Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA
METROPOLITANA
Unidad Cuajimalpa

Sistemas de E.D. y Retratos Fase

Maximiliano Barajas Sánchez

10 de octubre de 2023

1. Introducción

A lo largo de esta práctica analizaremos distintos sistemas de ecuaciones diferenciales que solucionaremos con ayuda de SciPy numéricamente en Python además de que obtendremos los retratos fase de las soluciones dados parámetros condiciones de inicio particulares para cada modelo.

2. Metodología

Inicialmente se compartieron en clase los siguientes ejemplos de sistemas de Ecuaciones Diferenciales de tal manera que nos familiarizáramos con las herramientas a utilizar a lo largo de la práctica; la ecuación logística, la ecuación asociada al movimiento armónico amortiguado, el sistema de ecuaciones del modelo presa depredador y finalmente el sistema de ecuaciones del modelo bruselador, mas tarde aplicamos lo aprendido al modelado de diversos sistemas de batalla con distintas condiciones.

3. Desarrollo

Inicialmente realizamos una serie de ejemplos de como se solucionan ecuaciones diferenciales comunes con ayuda de odeint d SciPy. A continuación observamos los ejemplos realizados.

3.1. Ecuación Logística

Esto se realizo con el objetivo de resolver la ecuación diferencial siguiente:

$$\frac{dP}{dt} = kP(M - P)$$

Se realizó el siguiente código:

```
import numpy as np
from scipy.integrate import odeint
import plotly.graph_objects as go
def logistica(P,t,k,M):
    dPDT = k * P * (M - P)
    return dPDT
P0 = 1
t = np.linspace(0, 10)
k = 0.1
M = 10
P = odeint(logistica, P0, t, args=(k, M))
```

```
t_values = t
P_values = P[:, 0]
fig = go.Figure(data=go.Scatter(x=t_values, y=P_values, mode='lines'))
fig.update_layout(
    title='Solucion numerica de la ecuacion logistica',
    xaxis=dict(title='t'),
    yaxis=dict(title='P'),
)
fig.show()
```

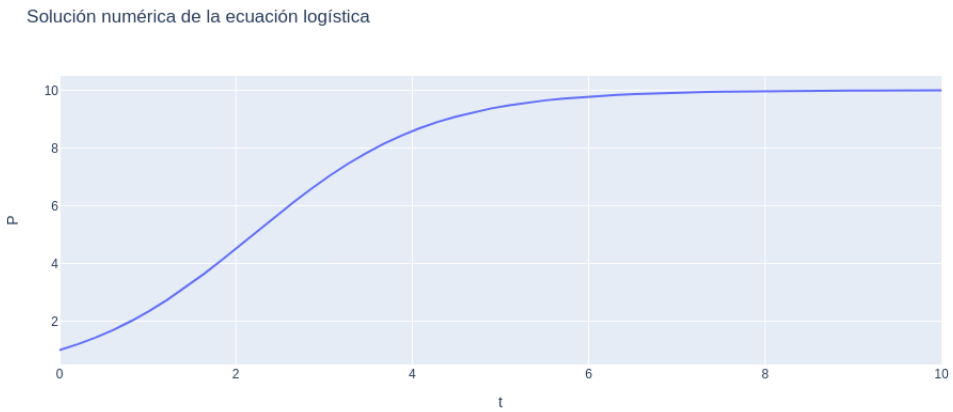

Figura 1: Solución de la Ecuación Logística

En este caso al considerar a k positiva resulta en una función logística de forma cóncava en el intervalo de interés.

3.2. Oscilador Amortiguado

En este caso se resolvió la siguiente ecuación diferencial que describe un movimiento oscilatorio amortiguado:

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0$$

Sin embargo se reescribe de la siguiente forma de tal suerte que podamos representarlo como un sistema:

$$\begin{cases} \frac{dx}{dt} = v \\ \frac{dv}{dt} = -\frac{cv+kx}{m} \end{cases}$$

Se realizó el siguiente código para solucionar numéricamente el sistema:

```
import numpy as np
from scipy.integrate import odeint
import plotly.graph_objects as go

def oscilador_amortiguado(y, t, m, c, k):
    x, v = y
    a = -(c * v + k * x) / m
    dydt = np.array([v, a])
    return dydt
```

```
y0 = np.array([1, 0])
t = np.linspace(0, 5, 200)
m, c, k = 0.5, 1, 50
y = odeint(oscilador_amortiguado, y0, t, args=(m, c, k))
v = y[:, 1]
P = m * v
fig = go.Figure()
fig.add_trace(go.Scatter(x=t, y=P, mode='lines', name='P vs t'))
fig.update_layout(
    title='Osciladorꣿ Amortiguado',
    xaxis=dict(title='t'),
    yaxis=dict(title='P')
)
fig.show()
```

Obteniendo la siguiente gráfica de la solución numérica:

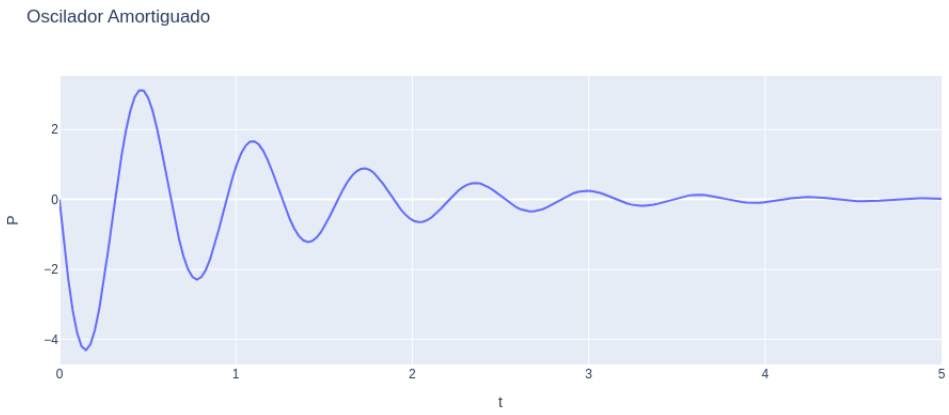

Figura 2: Solución del Oscilador Amortiguado

Notemos que en este caso al posee una constante de resorte bastante mayor a la masa obtenemos un movimiento oscilatorio que tarda relativamente mas de lo usual en neutralizarse.

