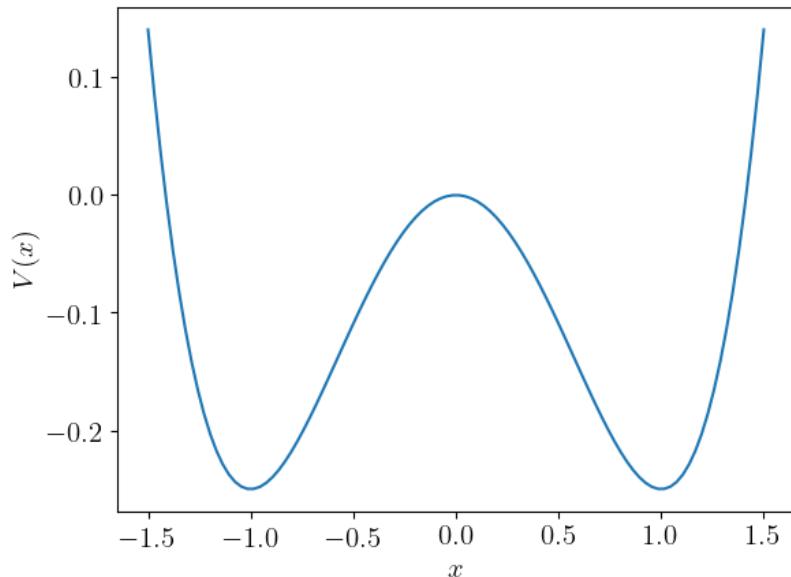


Figura 107: Función potencial

Las ecuaciones $V(x)$ y su derivada pueden programarse en python de esta manera:

```
V = lambda x: -0.5 * x**2 + (0.25 * x**4)
dVdx = lambda x: x**3 - x
```

Cuya gráfica puede generarse utilizando una grilla discreta de valores como se especifica en el código `duffing.py`.

La fuerza está dada por la derivada negativa de esta función potencial:

$$F(x) = m \ddot{x} = -\frac{dV}{dx} = x - x^3 \quad (5.4)$$

Para simplificar los cálculos, asumiremos que $m = 1\text{kg}$ en la Ecuación 5.4.

Como es usual en este tipo de sistemas, pueden representarse una ecuación diferencial en la forma de la segunda ley de Newton: $F = m.a$.

Esto nos permite generar dos ecuaciones diferenciales de primer orden:

$$\begin{aligned}\frac{dx}{dt} &= v \\ \frac{dv}{dt} &= x - x^3\end{aligned}$$

Estas dos ecuaciones ya las hemos analizado antes, Ecuación 5.2 y 5.3, son las ecuaciones de estado del oscilador de Duffing sin fuerza externa, y generan una órbita estable.

$V(x)$ puede considerarse como la velocidad, o la derivada primera de x , \dot{x} , por lo que si queremos calcular la Fuerza del oscilador necesitamos primero obtener la aceleración, por la segunda ley de Newton. Esto podemos hacerlo de esta manera:

$$\ddot{x} = -\frac{dV}{dx} - \delta \dot{x} + \gamma \cos(\omega t)$$

Sustituyendo la derivada de la función $V(x)$, la ecuación queda así:

$$\ddot{x} = x - x^3 - \delta \dot{x} + \gamma \cos(\omega t)$$

Y podemos reordenarla de la siguiente forma:

$$\ddot{x} + \delta \dot{x} - x + x^3 = \gamma \cos(\omega t)$$

Que no es ni mas ni menos que una ecuación de Duffing (Ecuación 5.1) parametrizada.

CAPÍTULO 6

Ecuaciones de Lotka–Volterra

Ya hemos analizado antes el caso Predador-Presa... las ecuaciones de Lotka-Volterra permiten modelar el comportamiento de este tipo de sistemas dinámicos en el que se producen crecimiento y decadencia de especies que conviven en un terreno, y bajo condiciones limitadas.

A pesar de la simpleza de estas ecuaciones, permiten modelar perfectamente un sistema poblacional como el descripto.

Las ecuaciones de Lotka-Volterra son las siguientes:

$$\dot{x} = r_1 \cdot x - p \cdot x \cdot y \quad (6.1)$$

$$\dot{y} = a \cdot p \cdot x \cdot y - r_2 \cdot y \quad (6.2)$$

Donde:

\dot{x} : Población de la presa (liebres)

\dot{y} : Población del predador (zorros)

p : Probabilidad de caza, $0 < p < 1$.

r_1 : Tasa de crecimiento de las presas.

r_2 : Tasa de mortalidad de predadores en ausencia de presa.

a : Tasa de crecimiento de predadores por cada presa cazada.

Las variables \dot{x} y \dot{y} representan las tasas de cambio instantáneas (derivadas) de las poblaciones de presas y predadores respectivamente. Así, si la velocidad de cambio de estas poblaciones fuera igual a cero, obtendríamos el **punto ideal de equilibrio** de ambas poblaciones.

Así, si $\dot{x} = 0$, tenemos de la ecuación 6.1 podemos despejar:

$$0 = r_1.x - p.x.y \Rightarrow p.x.y = r_1.x \Rightarrow y_e = \frac{r_1}{p}$$

Y de la ecuación 6.2 podemos hacer $\dot{y} = 0$ y despejar:

$$0 = a.p.x.y - r_2.y \Rightarrow a.p.x.y = r_2.y \Rightarrow x_e = \frac{r_2}{p.a}$$

En la Figura 108 se puede ver la variación temporal de ambas especies, mientras que en la 109 se aprecia el diagrama de fase. Nótese que dependiendo de los valores iniciales de ambas especies el diagrama de elongación se verá diferente, y el de fase tendrá una forma similar, pero variará su diámetro.

El diagrama de elongación muestra una solución periódica en la que un aumento de la población de las presas va seguido de un aumento de los predadores. Un gran número de predadores hace que disminuya la población de presas, lo que produce una temporada de hambruna entre los predadores.

Ahora podríamos preguntarnos qué habría pasado con diferentes cantidades iniciales de predadores y de presas. Lo que podríamos hacer ahora es repetir el proceso de simulación con diferentes condiciones para x_0 e y_0 , sin embargo se puede demostrar que independientemente de la cantidad de predadores y presas iniciales, las cantidades de ambas poblaciones varían y se conservan.

