Tarea 13: Métodos clásicos de solución de EDO's
Métodos Numéricos

Rafael Alejandro García Ramírez

19 de noviembre de 2023

1. Introducción

Para la física los modelos que puedan describir un sistema como función del tiempo o de cualquier otra u otras variables independientes es de vital importancia pues brinda una herramienta sencilla y poderosa al momento de describir sistemas completos de manera sencilla y directa. Se puede describir prácticamente cualquier sistema del mundo real mediante una ecuación diferencial o un sistema de ecuaciones diferenciales.

Muchas veces, aunque no sea evidente cómo ajustar un sistema a un conjunto de ecuaciones diferenciales, la facilidad con la que estas nos permiten realizar predicciones y a su vez entender el sistema hace que uno se vea casi en la necesidad de hacerlo. Para ello existen varias formas de resolver un sistema de ecuaciones diferencias una vez dado, así se puede tener datos teóricos que se puedan después ajustar a algún modelo en concreto de ser el caso.

2. Pseudocódigos

2.1. Método de Euler

Algorithm 1 Método de Euler

procedure EULER($N, a, b, y0, f$)
  $h \leftarrow (b-a)/N$
  $y \leftarrow$ vector tamaño $N+1$
  $y[0] \leftarrow y0$
  for $i \leftarrow 1$ **to** N **do**
    $y[i] \leftarrow y[i-1] + h \cdot f(a, y[i-1])$
    $a \leftarrow a + h$
  end for
  Devolver y
end procedure

2.2. Método de Heun

Algorithm 2 Método de Heun

procedure HEUN($N, a, b, y0, f$)
  $h \leftarrow (b-a)/N$
  $y \leftarrow$ vector tamaño $N+1$
  $y[0] \leftarrow y0$
  $a\_anterior \leftarrow a$
  for $i \leftarrow 1$ **to** N **do**
    $a \leftarrow a + h$
    $yDummy \leftarrow y[i-1] + h \cdot f(a\_anterior, y[i-1])$
    $y[i] \leftarrow y[i-1] + \frac{h}{2}(f(a\_anterior, y[i-1]) + f(a, yDummy))$
    $a\_anterior \leftarrow a$
  end for
  Devolver y
end procedure

2.3. Método de Taylor de Segundo Orden

Algorithm 3 Método de Taylor de Segundo Orden

procedure TaylorSegundoOrden($N, a, b, y0, f$)
  $h \leftarrow (b-a)/N$
  $y \leftarrow$ vector tamaño $N+1$
  derivada_x, derivada_y $\leftarrow 0$
  $h\_2 \leftarrow h^2/2$
  $y[0] \leftarrow y0$
  for $i \leftarrow 1$ **to** N **do**
    derivada_x $\leftarrow \partial_x f(a, y)$
    derivada_y $\leftarrow \partial_y f(a, y)$
    $y[i] \leftarrow y[i-1] + h \times f(a, y[i-1]) + h\_2 \times (\text{derivada\_x} + \text{derivada\_y} \cdot f(a, y[i-1]))$
    $a \leftarrow a + h$
  end for
  Devolver y
end procedure

2.4. Método de Runge - Kutta 4

Algorithm 4 Método de Runge-Kutta de Cuarto Orden

procedure RK4($N, a, b, y0, f$)
  $h \leftarrow (b-a)/N$
  $y \leftarrow$ vector tamaño $N+1$
  $k1, k2, k3, k4 \leftarrow 0$
  $y[0] \leftarrow y0$
  for $i \leftarrow 1$ **to** N **do**
    $k1 \leftarrow h \cdot f(a, y[i-1])$
    $k2 \leftarrow h \cdot f(a + \frac{h}{2}, y[i-1] + \frac{k1}{2})$
    $k3 \leftarrow h \cdot f(a + \frac{h}{2}, y[i-1] + \frac{k2}{2})$
    $k4 \leftarrow h \cdot f(a + h, y[i-1] + k3)$
    $y[i] \leftarrow y[i-1] + \frac{1}{6}(k1 + 2k2 + 2k3 + k4)$
    $a \leftarrow a + h$
  end for
  Devolver y
end procedure