3.3. Presa Depredador

De igual manera realizamos el mismo proceso para el modelo clásico Presa-Depredador solucionando el siguiente sistema de ecuaciones:

$$\begin{cases} \frac{dx}{dt} & = x(a - py) \\ \frac{dy}{dt} & = y(-b + qx) \end{cases}$$

En este caso se realizó el código a continuación el cuál arroja una gráfica de la cantidad de presas y de depredadores en función del tiempo.

```
import numpy as np
from scipy.integrate import odeint
import plotly.graph_objects as go
def predator_prey(pp, t, a, p, b, q):
    x, y = pp
    dxdt = x * (a - p * y)
    dydt = y * (-b + q * x)
```

```
    dppdt = np.array([dxdt, dydt])
    return dppdt

pp0 = np.array([70, 10])
t = np.linspace(0, 30, 200)
args = (0.2, 0.005, 0.5, 0.01)
pp = odeint(predator_prey, pp0, t, args=args)
fig = go.Figure()

fig.add_trace(go.Scatter(x=t, y=pp[:, 0], mode='lines', name='Presas', line=dict
    (color='black')))
fig.add_trace(go.Scatter(x=t, y=pp[:, 1], mode='lines', name='Depredadores',
    line=dict(color='blue')))
fig.update_layout(
    title='Modelo␣Presa␣Depredador',
    xaxis=dict(title='t'),
    yaxis=dict(title=''),
    legend=dict(x=0.7, y=1)
)
fig.show()
```

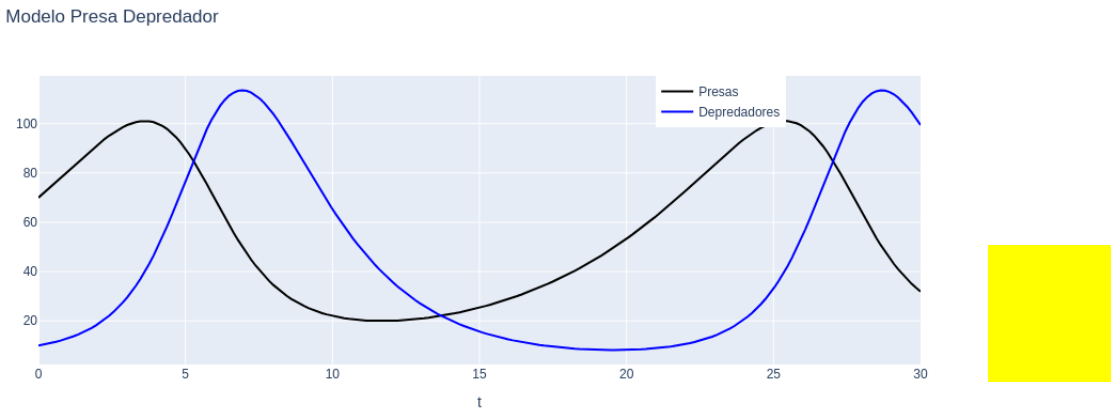

Figura 3: Presas y Depredadores contra el tiempo

3.4. Brusselator

Finalmente se hizo un ejemplo del modelo Brusselator en el cuál se graficó el retrato de fase, esto a raíz de que se nos pidió realizar el retrato de fase de los sistemas de ecuaciones diferenciales referentes al modelado de batalla, el código se encuentra a continuación:

```
import numpy as np
import plotly.graph_objects as go
from scipy.integrate import odeint
a = 0.55
b = 0.25
def brusselator(u, t):
    du = np.zeros(2)
    du[0] = 1 - (b + 1) * u[0] + a * u[0] * u[1]
    du[1] = b * u[0] - a * u[0] * u[0] * u[1]
```

```
    return du
t = np.linspace(0, 100, 1000)
u_inits = [
    ([-1, 3.5]),
    ([-1, 2.5]),
    ([3, -3]),
    ([-1, 0.5]),
    ([-1, 2]),
    ([3, 3.5])
]
fig = go.Figure()
for u_init in u_inits:
    u = odeint(brusselator, u_init, t)
    fig.add_trace(go.Scatter(x=[z[0] for z in u], y=[z[1] for z in u], mode='
       lines', name=str(u_init)))
fig.update_layout(
    title='Brusselator',
    xaxis=dict(title='x'),
    yaxis=dict(title='y'),
    legend=dict(x=0, y=1)
)
fig.show()
```

El retrato de fases resultante es la gráfica siguiente:

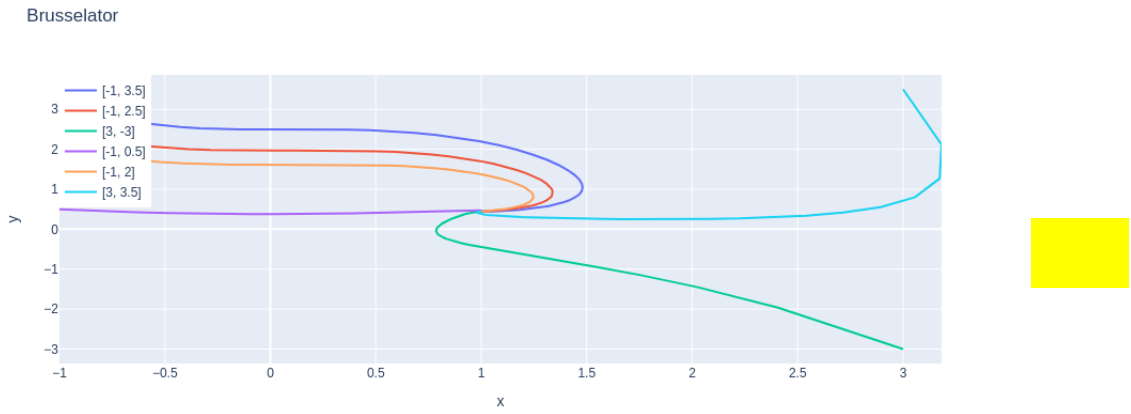

Figura 4: Retratos de fase del modelo Brusselator

Notemos que el retrato muestra como todas las condiciones iniciales convergen a un punto de equilibrio final.

4. Modelos de Batalla

A lo largo de esta sección analizaremos los modelos de batalla realizados bajo diversas condiciones junto con su retrato de fases.

4.1. Modelo de fuego directo

El primer modelo que implementamos fue el modelo de batalla de fuego directo el cual es descrito bajo el siguiente sistema de ecuaciones:

$$\begin{cases} \frac{dx}{dt} &= -\alpha y(t) \\ \frac{dy}{dt} &= -\beta x(t) \end{cases}$$