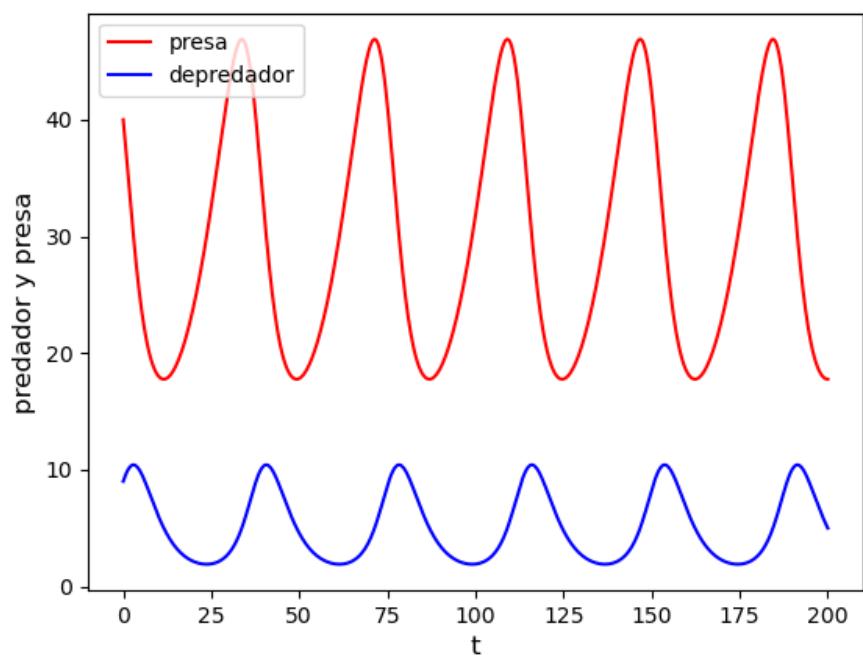


Figura 108: Lotka-Volterra - Elongación estable

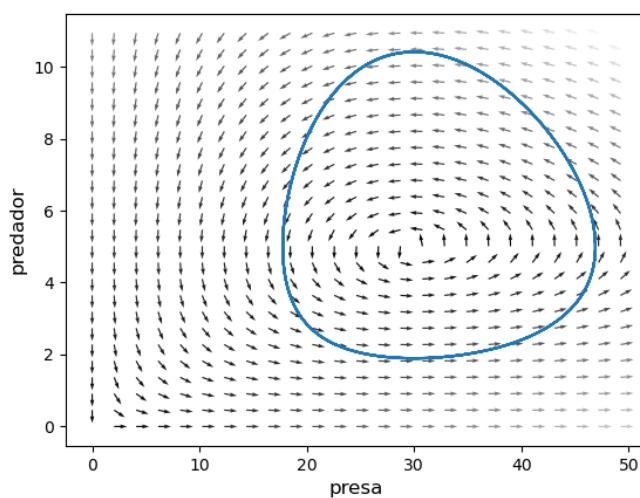


Figura 109: Lotka-Volterra - Fase estable

Vamos a generar una ecuación que nos va a permitir graficar las fases de contorno.

Partamos de las ecuaciones del modelo, dividamos la ecuación 6.2 por la ecuación 6.1:

$$\frac{\dot{y} = a.p.x.y - r_2.y}{\dot{x} = r_1.x - p.x.y}$$

O, lo que es lo mismo:

$$\frac{\frac{dy}{dt} = a.p.x.y - r_2.y}{\frac{dx}{dt} = r_1.x - p.x.y}$$

Separando la igualdad, y cancelando los dt tenemos:

$$\frac{dy}{dx} = \frac{a.p.x.y - r_2.y}{r_1.x - p.x.y}$$

Sacando factor común y en el numerador del segundo término, y x en el denominador del segundo término:

$$\frac{dy}{dx} = \frac{y.(a.p.x - r_2)}{x.(r_1 - p.y)}$$

Reordenando:

$$\frac{(r_1 - p.y)}{y} dy = \frac{(a.p.x - r_2)}{x} dx$$

Aplicando integración m.a.m.:

$$\int \frac{(r_1 - p.y)}{y} dy = \int \frac{(a.p.x - r_2)}{x} dx$$

Reordenando y cancelando según corresponda:

$$\int \left(\frac{r_1}{y} - p \right) dy = \int \left(a.p - \frac{r_2}{x} \right) dx$$

Reordenando:

$$\int \frac{r_1}{y} dy - \int p dy = \int a.p dx - \int \frac{r_2}{x} dx$$

Resolviendo las integrales:

$$r_1 \log(y) - p.y + C_1 = a.p.x - r_2 \log(x) + C_2$$

Reordenando y unificando las constantes de integración C_1 y C_2 en C :

$$C = r_1 \log(y) - py + r_2 \cdot \log(x) - a.p.x$$

Ecuación que nos permite realizar un diagrama de contornos como el que se ve en la Figura 110.

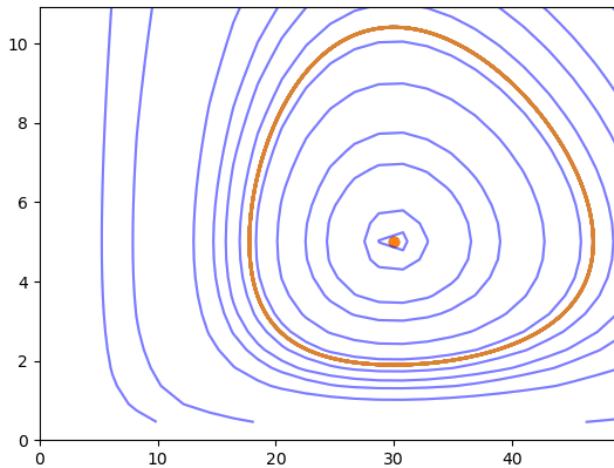


Figura 110: Lotka-Volterra - Fase Contorno

6.1 Mejorando el modelo

Además, podemos mejorar este modelo agregando algún parámetro adicional.

Como se puede observar, este modelo tiene algunas deficiencias propias de su simplicidad y derivadas de las hipótesis bajo las que se ha formulado. Una modificación razonable es cambiar el modelo de crecimiento de las presas en ausencia de depredadores, suponiendo que en vez de aumentar de forma exponencial, lo hacen según una función logística.

Podríamos definir la siguiente función logística:

$$l = a \frac{1 + me^{-t/\tau}}{1 + ne^{-t/\tau}}$$

Parametrizada con $a = 1$, $m = 0$, $n = 1$, y $\tau = 1$ queda:

$$l = 1 - \frac{1}{1 + e^{-t}}$$

Y representada en código sería algo así (ver `lv1.py`):

```
def logistic_curve(t, a=1, m=0, n=1, tau=1):
    e = np.exp(-t / tau)
    return a * (1 + m * e) / (1 + n * e)
```

Esta curva crece de forma similar a una exponencial al principio, moderándose después y estabilizándose asintóticamente en un valor. Este modelo de crecimiento representa mejor las limitaciones en el número de presas debidas al medio (falta de alimento, territorio).