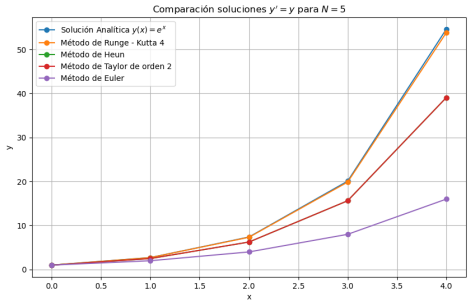
3. Resultados

1. Las funciones utilizadas para realizar los métodos son los siguientes

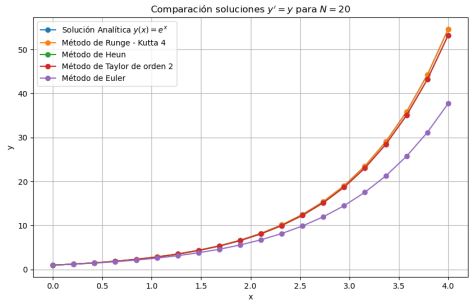
```c
#include <stdio.h>
#include <stdlib.h>

double *euler(int N, double a, double b, double y0, double (*f)(double, double))
{
    double h = (b - a) / N;
    double *y = malloc((N + 1) * sizeof(double));
    y[0] = y0;
    for (int i = 1; i <= N; i++)
    {
        y[i] = y[i - 1] + h * f(a, y[i - 1]);
        a += h;
    }
    return y;
}
double *heun(int N, double a, double b, double y0, double (*f)(double, double))
{
    double h = (b - a) / N;
    double *y = malloc((N + 1) * sizeof(double));
    double yDummy, a_anterior = a;
    y[0] = y0;
    for (int i = 1; i <= N; i++)
    {
        a += h;
```

```
        yDummy = y[i - 1] + h * f(a_anterior, y[i - 1]);
        y[i] = y[i - 1] + (h / 2) * (f(a_anterior, y[i - 1]) + f(a, yDummy));
        a_anterior = a;
    }
    return y;
}
double derivada_parcial(double (*f)(double, double), double x, double y, double h, int variable)
{
    if (variable == 0)
    {
        return (f(x + h, y) - f(x - h, y)) / (2 * h);
    }
    if (variable == 1)
    {
        return (f(x, y + h) - f(x, y - h)) / (2 * h);
    }
    else
    {
        printf("Error en la variable\n");
        return 0;
    }
}
double *taylor_segundo_orden(int N, double a, double b, double y0, double (*f)(double, double))
{
    double h = (b - a) / N;
    double *y = malloc((N + 1) * sizeof(double));
    double derivada_x, derivada_y;
    double h_2 = (h * h) / 2;
    y[0] = y0;
    for (int i = 1; i <= N; i++)
    {
        derivada_x = derivada_parcial(f, a, y[i - 1], h, 0);
        derivada_y = derivada_parcial(f, a, y[i - 1], h, 1);
        y[i] = y[i - 1] + h * f(a, y[i - 1]) + h_2 * (derivada_x + derivada_y * f(a, y[i - 1]));
        a += h;
    }

    return y;
}
double *RK4(int N, double a, double b, double y0, double (*f)(double, double))
{
    double h = (b - a) / N;
    double *y = malloc((N + 1) * sizeof(double));
    double k1, k2, k3, k4;
    y[0] = y0;
    for (int i = 1; i <= N; i++)
    {
        k1 = h * f(a, y[i - 1]);
        k2 = h * f(a + h / 2, y[i - 1] + k1 / 2);
        k3 = h * f(a + h / 2, y[i - 1] + k2 / 2);
        k4 = h * f(a + h, y[i - 1] + k3);
        y[i] = y[i - 1] + (1.0 / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4);
        a += h;
    }
    return y;
}
```