Este modelo consiste en el mas simple de todos y solo posee dos parámetros además de las condiciones iniciales los cuales son α y β, el código que realicé para este modelo se encuentra a continuación:

```
import numpy as np
from scipy.integrate import odeint
import plotly.graph_objects as go
#Sistema de batalla de fuego directo simple
def fuego_directo(y, t, alpha=20, beta=15):
    a, b = y
    da_dt = -alpha * b
    db_dt = -beta * a
    return [da_dt, db_dt]
#Funcion que encuentra el tiempo en el que un equipo se queda sin tropas
def encontrar_tiempo_final(alpha, beta, a0, b0, t_max):
    t = np.linspace(0, t_max, 1000)
    y0 = [a0, b0]
    solucion_sistema = odeint(fuego_directo, y0, t, args=(alpha, beta))
    a_values, b_values = solucion_sistema.T
    tiempo_final = None
    for i in range(len(t)):
        if a_values[i] <= 0 or b_values[i] <= 0:
            tiempo_final = t[i]
            break
    return tiempo_final
#Funcion para graficar las funciones soluciones desde el tiempo 0 al tiempo
   final
def graficar_simulacion(alpha, beta, a0, b0, t_max):
    tiempo_final = encontrar_tiempo_final(alpha, beta, a0, b0, t_max)
    if tiempo_final is None:
        print("Ambos sobreviven")
        return
    t = np.linspace(0, tiempo_final, 1000)
    y0 = [a0, b0]
    solucion_sistema = odeint(fuego_directo, y0, t, args=(alpha, beta))
    a_valores, b_valores = solucion_sistema.T
    #Creamos la grafica de plotly
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=t, y=a_valores, mode='lines', name='Tropas A'))
    fig.add_trace(go.Scatter(x=t, y=b_valores, mode='lines', name='Tropas B'))
    fig.update_layout(
        xaxis_title='Tiempo',
        yaxis_title='Tropas',
        title='Fuego directo',
    )
    #Mostramos la grafica
    fig.show()
    print(f"Algun equipo se queda sin tropas en el tiempo t = {tiempo_final:.24f
       }")
```

Inicialmente consideramos los valores de 20 y 15 para ellos respectivamente, junto con una cantidad de tropas de 1000 y 800 como condiciones iniciales obteniendo la siguiente gráfica:

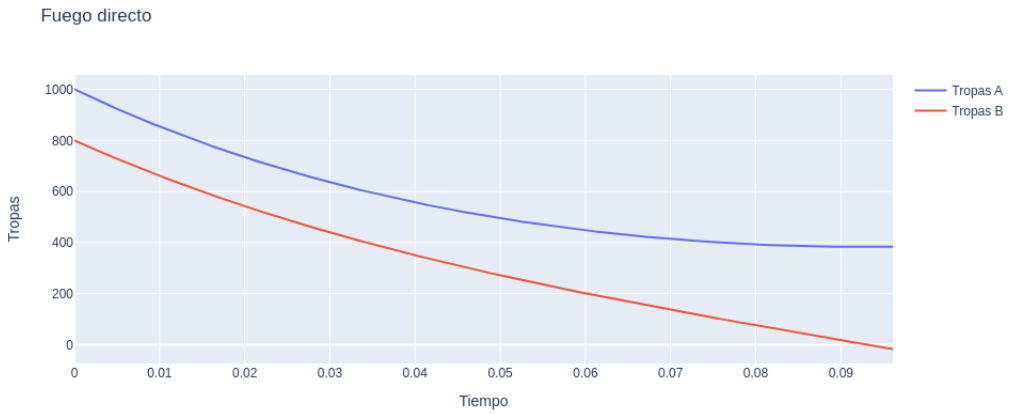

Figura 5: Modelo de batalla de fuego directo con $\alpha = 20$ y $\beta = 15$,

Ahora bien, notemos en la figura que el ejercito con la mayor cantidad de tropas resulta dominante a pesar de que la capacidad de combatir según los parámetros que elegimos, el código para realizar los retratos de fase con una serie de condiciones iniciales y mismos parámetros $\alpha = 20$ y $\beta = 15$:

```
import numpy as np
import plotly.graph_objs as go
from scipy.integrate import odeint
def retrato_fase_fuego_directo(uinits, max_time=20):
    #Creamos un intervalo grande de tiempo
    t = np.linspace(0, max_time, 1000000)
    #Creamos la gr fica de plotly
    fig = go.Figure()
    #Resolvemos el sistema dada la lista de condiciones iniciales
    for uinit in uinits:
        u = odeint(fuego_directo, uinit, t)
        a_values = u[:, 0]
        b_values = u[:, 1]
        for i in range(len(t)):
            #Determinamos el tiempo final de cada soluci n con el que
               graficaremos la l nea
            if a_values[i] <= 0 or b_values[i] <= 0:
                tiempo_final = t[i]
                a_values = a_values[0:i + 1]
                b_values = b_values[0:i + 1]
                break
        fig.add_trace(go.Scatter(x=a_values, y=b_values, mode='lines', name=str(
           uinit)))
    fig.update_layout(
        xaxis=dict(scaleanchor="y", scaleratio=1),
        yaxis=dict(scaleanchor="x", scaleratio=1),
        xaxis_title='Tropas de A',
        yaxis_title='Tropas de B',
        title='Retrato de fase de Fuego directo',
        legend_title='Condiciones iniciales'
```

```
    )
    fig.show()

#Lista de condiciones iniciales
uinits = [[1000, 800], [2000, 1900], [8000, 6000],[6000,5000],[3000,4000]]
retrato_fase_fuego_directo(uinits)
```

La gráfica resultante es la siguiente del retrato de fases es el siguiente, he decidido utilizar una lista de condiciones iniciales para generar los retratos fase, usaré la misma lista para el resto de retratos con la finalidad de poder comparar los modelos:

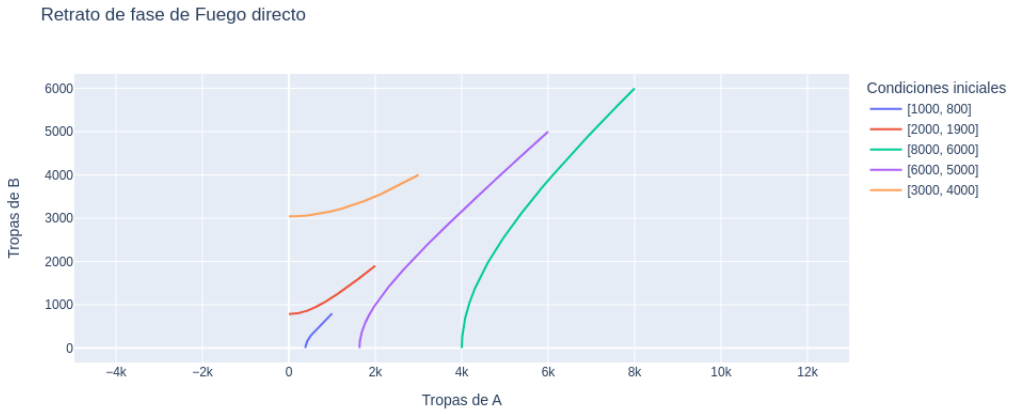

Figura 6: Retratos de fase con $\alpha = 20$ y $\beta = 15$,

Notemos que en general las tropas del ejercito A dominan sobre B en diversas condiciones iniciales. Si cambiamos los parámetros α y β a $\alpha = 29$ y $\beta = 12$, obtenemos inicialmente con las mismas condiciones iniciales de nuestro primer ejemplo gráfica a continuación:

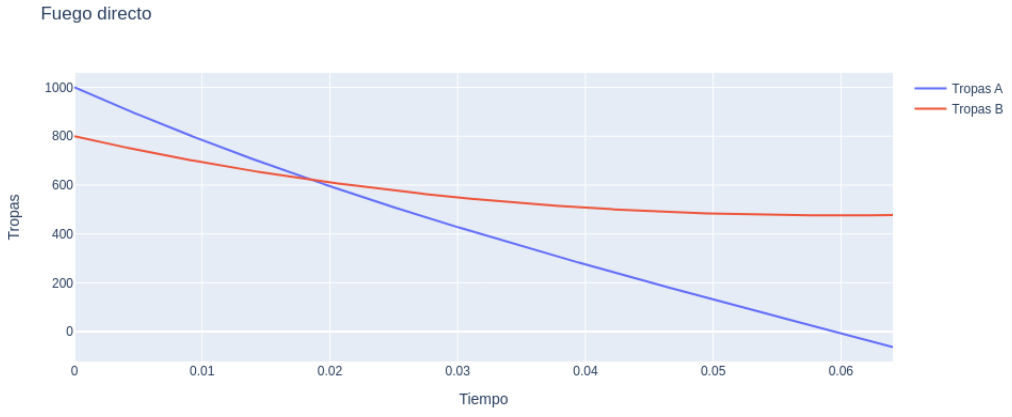

Figura 7: Modelo de batalla de fuego directo con $\alpha = 29$ y $\beta = 12$,

Notemos que en este caso predominan las tropas de B sobre las de A, esto también lo podemos observar en el retrato de fase de las soluciones con estos nuevos parámetros y el mismo conjunto de condiciones iniciales:

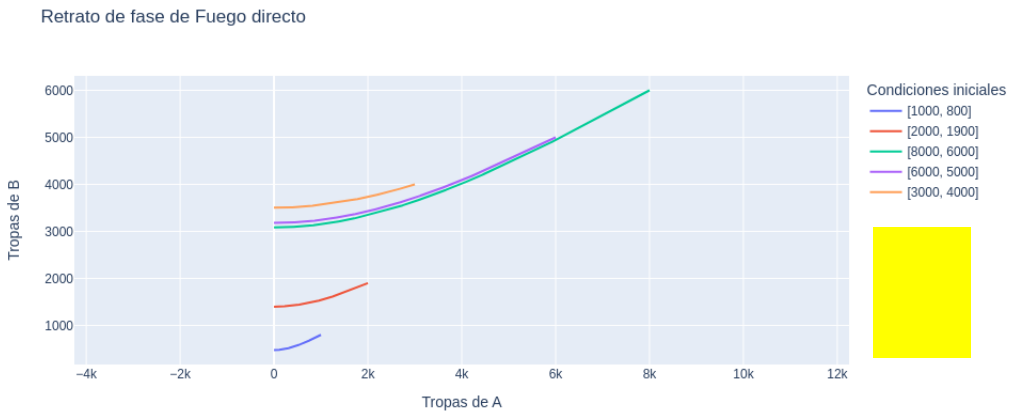

Figura 8: Retratos de fase con $\alpha = 29$ y $\beta = 12$,

Notemos que en este retrato las tropas de B dominan sobre A en todos los casos.

4.2. Modelo con refuerzos

En este modelo consideramos 2 funciones en el tiempo que representan la cantidad de tropas de suministro como refuerzos a cada uno de los grupos de tropas, este modelo se ve regido a través del siguiente sistema

$$\begin{cases} \frac{dx}{dt} &= -\alpha y(t) + f(t) \\ \frac{dy}{dt} &= -\beta x(t) + g(t) \end{cases}$$

En este caso decidí en primera instancia utilizar los valores de $\alpha = 20$ y $\beta = 15$ y para las funciones de refuerzos $f = 34000 * cos(99 * t)$ y $g = 35000 * sin(58 * t)$ junto con las mismas condiciones iniciales (cantidad de tropas) , el código a continuación:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
import plotly.graph_objects as go
#Funci n de refuerzos en cada instante de tiempo
def funciones_refuerzos(t):
    r_a = 34000 * np.cos(99*t)
    r_b = 35000* np.sin(58*t)
    return r_a, r_b
#Modelo de batalla con refuerzos
def batalla_con_refuerzos(y, t, alpha=20, beta=15, r_func=None):
    a, b = y
    if r_func:
        r_a, r_b = r_func(t)
        da_dt = -alpha * b + r_a
        db_dt = -beta * a + r_b
    else:
        da_dt = -alpha * b
        db_dt = -beta * a
    return [da_dt, db_dt]
#Funci n para encontrar el tiempo en el que uno de los equipos se queda sin
   tropas
def encontrar_tiempo_final(alpha, beta, a0, b0, r_func, t_max):
```

```
    t = np.linspace(0, t_max, 1000)
    y0 = [a0, b0]
    solucion_sistema = odeint(batalla_con_refuerzos, y0, t, args=(alpha, beta,
        r_func))
    a_valores, b_valores = solucion_sistema.T
    tiempo_final = None
    for i in range(len(t)):
        if a_valores[i] <= 0 or b_valores[i] <= 0:
            tiempo_final = t[i]
            break
    return tiempo_final
#Funci n para graficar la simulaci n del modelo entre 0 y el tiempo final
    obtenido
def graficar_simulacion_refuerzos(alpha, beta, a0, b0, r_func, t_max):
    tiempo_final = encontrar_tiempo_final(alpha, beta, a0, b0, r_func, t_max)
    if tiempo_final is None:
        print("Ambos sobreviven")
        return
    t = np.linspace(0, tiempo_final, 1000)
    y0 = [a0, b0]
    solucion_sistema = odeint(batalla_con_refuerzos, y0, t, args=(alpha, beta,
        r_func))
    a_valores, b_valores = solucion_sistema.T
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=t, y=a_valores, mode='lines', name='Force A'))
    fig.add_trace(go.Scatter(x=t, y=b_valores, mode='lines', name='Force B'))
    fig.update_layout(
        xaxis_title='Tiempo',
        yaxis_title='Tropas',
        title='Simulaci n de batalla con refuerzos',
    )
    fig.show()
    print(f"ALg n equipo se queda sin tropas en el tiempo t = {tiempo_final:.24
        f}")