La gráfica de esta función es similar a otras ecuaciones logísticas que vimos antes: Figura [111](#)

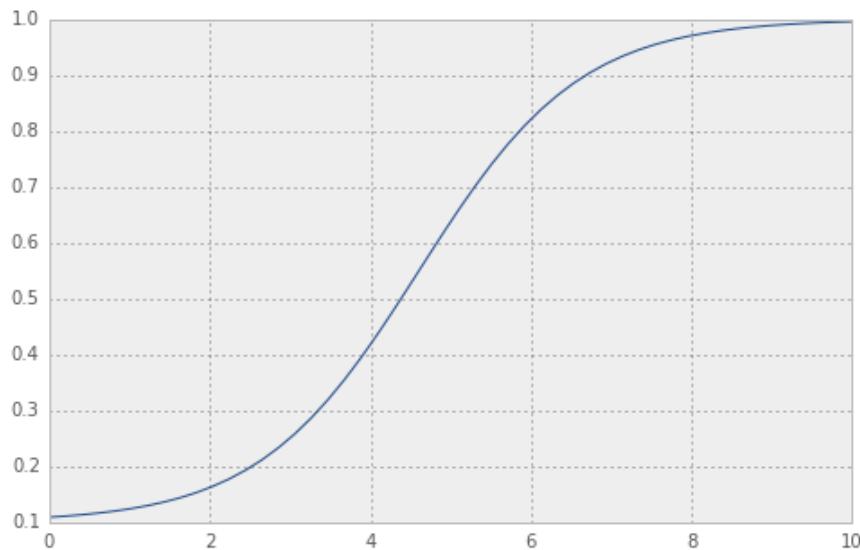


Figura 111: Lotka-Volterra - Ecuación logística

Si incorporamos este comportamiento logístico mediante un parámetro adicional, cuadrático, por ejemplo, en una de las ecuaciones de estado de Lotka-Volterra, por ejemplo la ecuación que define la evolución de la población de presas, Ecuación ??, forzaríamos a la población de predadores a modificar también su comportamiento, ya ambas poblaciones se generan de manera competitiva, y los valores de una de ella dependen de los de la otra en cada instante de tiempo.

Agreguemos un parámetro cuadrático en la ecuación ??, por ejemplo, kx^2 y analicemos la evolución del sistema completo.

Las ecuaciones nuevas con este cambio serían las siguientes:

$$\dot{x} = r_1.x - k.x^2 - p.x.y \quad (6.3)$$

$$\dot{y} = a.p.x.y - r_2.y \quad (6.4)$$

Si establecemos, por ejemplo, los valores iniciales $r_1 = 0,1$, $p = 0,02$, $r_2 = 0,3$, $a = 0,01$ y finalmente $k = 0,001$, la gráfica de elongación queda como se ve en la Figura 112

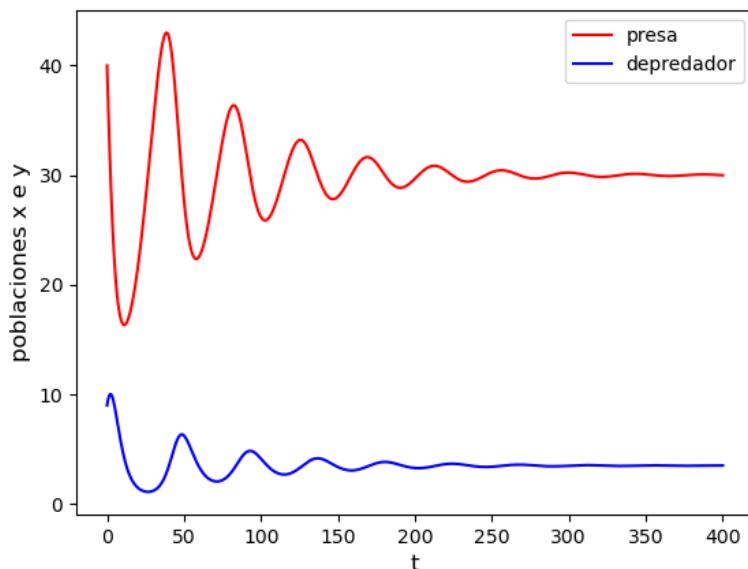


Figura 112: Lotka-Volterra - Logístico - Elongación

Se puede apreciar que ambas poblaciones (no solamente la de la presa) comienzan a interactuar y a atenuarse mutuamente, hasta que la amplitud de las mismas se estabiliza en lo que antes definimos como el **punto de equilibrio**.

El diagrama de fase de la Figura 113 muestra la interacción entre ambas poblaciones, y cómo, en la medida en que avanza el tiempo, se acercan ambas al punto de equilibrio, reduciendo sus amplitudes de oscilación.

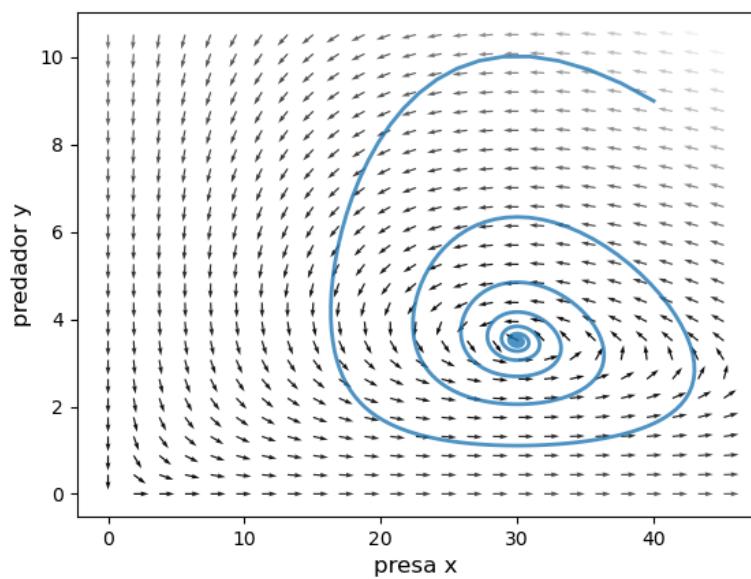


Figura 113: Lotka-Volterra - Logístico - Fase

CAPÍTULO 7

Control y Aptitud de modelos

edit

Una vez seleccionado el modelo, el tema principal es evaluar el comportamiento del mismo, es decir, responder si el modelo realizado es adecuado para el objetivo planteado. En general los datos simulados diferirán de los de la realidad (medidos). Esta diferencia se la denomina **incertidumbre** en la predicción. El origen de estas incertidumbres se da en las numerosas inexactitudes propias del modelo, desde la varianza de las mediciones, defectos en el modelo verbal, imprecisiones en la determinación de los parámetros, hasta errores de redondeo o de cálculo. Por lo tanto es importante primero caracterizar la incertidumbre del modelo y estimar la sensibilidad del modelo a las diversas variables usadas.