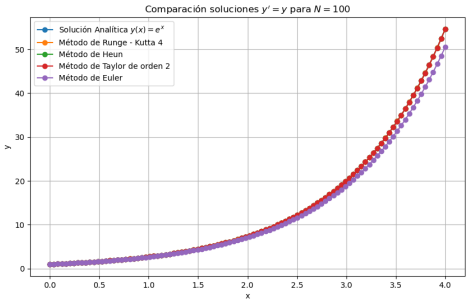
Con estas funciones se resolvió la ecuación diferencial $y' = y$ cuya solución es $y(x) = e^x$. Graficando las soluciones para $N = 4, 20, 100$



(a) $N = 4$

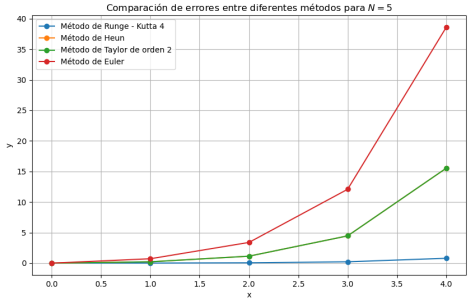


(b) $N = 20$

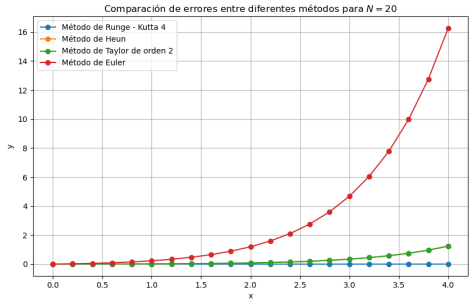


(c) $N = 100$

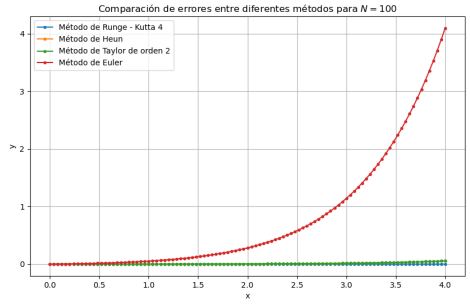
Podemos notar que para N pequeñas todos los modelos difieren en su exactitud a excepción del Runge - Kutta 4, aunque para N grandes el métoto de Euler proporciona el mayor error de todos, para poder ver esto más a detalle grafiquemos los datos con respecto a su valor real en cada punto



(a) $N = 4$



(b) $N = 20$



(c) $N = 100$

Como podemos ver, aunque se aumente el valor de N, el método de Euler genera un error que es acumulativo durante todo el intervalo.

2. a) Utilizando el método de la cuadratura gaussiana para 3 puntos tenemos que el valor de la integral es

$$I = \int_0^2 \sqrt{1 + t^3} dt \approx 3.241815612$$

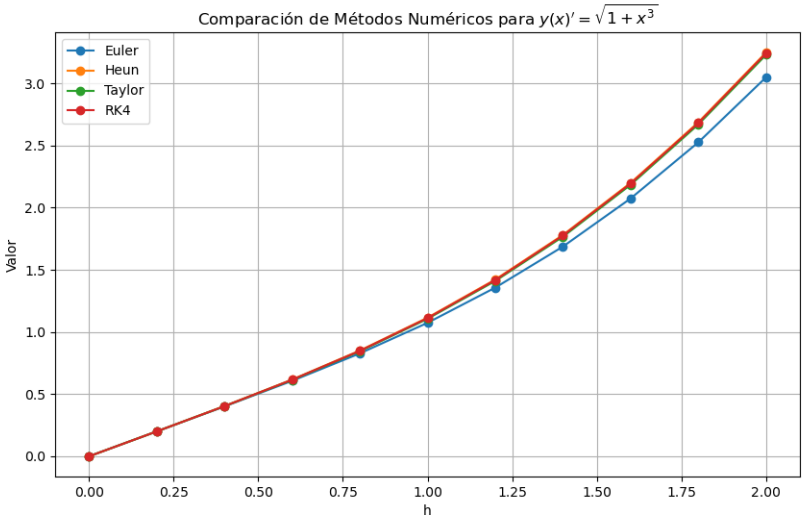
b) Los datos para la integral de cada método para cada paso de h se muestran en la siguiente tablas