# Par meteros
alpha = 20
beta = 15
a0 = 1000
b0 = 800
t_max = 32
graficar_simulacion_refuerzos(alpha, beta, a0, b0, funciones_refuerzos, t_max)
```

Obtenemos inicialmente la siguiente gráfica:

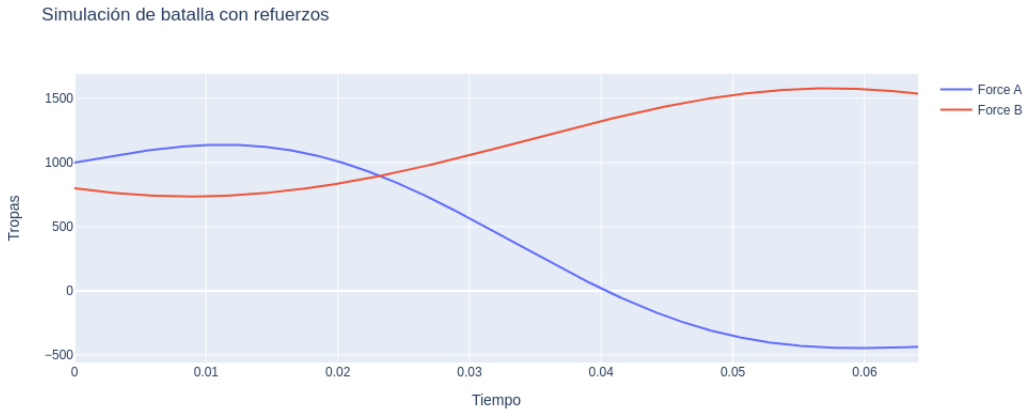

Figura 9: Modelo de batalla de fuego directo con $\alpha = 20$ y $\beta = 15$,

Notemos en este caso que las tropas B sobrepasan a las Tropas A y dominan sobre ellas, la interpretación que yo le doy a esto es la diferencia de periodos como la amplitud de las funciones de refuerzos, a continuación se muestra el código para el retrato de fases para este modelo:

```
import numpy as np
import plotly.graph_objs as go
from scipy.integrate import odeint
def retrato_refuerzos(uinits,r_func, max_time=20):
    t = np.linspace(0, max_time, 1000000)
    fig = go.Figure()
    for uinit in uinits:
        u = odeint(batalla_con_refuerzos, uinit, t,args=(alpha, beta, r_func))
        a_values = u[:, 0]
        b_values = u[:, 1]
        for i in range(len(t)):
            if a_values[i] <= 0 or b_values[i] <= 0:
                tiempo_final = t[i]
                a_values = a_values[0:i + 1]
                b_values = b_values[0:i + 1]
                break
        fig.add_trace(go.Scatter(x=a_values, y=b_values, mode='lines', name=str(
            uinit)))
    fig.update_layout(
        xaxis=dict(scaleanchor="y", scaleratio=1),
        yaxis=dict(scaleanchor="x", scaleratio=1),
        xaxis_title='Tropas de A',
        yaxis_title='Tropas de B',
        title='Retrato de fase de batalla con refuerzos',
        legend_title='Condiciones iniciales'
    )
    fig.show()
uinits = [[1000, 800], [2000, 1900], [8000, 6000],[6000,5000],[3000,4000]]
retrato_refuerzos(uinits,funciones_refuerzos)
```

La gráfica resultante de este código para el retrato de fases es la siguiente:

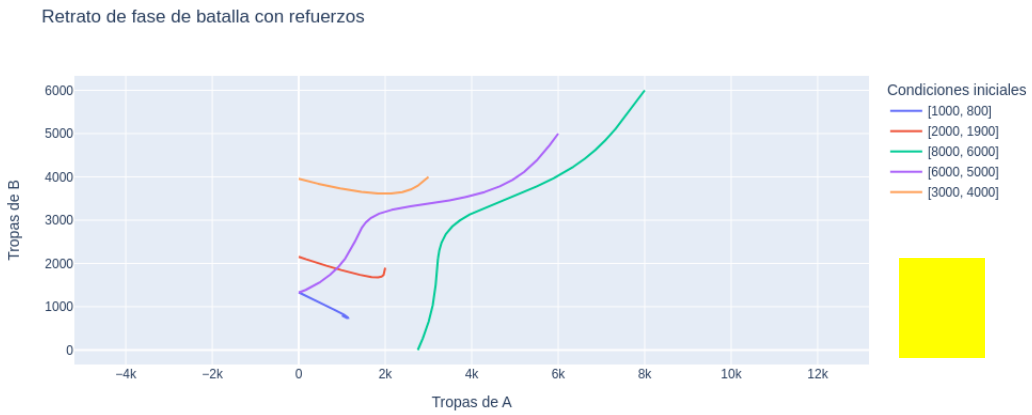

Figura 10: Retratos de fase del modelo con refuerzos con $\alpha = 20$ y $\beta = 15$, utilizando la misma lista de condiciones iniciales

Notamos que mayoritariamente las tropas de B son dominantes en la gran mayoría de los casos aunque en todos los casos es intermitente el que tanto domina uno de los grupos de tropas. Por otra parte si modificamos los valores de los parámetros a $\alpha = 29$ y $\beta = 12$ obtenemos la siguiente gráfica de la simulación de batalla con las mismas funciones de refuerzos:

Figura 11: $Simulacióndebatallaconrefuerzoscon\alpha = 29$ y $\beta = 12$, utilizando como condiciones iniciales 1000 y 800

En este caso los retratos de fase resultan en la siguiente gráfica:

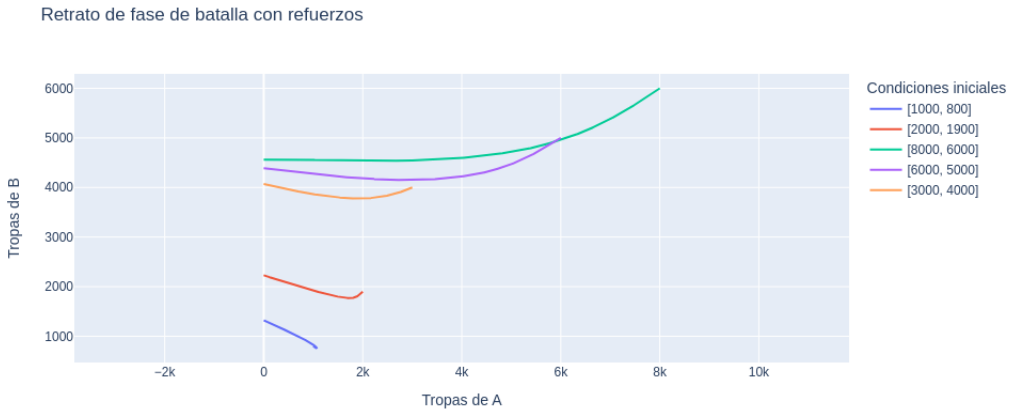

Figura 12: Retratos de fase del modelo con refuerzos con $\alpha = 29$ y $\beta = 12$, utilizando la misma lista de condiciones iniciales

Notamos en este caso que las tropas de B dominan en prácticamente todos los casos de las condiciones iniciales con las que instanciamos el modelo con refuerzos bajo estos parámetros.

4.3. Modelo con artillería

En este modelo se consideran funciones que representan la capacidad armamento de cada ejercito de manera proporcional a la capacidad militar definida por los parámetros del modelo de fuego directo:

$$\begin{cases} \frac{dx}{dt} & = -\alpha y(t)x(t) \\ \frac{dy}{dt} & = -\beta x(t)y(t) \end{cases}$$

El código para solucionar este sistema con los parámetros $\alpha = 20$ y $\beta = 15$ y por otra parte condiciones iniciales 1000 y 800 se encuentra a continuación:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
import plotly.graph_objects as go
#Funciones de artilleria en el tiempo
def funciones_artilleria(t):
    r_a = 3*np.exp(5*t)
    r_b = 2*np.exp(6*t)
    return r_a, r_b
#Modelo con artilleria
def batalla_con_artilleria(y, t, alpha=20, beta=15, a_func=None):
    a, b = y
    if a_func:
        a_a, a_b = a_func(t)
        da_dt = -alpha * b *a_a
        db_dt = -beta * a*a_b
    else:
        da_dt = -alpha * b
        db_dt = -beta * a
    return [da_dt, db_dt]
#Funci n para encontrar el tiempo en que un equipo se queda sin tropas
def encontrar_tiempo_final(alpha, beta, a0, b0, r_func, t_max):
```

```
    t = np.linspace(0, t_max, 1000)
    y0 = [a0, b0]
    solucion_sistema = odeint(batalla_con_artilleria, y0, t, args=(alpha, beta,
        r_func))
    a_valores, b_valores = solucion_sistema.T
    tiempo_final = None
    for i in range(len(t)):
        if a_valores[i] <= 0 or b_valores[i] <= 0:
            tiempo_final = t[i]
            break
    return tiempo_final
def graficar_simulacion_artilleria(alpha, beta, a0, b0, r_func, t_max):
    tiempo_final = encontrar_tiempo_final(alpha, beta, a0, b0, r_func, t_max)
    if tiempo_final is None:
        print("Ambos sobreviven")
        return
    t = np.linspace(0, tiempo_final, 1000)
    y0 = [a0, b0]
    solucion_sistema = odeint(batalla_con_artilleria, y0, t, args=(alpha, beta,
        r_func))
    a_valores, b_valores = solucion_sistema.T
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=t, y=a_valores, mode='lines', name='Force A'))
    fig.add_trace(go.Scatter(x=t, y=b_valores, mode='lines', name='Force B'))
    fig.update_layout(
        xaxis_title='Tiempo',
        yaxis_title='Tropas',
        title='Batalla con artilleria',
    )
    fig.show()
    print(f"ALg n equipo se queda sin tropas en el tiempo t = {tiempo_final:.24
        f}")
# Par metros
alpha = 20
beta = 15
a0 = 1000
b0 = 800
t_max = 32
graficar_simulacion_artilleria(alpha, beta, a0, b0, funciones_artilleria, t_max)
```