Una vez hecho esto, el segundo paso es generar datos lo más parecido posible a la realidad, a fin de estudiar la posibles salidas del modelo a escenarios distintos de los medidos. Ambos problemas plantean la necesidad de simular datos aleatorios de acuerdo con una función de probabilidad determinada. Esta función de probabilidad viene definida por la serie de observaciones realizadas sobre el sistema real. El cálculo de la sensibilidad e incertidumbre de un modelo puede realizarse por diversos medios. Entre ellos se pueden mencionar el método de la propagación de la incertidumbre, especialmente para estimar la influencia de cambios en los parámetros de entrada; el análisis de incertidumbre, que

estudia el efecto que tiene la incertidumbre de las variables de entrada en la incertidumbre de las variables de salida; el análisis de sensibilidad, que permite determinar el peso relativo en cambios en las variables de entrada; el estudio de los escenarios posibles, similar al anterior pero pesando cada variable de entrada con una probabilidad de ocurrencia, y analizando sus efectos a la salidas, etc. Todos estos métodos tienen versiones analíticas y numéricas. Debido a la complejidad de algunos métodos analíticos frente a sistemas con más de dos o tres variables, se han desarrollado varios métodos numéricos. En este capítulo desarrollaremos principalmente el método de simulación de Monte Carlo y similares, que nos permitirá la evaluación de un modelo y determinar su incertidumbre y sensibilidad.

7.1 Fuentes de la incertidumbre

Conviene primero distinguir entre variabilidad e incertidumbre de una variable, ya que usualmente se usan como sinónimos. Cuando se realiza una medición, la variable medida puede presentar cambios propios de su naturaleza (variabilidad); sin embargo el instrumento tiene imprecisiones o errores propios del método de medición (incertidumbre). Por ejemplo si mido la velocidad del viento, la experiencia nos indica que la intensidad del viento es altamente variable de acuerdo a la hora del día, estado del tiempo, etc., esta variabilidad es independiente del equipo de medición. En cambio, cuando mido la intensidad del viento con un anemómetro, éste tiene un error de medición propio de la inercia del instrumento, p.ej. de 0,5 m/s. Esta incertidumbre hará que fracciones inferiores a 0.5 m/s no sean detectadas por el instrumento, dando un error en la lectura. En conclusión, la variabilidad de una variable depende de la naturaleza física del proceso; mientras que la incertidumbre se origina por un error en su medición o determinación. Este último error en general se reduce aumentando el número de mediciones o mejorando el método de medición.

La incertidumbre en la predicción o modelación, como se acaba de decir proviene de una divergencia o diferencia entre los resultados predichos por el modelo y los valores observados de la realidad, generalmente obtenidos por medición u observación. En general en cada etapa de la formulación del modelo existen incertidumbres inherentes que se deben considerar. Veamos el origen de estos errores.

- **Planteo del problema:** El planteo del problema se sintetiza en el modelo verbal desarrollado, conforme al objeto del modelo. Un planteo incompleto o incorrecto de la naturaleza física del problema llevará a resultados bastante distintos de los observados. Por lo tanto si las diferencias son grandes, es probable que haya un error u omisión en la definición de las variables y parámetros del problema.
- **Diseño del modelo:** Aún cuando el modelo verbal sea adecuado, puede haber errores en el diseño del diagrama de efectos. La interrelación y realimentación puede ser incorrecta o incompleta.
- **Planteo matemático:** La resolución matemática del modelo y de las ecuaciones planteadas pueden ser incorrectas, o la selección de los métodos numéricos tienen poca resolución o tienen errores de aproximación; por ejemplo, poca definición en los pasos de integración, errores de redondeo, propagación de error en la inversión de una matriz casi singular, etc.
- **Inexactitud de los parámetros:** Muchas veces es necesario estimar un conjunto de factores y parámetros, de los que no se dispone de abundante experimentación. Una estimación pobre de estos parámetros conduce a errores sistemáticos importantes.
- **Documentación:** Los cálculos con valores inciertos o imprecisos producen errores que se propagan en el resultado final. Una inadecuada documentación de la precisión de estos parámetros y de cómo se propaga esta incertidumbre conduce a una baja calidad de los modelos.

Las primeras tres causas de incertidumbres o errores tiene un carácter cualitativo y dependen del conocimiento a priori y de la naturaleza del problema a resolver. En general estos errores son de tipo sistemático, y en principio salvables, en función de los conocimientos de la ciencia particular. En cada caso será el estado del arte de esa ciencia particular quien definirá el grado posible de certidumbre de los modelos planteados. Sin embargo los errores de las dos causas últimas, tiene un carácter propiamente numérico y son los de mayor importancia estadística. A la determinación de los dos últimos casos nos referiremos a continuación.

7.2 Análisis de Incertidumbre

Supongamos que nuestro sistema real bajo estudio está definido por una serie temporal de datos $y = (y_1, y_2, y_3, \dots, y_n)$ definida para los tiempos $t_1, t_2, t_3, \dots, t_n$ respectivamente. Y nuestro modelo genera una serie de datos de salidas que será función de un conjunto de variables de entrada X_1, X_2, \dots, X_n , y de un conjunto de parámetros de conocimiento imprecisos P_1, P_2, \dots, P_n , y del tiempo t . Esta imprecisión en el conocimiento instantáneo de X y P hacen que estas variables se conviertan en variables aleatorias.

$$\hat{y} = (\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n) = f(X_1, X_2, \dots, X_n; P_1, P_2, \dots, P_n, t)$$

La diferencia entre los datos medidos y , y los datos calculados \hat{y} : $(y - \hat{y})$, será en general también una variable aleatoria con una distribución cualquiera. Esto puede apreciarse en la Figura 114.