Obtenemos la siguiente gráfica como solución a este problema particular:

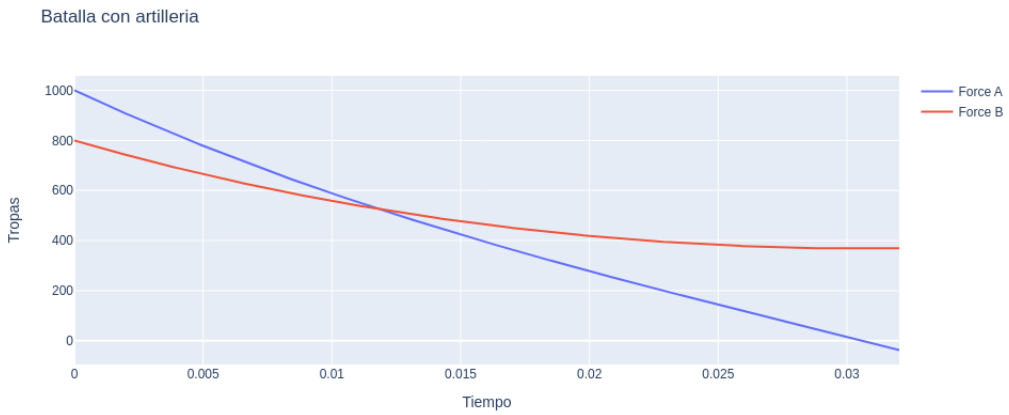

Figura 13: Modelo de artillería con $\alpha = 20$ y $\beta = 15$

Notamos en este caso que la artilleria genera que las tropas de B a pesar de iniciar con una menor cantidad resulten dominantes sobre las tropas de A después de una cantidad de tiempo considerable,por otra parte a continuación el código para realizar los retratos fase de este modelo:

```
import numpy as np
import plotly.graph_objs as go
from scipy.integrate import odeint

def retrato_artilleria(uinits,r_func, max_time=20):
    t = np.linspace(0, max_time, 1000000)
    fig = go.Figure()
    for uinit in uinits:
        u = odeint(batalla_con_artilleria, uinit, t,args=(alpha, beta, r_func))
        a_values = u[:, 0]
        b_values = u[:, 1]
        for i in range(len(t)):
            if a_values[i] <= 0 or b_values[i] <= 0:
                tiempo_final = t[i]
                a_values = a_values[0:i + 1]
                b_values = b_values[0:i + 1]
                break
        fig.add_trace(go.Scatter(x=a_values, y=b_values, mode='lines', name=str(
            uinit)))
    fig.update_layout(
        xaxis=dict(scaleanchor="y", scaleratio=1),
        yaxis=dict(scaleanchor="x", scaleratio=1),
        xaxis_title='Tropas de A',
        yaxis_title='Tropas de B',
        title='Retrato de Fase batalla con artilleria',
        legend_title='Condiciones iniciales'
    )
    fig.show()

uinits = [[1000, 800], [2000, 1900], [8000, 6000],[6000,5000],[3000,4000]]
retrato_artilleria(uinits,funciones_artilleria)
```

El retrato fase utilizando nuestra lista usual de condiciones iniciales resulta en la siguiente gráfica:

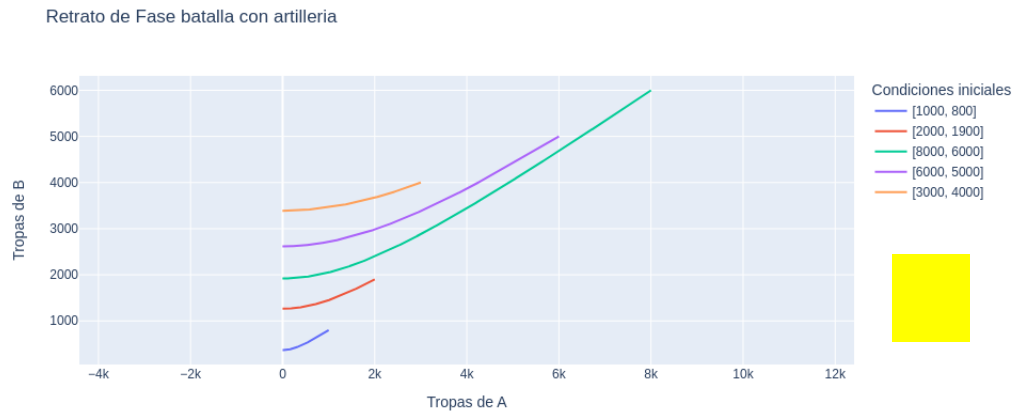

Figura 14: Retratos de fase del modelo con artillería con $\alpha = 20$ y $\beta = 15$, utilizando la misma lista de condiciones iniciales