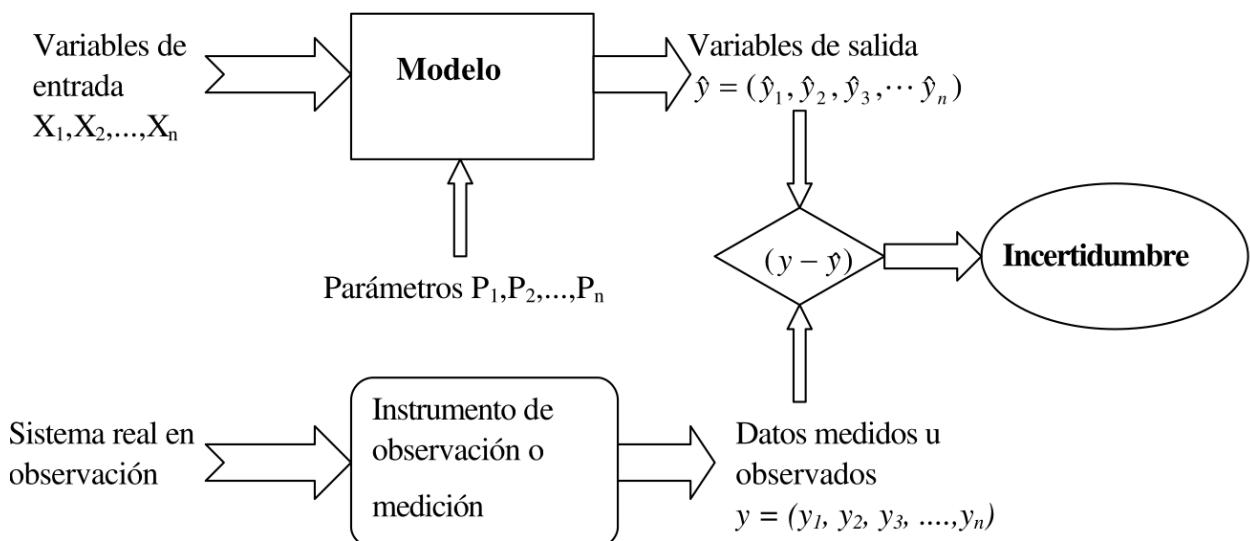


Figura 114: Incertidumbre

El análisis de incertidumbre se dedica entonces a estudiar la variación o imprecisión de \hat{y} respecto de y , que resulta de la variación en conjunto de los parámetros P , y variables de entrada del modelo X .

El análisis de incertidumbre se pregunta, por ejemplo:

- ¿Cuál es el rango de variación de \hat{y} , cuál es su media y si mediana?

- ¿Qué varianza y que función de distribución tiene \hat{y} ?

Estas preguntas implican conocer en detalle la distribución de probabilidades de la variable aleatoria \hat{y} . Si las variables de entrada son aleatorias y los parámetros también lo son, entonces la transformación realizada por el modelo producirá como salida, también una variable aleatoria \hat{y} . Por lo tanto el análisis de incertidumbre consiste en estimar la distribución de probabilidades de $\hat{y}F(\hat{y})$, pues este conocimiento lleva implícito las respuestas a las preguntas.

Los métodos desarrollados más abajo, consiste en determinar la función de distribución $F(\hat{y})$, a partir del conocimiento de las distribuciones de probabilidades de las variables de entrada X y los parámetros P .

7.3 Análisis de sensibilidad

Al análisis de incertidumbre, siempre le sigue el de la sensibilidad del modelo. El análisis de sensibilidad consiste en detectar qué parámetros y variables de entrada contribuyen en mayor medida a la incertidumbre de \hat{y} . Para determinar la sensibilidad, se estudia el cambio en la salida del modelo producido por un cambio en una variable o parámetro de entrada. De esta manera se identifica en qué proporción influye cada entrada en la salida, es decir qué variables de entrada son las más importantes en la incertidumbre del modelo. Posteriormente se indicarán métodos para determinar esta magnitud.

7.4 Pasos del análisis de incertidumbre

El método análisis de incertidumbre por muestreo de **Monte Carlo** y otros métodos similares se basan en una secuencia de simulación, generalmente caracterizada en tres pasos importantes:

1. Caracterización de los parámetros y variables de entrada, y generación de muestras aleatorias
2. Ejecución del modelo
3. Caracterización estadística de los valores de la salida

7.4.1 Caracterización de las entradas

El primer paso es estudiar el comportamiento aleatorio de las variables y parámetros de entrada, y asignar una función de distribución lo más realista posible. Luego se procede a tomar muestras aleatorias para cada variable o parámetro de entrada. Aquí es donde se aplican las técnicas de Monte Carlo o Muestreo Hipercubo Latino, que se verán más adelante.

Supongamos que tenemos tres variables de entradas X_1, X_2, X_3 y dos parámetros P_1, P_2 ; y se generan n números aleatorios por cada variables, tendremos unas matrices como las siguientes:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & x_{n3} \end{bmatrix} \text{y} \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \\ \vdots & \vdots \\ p_{n1} & p_{n2} \end{bmatrix}$$

Siendo x_{ni} y p_{ni} valores individuales y aleatorios de las variables X_1, X_2, X_3 , y de los parámetros P_1 y P_2 .

7.4.2 Ejecución del modelo

Una vez obtenido el conjunto de n muestras aleatorias para cada variable se corre el programa n veces; obteniéndose por lo tanto n salidas \hat{y} :

$$\hat{y} = (\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n)$$

7.4.3 Caracterización de las salidas

El conjunto de valores de salidas obtenidas \hat{y} , deben analizarse estadísticamente: obtener la función de distribución, cálculo de la media, desviación, etc.

7.5 Método de Monte Carlo

Este método se origina a mediados del siglo XIX como una forma de desarrollar una martingala que me permita simular los números de la ruleta. De allí que su nombre toma

el del famoso casino de la ciudad de Monte Carlo en el reinado de Mónaco. Sin embargo la forma actual y su justificación matemática se origina en la década de 1940 por John von Neumann durante las experiencias nucleares para desarrollar la primera bomba atómica.

Existen numerosos problemas en los que por su complejidad o por contener un número grande de variables, o por tratarse de variables aleatorias se hace difícil evaluar el comportamiento de estas variables. Por ello se trata de simular números aleatorios que representen adecuadamente la función de distribución de dichos números.

La mayoría de los generadores de números aleatorios (*random*) que se encuentran en los algoritmos y programas habituales de cálculo representan números aleatorios con una función de distribución uniforme. El tema central, entonces, es generar una secuencia de números aleatorios cuya distribución sea la deseada, pero a partir de una secuencia de números uniformes. Veamos la justificación matemática de este procedimiento.