Notemos que generalmente las tropas de B resultan dominantes sobre las tropas de A en prácticamente todos los casos a mi parecer esto se debe a la diferencia inicial entre ambos grupos de tropas en condiciones iniciales. Si decidimos cambiar los parámetros a $\alpha = 29$ y $\beta = 12$ para el primer ejemplo obtenemos la siguiente solución:

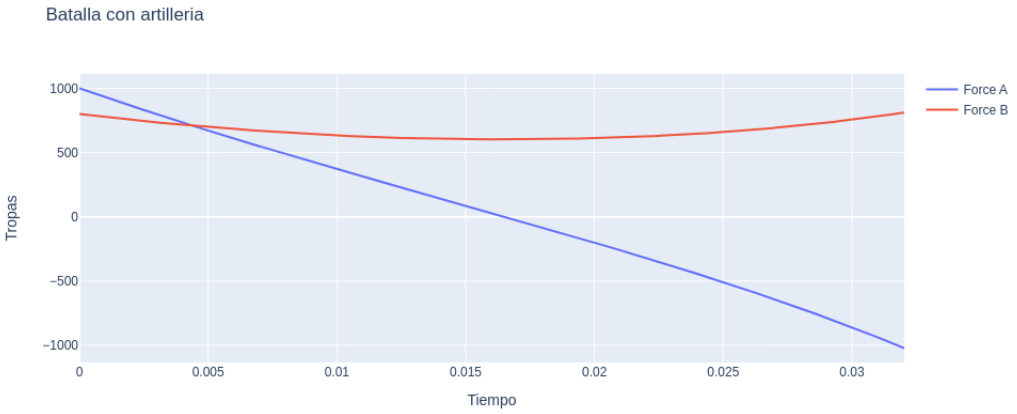

Figura 15: Modelo de artillería con $\alpha = 29$ y $\beta = 12$

Notemos que esto beneficia a las tropas de B y resultan dominantes en un menor tiempo que el caso anterior, nuevamente atribuyo esto a las diferencias de tropas iniciales. Si dibujamos los retratos fase de nuestra lista de condiciones iniciales obtenemos la siguiente gráfica:

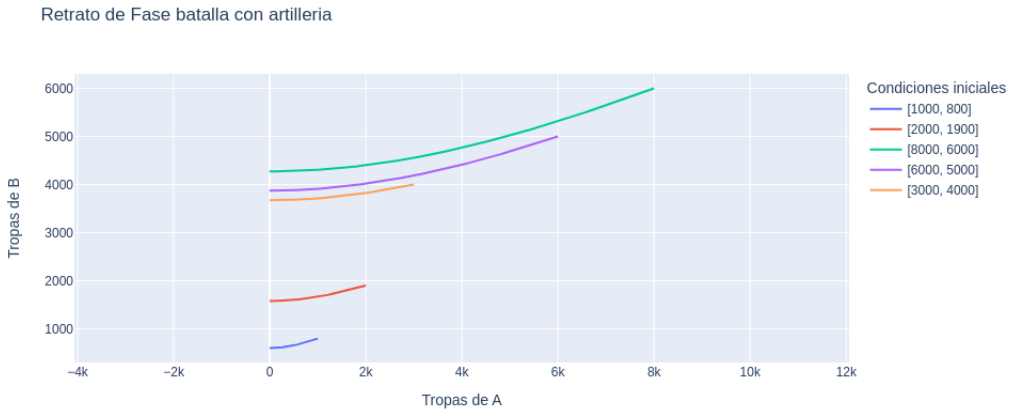

Figura 16: Retratos de fase del modelo con artillería con $\alpha = 29$ y $\beta = 12$, utilizando la misma lista de condiciones iniciales

Notemos que en este retrato fase se acentuan mas rápido las curvas resultando en las tropas B dominantes en absolutamente todos los casos.

4.4. Artillería con refuerzos

Ahora analizaremos el sistema que representa la simulación de batalla con refuerzos y artillería, que se puede modelar con el siguiente sistema:

$$\begin{cases} \frac{dx}{dt} &= -\alpha y(t)x(t) + f(t) \\ \frac{dy}{dt} &= -\beta x(t)y(t) + g(t) \end{cases}$$

El código para resolver un problema particular que siga este modelo es el siguiente:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
import plotly.graph_objects as go
def funciones_artilleria(t):
    r_a = 3*np.exp(5*t)
    r_b = 2*np.exp(6*t)
    return r_a, r_b
def funciones_refuerzos(t):
    r_a = 34000 * np.cos(99*t)
    r_b = 35000* np.sin(58*t)
    return r_a, r_b
def batalla_con_artilleria_refuerzos(y, t, alpha=20, beta=15, funcion_artilleria
   =None,funcion_refuerzos=None):
    a, b = y
    if funcion_artilleria and funcion_refuerzos:
        a_a, a_b =funcion_artilleria(t)
        r_a,r_b=funcion_refuerzos(t)
        da_dt = -alpha * b *a_a + r_a
        db_dt = -beta * a*a_b + r_b
    else:
        da_dt = -alpha * b
        db_dt = -beta * a
```