Supongamos que u es una variable aleatoria que tiene una función de distribución acumulada $G(u)$ (en mayúsculas) o de probabilidad $g(u)$ (en minúsculas) conocida, y deseamos obtener valores de u que sean conformes a esa distribución $G(u)$.

En la Figura 115 se aprecia una función de distribución acumulada $G(u)$ en valores normalizados (de 0 a 1) cuyos valores indican la probabilidad de encontrar un valor u entre 0 y u . El objeto del método consiste en lo siguiente: dados un valor aleatorio de $G(u)$, a qué valor de u le corresponden ese valor de $G(u)$. En el gráfico se indican particularmente el valor de u correspondiente a la probabilidad de 0.2, de 0.5 y de 0.7 respectivamente. Esto es, si del generador aleatorio hubiese salido el valor $G(u) = 0,2$, a qué valor de u le corresponde?; en este caso particular $u = 19,5$ aproximadamente, mientras que para $G(u) = 0,5$, $u = 28,5$.

u	g(u)	G(u)	G(u) norm
1	0.1	0.1	0.002
5	1.3	1.4	0.028
10	2.25	3.65	0.073
15	3.47	7.12	0.142
20	6.2	13.32	0.266
25	8.4	21.72	0.434
30	10.3	32.02	0.640
35	8.4	40.42	0.808
40	5.26	45.68	0.914
45	2.5	48.18	0.964
50	1.54	49.72	0.994

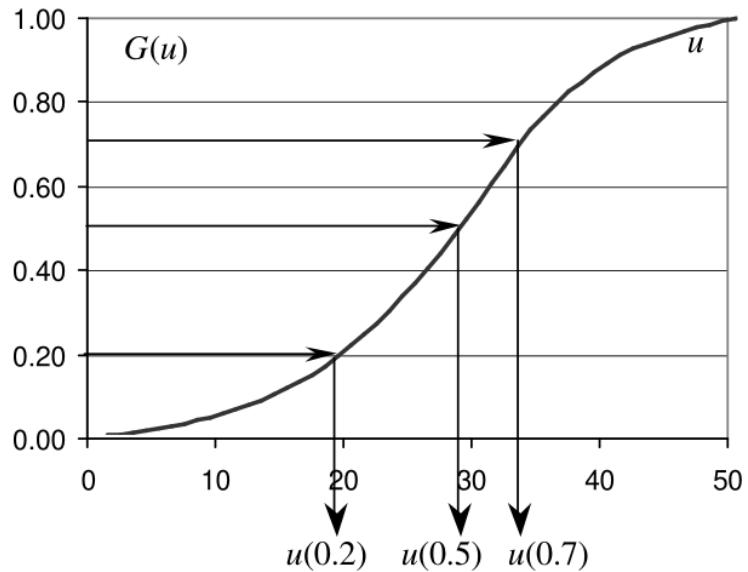


Figura 115: Montecarlo - Distribución acumulada

Si se dispone de la tabla que acompaña a la Figura (Figura 116), entonces, conceptualmente, el método sería equivalente a encontrar el valor de u para un $G(u)$ dado, lo implica entrar en la tabla en forma inversa, por ejemplo, encontrar el valor de u para $G(u) = 0,26$.

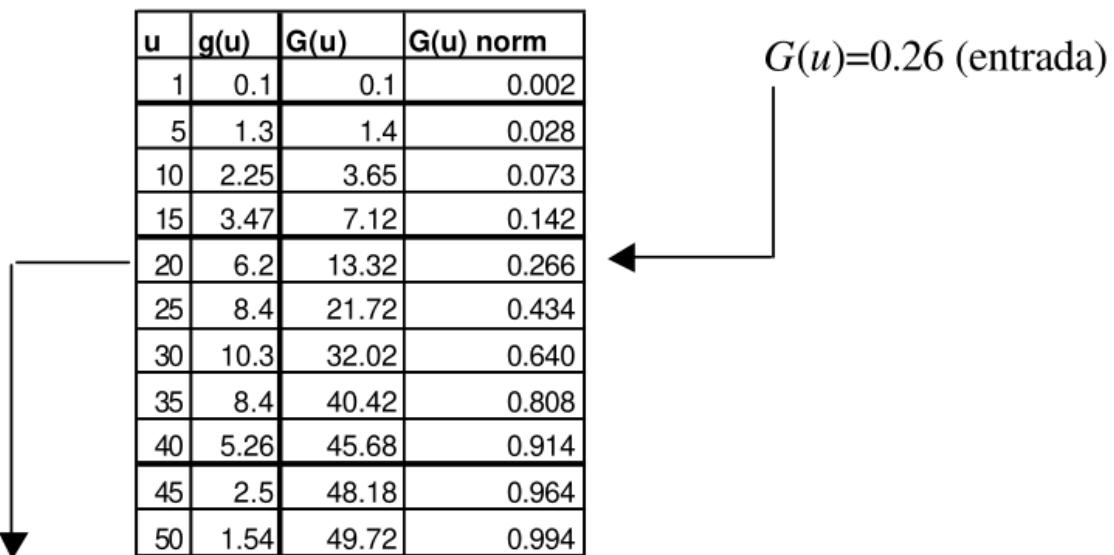


Figura 116: Montecarlo - Tabla ejemplo

Es evidente que esta forma de obtener valores de la variable u , mediante una búsqueda en tabla no es la más apropiada. Veremos un método computacional más adecuado.

Supongamos que tenemos un generador de números aleatorios, con distribución uniforme, que genera una secuencia de n números entre 0 y 1: $v = (v_1, v_2, \dots, v_i, \dots, v_n)$, que lo igualamos a la distribución de la variable u :

$$v = G(u)$$

Se desea encontrar los valores de u , que correspondan a los valores de $v = G(u)$:

$$u = G^{-1}(v)$$

esto, es se debe encontrar la inversa de la función $G(u)$.

Si se conoce la función matemática y es sencilla, entonces, G^{-1} , podrá calcularse analíticamente, por ejemplo, $v = G(u) = e^{-u}$; entonces $u = \ln(G(u)) = \ln(v)$; pero esto no es siempre el caso.

Si se conoce la función de distribución de probabilidades de u , $g(u)$, y recordando que $G(u)$ es la función de distribución acumulada de $g(u)$; entonces, para hallar un u_i particular, tal que su integral sea v_i , se debe integrar desde el mínimo valor de u , hasta que la función acumulada $G(u)$ sea igual a v_i :

$$v_i = G(u_i) = \int_{-\infty}^{u_i} g(\tau) d\tau \approx \sum_{min}^{u_i} g(\tau_i) \Delta_i$$

Siendo Δ_i el rango de la variable u , (en la figura 115, el rango es 5: 1-5, 5-10, etc).

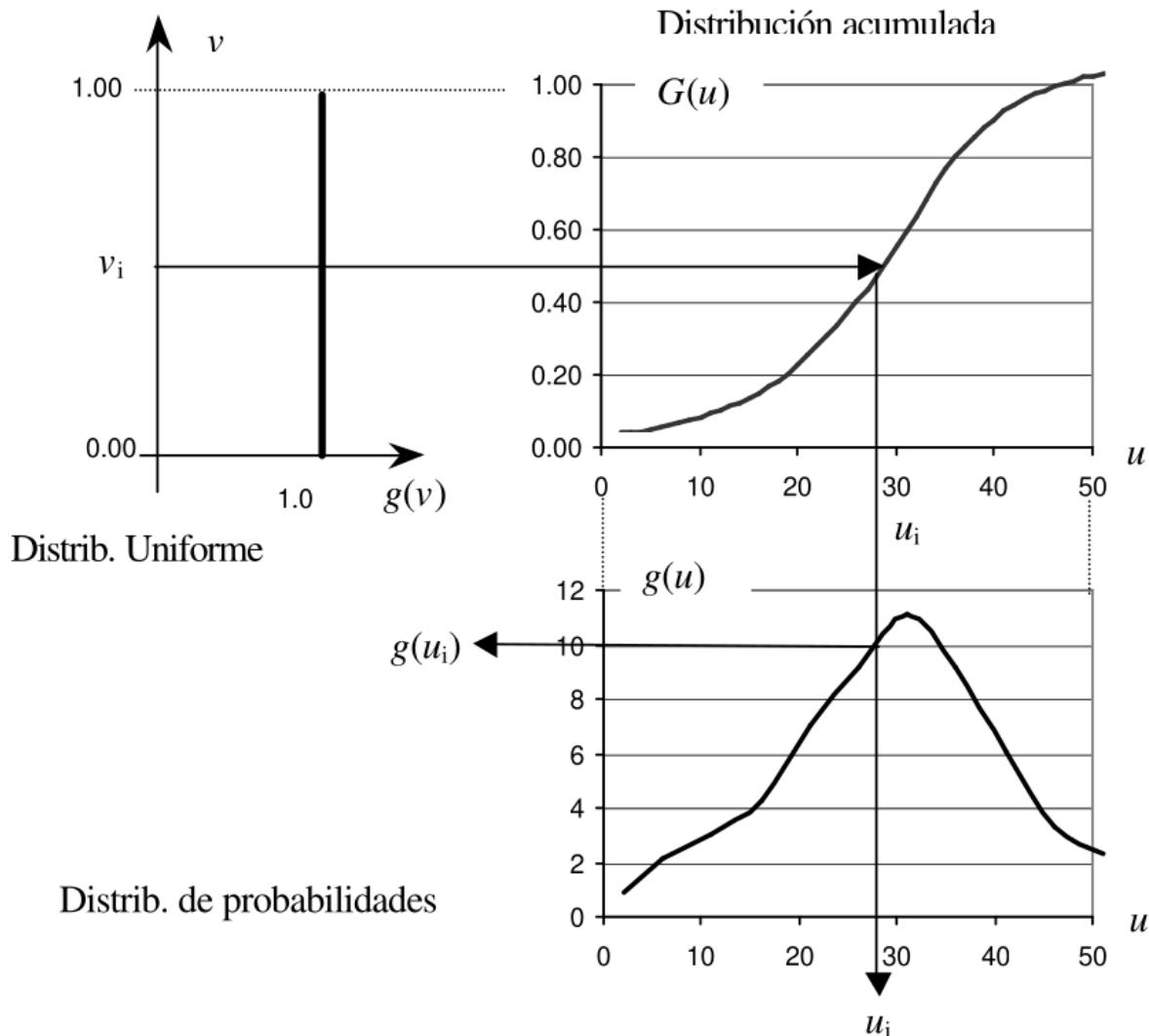


Figura 117: Montecarlo - Acceso inverso

Este mismo procedimiento debe calcularse n veces para todos los v_i deseados. De esta manera conformamos la matriz de entrada con números aleatorios que responden a la función de distribución respectiva para cada entrada o parámetro.

Cuando se realiza este muestreo aleatorio por Monte Carlo, deben tenerse en cuenta algunas precauciones:

1. El método es más preciso mientras mayor es el número de muestras que se tome. El intervalo de confianza (varianza) del resultado es inversamente proporcional a la raíz cuadrada del número de simulaciones realizadas ($\frac{1}{\sqrt{n}}$).

2. Depende de la función de distribución del generador de número aleatorio. Es importante primero constatar, que el generador aleatorio sea verdaderamente uniforme. Para ello se debe estudiar al función de probabilidad del generador usado, para la cantidad n deseada. Recordemos que los algoritmos de los generadores (seudo-) aleatorios, son cíclicos y tienen una distribución casi uniforme (todos los números tiene igual probabilidad de salir) para un número limitado de n . Así algunos generadores aseguran una uniformidad de probabilidades para $n = 10000$, otros 100000 veces etc. Depende de cuan larga sea la secuencia que se desea generar.
3. Cuando se generan, relativamente pocas muestras, se corre el riesgo que no todas las posibles partes de la curva de distribución haya sido muestreada, con lo que le modelo no es evaluada para esas condiciones de entrada.

7.5.1 Muestreo estratificado

El método de muestreo por Monte Carlo, dijimos, tiene el inconveniente que su precisión aumenta con el número de muestras realizadas. Esto exige que se realicen numerosas corridas del programa de simulación del modelo, lo que requiere tiempo y disponibilidad computacional, En algunos modelos complejos, este problema puede ser una limitación seria.

Una forma de salvar estos inconvenientes es la planteada por el método de muestreo de Monte Carlo estratificado denominado *Muestreo Hipercubo Latino*.

En este método se divide en tres partes:

1. **Estratificación:** La estratificación consiste en dividir la función de distribución de entrada en m intervalos de igual probabilidad. Es decir obtenemos m sub-intervalos, cuya función de densidad de probabilidad queda dividida en áreas iguales. En la Figura 118 se muestra función de densidad de probabilidad normal dividida en 5 áreas de 0.2 cada una.
2. **Muestreo en cada intervalo:** El muestreo se realiza dentro de cada intervalo de la misma forma que en el método de Monte Carlo. Así por ejemplo, se toman 10 muestras aleatorias en cada intervalo. De esta forma se asegura que toda la función de distribución está igualmente muestreada.