```
    return [da_dt, db_dt]
def encontrar_tiempo_final(alpha, beta, a0, b0, a_func,r_func,t_max):
    t = np.linspace(0, t_max, 1000)
    y0 = [a0, b0]
    solution = odeint(batalla_con_artilleria_refuerzos, y0, t, args=(alpha, beta
        , a_func,r_func))
    a_values, b_values = solution.T
    tiempo_final = None
    for i in range(len(t)):
        if a_values[i] <= 0 or b_values[i] <= 0:
            tiempo_final = t[i]
            break
    return tiempo_final
def graficar_simulacion_artilleria_refuerzos(alpha, beta, a0, b0, a_func,r_func,
    t_max):
    tiempo_final = encontrar_tiempo_final(alpha, beta, a0, b0, a_func,r_func,
        t_max)
    if tiempo_final is None:
        print("Both forces survive the battle.")
        return
    t = np.linspace(0, tiempo_final, 1000)
    y0 = [a0, b0]
    solution = odeint(batalla_con_artilleria_refuerzos, y0, t, args=(alpha, beta
        , a_func,r_func))
    a_values, b_values = solution.T
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=t, y=a_values, mode='lines', name='Force A'))
    fig.add_trace(go.Scatter(x=t, y=b_values, mode='lines', name='Force B'))
    fig.update_layout(
        xaxis_title='Tiempo',
        yaxis_title='Tropas',
        title='Batalla con artilleria y refuerzos',
    )
    fig.show()
    print(f"Either Force A or Force B reaches 0 at time t = {tiempo_final:.24f}"
        )
# Param tros
alpha = 20
beta = 15
a0 = 1000
b0 = 800
t_max = 32
graficar_simulacion_artilleria_refuerzos(alpha, beta, a0, b0,
    funciones_artilleria,funciones_refuerzos, t_max)
```

En este caso obtenemos la siguiente gráfica para el primer ejemplo de este modelo, utilizando las funciones de refuerzos del ejemplo de la sección anterior y los valores para los parámetros de igual manera de la sección anterior:

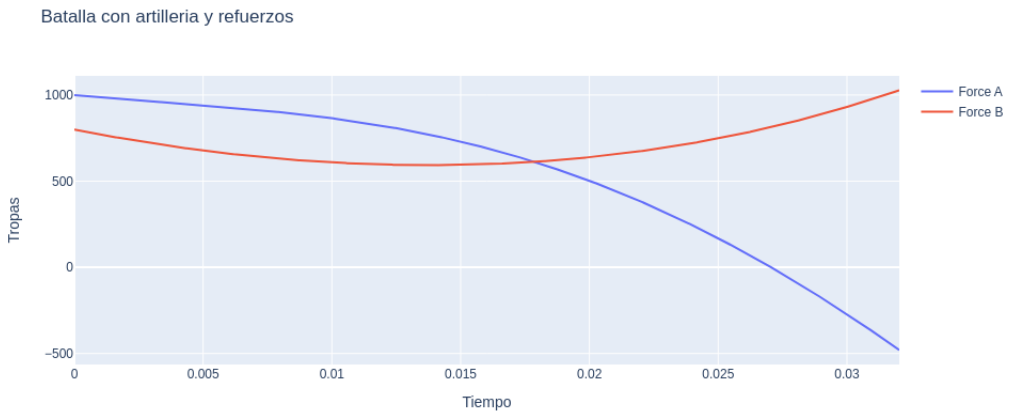

Figura 17: Modelo de artillería y refuerzos con $\alpha = 20$ y $\beta = 15$

Notamos que las tropas de B además de crecer sobrepasan a las tropas A de manera mas pronunciada que el caso anterior, a continuación los retratos fase tomando en cuenta la lista de condiciones iniciales que hemos usado a lo largo de la práctica:

Figura 18: Retratos de fase del modelo con artillería y refuerzos con $\alpha = 20$ y $\beta = 15$, utilizando la misma lista de condiciones iniciales

Notemos que generalmente las tropas B son dominantes sobre las tropas A nuevamente como la sección anterior a diferencia de los otros dos modelos iniciales. Finalmente si decidimos modificar los parámetros a $\alpha = 29$ y $\beta = 12$ obtenemos la siguiente simulación bajo las condiciones iniciales del ejemplo 1 de esta sección:

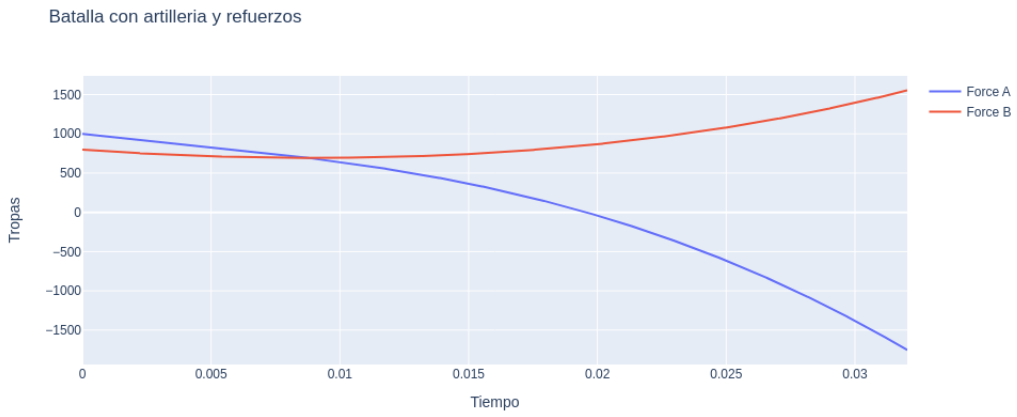

Figura 19: Modelo de artillería y refuerzos con $\alpha = 29$ y $\beta = 12$

Notemos que se balancean de forma mucho mas rápida la cantidad de tropas, sin embargo nuevamente las tropas de B resultan dominando a las de A. Ahora analicemos el retrato fase de estos parámetros modificados y la lista de condiciones iniciales que hemos manejado a lo largo de la práctica:

Figura 20: Retratos de fase del modelo con artillería y refuerzos con $\alpha = 29$ y $\beta = 12$, utilizando la misma lista de condiciones iniciales

Notemos que en estos casos se pronuncian mucho mas rápidamente las curvas del retrato a diferencia del ejemplo anterior sin embargo otra vez resultan dominantes las tropas de B

5. Discusión

Se presentaron algunos problemas a la hora de elegir los parámetros para probar los modelos, en particular el tiempo a veces resultaba demasiado grande que no se podía apreciar con claridad el comportamiento del modelo en el intervalo de interés, de igual manera a veces se presentaron advertencias de desbordamiento de memoria al trabajar con números ya fueran muy grandes o muy pequeños.

6. Conclusiones

Al realizar todos los ejemplos anteriores podemos concluir lo siguiente, el poder dar solución a modelos de sistemas de ecuaciones de diferenciales resulta en una herramienta increíblemente potente tanto computacionalmente como matemáticamente hablando dado que nos ayuda no solo a encontrar soluciones a dichos problemas sino comprender sobre la naturaleza de su comportamiento junto con otro tipo de apoyos tanto gráficos como analíticos, tal es el caso del retrato de fases.

7. Bibliografía

- Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., ... and Van Mulbregt, P. (2020). SciPy 1.0: fundamental algorithms for scientific computing in Python. Nature methods, 17(3), 261-272.
- Virtanen, P., Gommers, R., Oliphant, T. E., Burovski, E., Cournapeau, D., Weckesser, W., ... and Larson, E. (2020). Scipy/Scipy: Scipy 0.19. 0. Zenodo.
- Enright, W. H. (2000). Continuous numerical methods for ODEs with defect control. Journal of Computational and Applied Mathematics, 125(1-2), 159-170.
- Cao, J., Fussmann, G. F., Ramsay, J. O. (2008). Estimating a predator-prey dynamical model with the parameter cascades method. Biometrics, 64(3), 959-967.
- Coleman, C. S. (1983). Combat models. In Differential Equation Models (pp. 109-131). New York, NY: Springer New York.
- Tikhonov, A. N. (1952). Systems of differential equations containing small parameters in the derivatives. Mat. Sb.(NS), 31(73), 575-586.