3. **Apareamiento aleatorio:** Cada variable de entrada ha sido muestreado n veces en cada intervalo, siguiendo el ejemplo anterior, tendríamos 50 valores por cada variable:

$$x_{11}, x_{12}, x_{13}, \dots, x_{150}$$

$$x_{21}, x_{22}, x_{23}, \dots, x_{250}$$

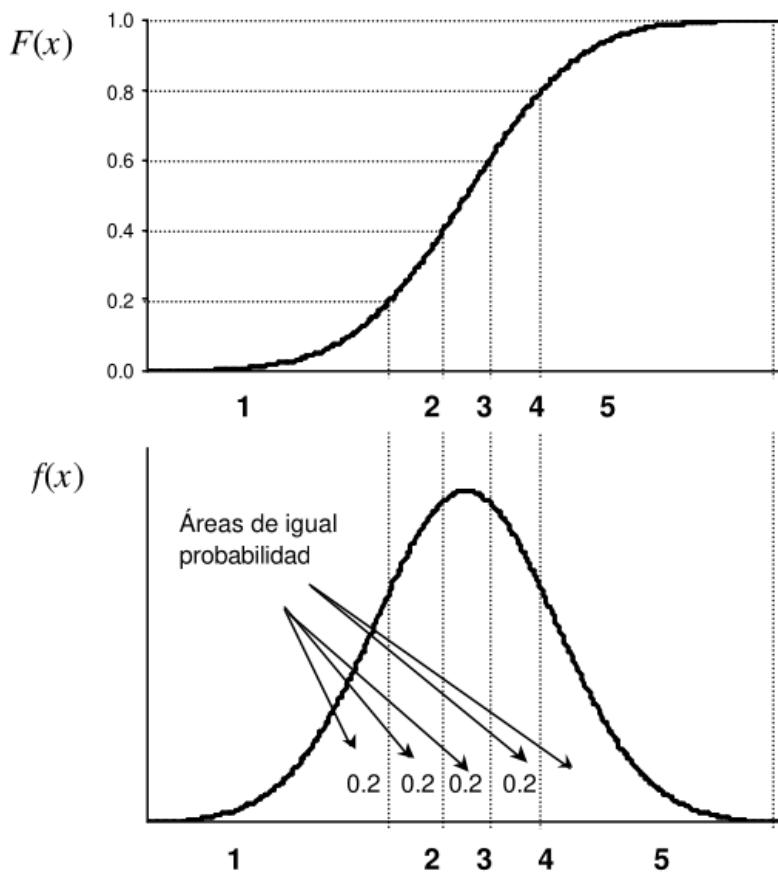


Figura 118: Montecarlo - Muestreo estratificado

A fin de evitar correlaciones indeseadas, productos de la estratificación secuencial, las muestras de cada variable se combinan en forma aleatoria. Para ello se le asigna a cada muestra un número de orden ascendente, por ejemplo: $1, 2, 3, \dots, N$. Una vez obtenido este orden, se debe permutar aleatoriamente para cada variable.

Así, por ejemplo, podríamos generar la siguiente secuencia permutada:

$$x_{13}, x_{11}, x_{14}, \dots$$

$$x_{21}, x_{24}, x_{22}, \dots$$

Y finalmente calcular las salidas \hat{y} considerando estos datos de las entradas x por columna, es decir:

$$\hat{y}_1 = f(x_{13}, x_{21})$$

$$\hat{y}_2 = f(x_{11}, x_{24})$$

$$\hat{y}_3 = f(x_{14}, x_{22})$$

⋮

La ventaja del método del muestreo estratificado es que con menor número de muestras se asegura una cobertura de todo el rango de posibles valores de cada variable.

CAPÍTULO 8

Bibliografía de consulta

Los temas expuestos en el presente trabajo fueron extraídos de diferentes fuentes, sitios webs, y apuntes anteriores de la materia.

A continuación se listan algunas de las fuentes consultadas.

Estructura y fundamentos generales

<https://github.com/AllenDowney/ModSimPy>

<https://greenteapress.com/wp/modsimpy/> (modelos bikeshare, poblacionales y epidemiológicos)

Python

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html>

Ecuaciones diferenciales

http://hplgit.github.io/prog4comp/doc/pub/_p4c-solarized-Python019.html

Modelos poblacionales

http://hplgit.github.io/prog4comp/doc/pub/_p4c-solarized-Python020.html

http://hplgit.github.io/prog4comp/doc/pub/_p4c-solarized-Python021.html

https://en.wikipedia.org/wiki/Feigenbaum_constants

https://en.wikipedia.org/wiki/Mortality_rate

Masa-Resorte-Amortiguador

http://hplgit.github.io/prog4comp/doc/pub/_p4c-solarized-Python022.html

Duffing

https://en.wikipedia.org/wiki/Duffing_equation

<https://guillaumecantin.pythonanywhere.com/animation/1/>

<https://scipython.com/blog/a-quartic-oscillator/>

<https://scipython.com/blog/the-duffing-oscillator/>

<http://www.cfm.brown.edu/people/dobrush/am34/Mathematica/ch3/duffing.html>

<https://github.com/vkulkar/Duffing>

Lotka Volterra

<https://scipy-cookbook.readthedocs.io/items/LotkaVolterraTutorial.html>

<https://scipy.github.io/old-wiki/pages/Cookbook/LotkaVolterraTutorial>

<https://pybonacci.org/2015/01/05/ecuaciones-de-lotka-volterra-modelo-presa-depredador/>

https://es.wikipedia.org/wiki/Funci%C3%B3n_log%C3%ADstica

https://github.com/AlexS12/pybonacci_contributions/blob/master/lotka-volterra/Lotka-Volterra.ipynb

https://en.wikipedia.org/wiki/Lotka-Volterra_equations

<https://github.com/dh4gan/lotka-volterra>

Puliafito, Enrique. Apuntes de cátedra de Modelos y Simulación - Capítulo 2: Modelos.

Universidad de Mendoza. Facultad de Ingeniería. 2003

Control de aptitud y Montecarlo

Puliafito, Enrique. Apuntes de cátedra de Modelos y Simulación - Capítulo 4: Control de aptitud de un modelo. Universidad de Mendoza. Facultad de Ingeniería. 2003

General

http://hplgit.github.io/prog4comp/doc/pub/_p4c-bootstrap-Python002.html

<https://pybonacci.org/2012/10/15/el-salto-de-felix-baumgartner-en-python/>