h	Euler	Heun	Taylor	RK4
0.000000	0.000000	0.000000	0.000000	0.000000
0.200000	0.200000	0.200399	0.200400	0.200200
0.400000	0.400798	0.403949	0.402774	0.403171
0.600000	0.607099	0.617372	0.614011	0.615733
0.800000	0.827644	0.850607	0.844462	0.847996
1.000000	1.073571	1.114992	1.105964	1.111446
1.200000	1.356413	1.421580	1.409908	1.417210
1.400000	1.686747	1.780241	1.766277	1.775166
1.600000	2.073735	2.199478	2.183554	2.193798
1.800000	2.525221	2.686602	2.668984	2.680395
2.000000	3.047983	3.247983	3.228874	3.241307

Los errores absolutos con respecto al valor calculado con la cuadratura gaussiana de tres puntos son los siguientes

	Euler	Heun	Taylor	RK4
Error absoluto	0.193832	0.006168	0.012941	0.000508

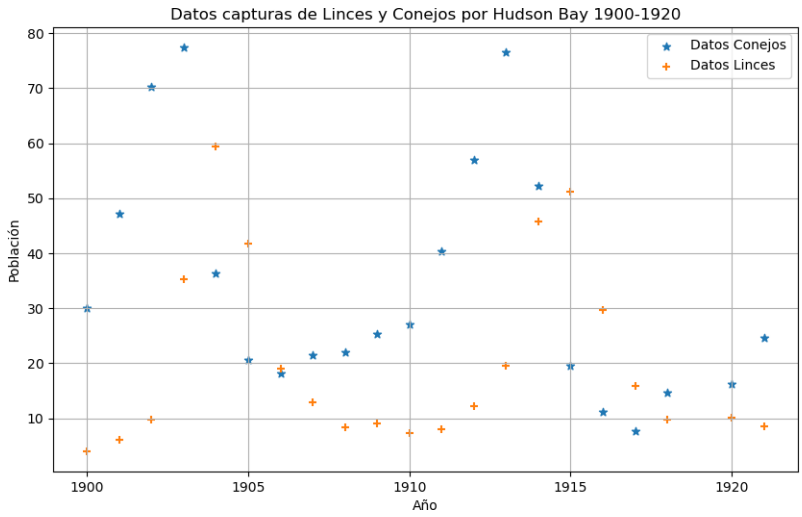
La siguiente gráfica nos puede ayudar a visualizar mejor los valores para $h = 0.2$ que se encontraban en la tabla



Para un $h = 1/2$ con $0 \leq x \leq 2$ con el método de Taylor de segundo orden tenemos los siguiente datos

i	$x + ih$	y
0	0.000000	0.000000
1	0.500000	0.515656
2	1.000000	1.097763
3	1.500000	1.933743
4	2.000000	3.177791

3. a) Graficando los datos de la tabla de Hudson Bay



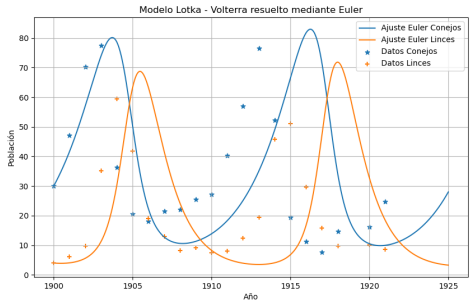
b) Las funciones mostradas anteriormente para resolver el sistema de ecuaciones se modificaron para resolver un sistema de ecuaciones

```c
#include <stdio.h>
#include <stdlib.h>

double *lotka_volterra(double t, double *xy)
{
    double *derivatives = malloc(2 * sizeof(double));
    derivatives[0] = 0.4 * xy[0] - 0.018 * xy[0] * xy[1];
    derivatives[1] = -0.8 * xy[1] + 0.023 * xy[0] * xy[1];
    return derivatives;
}

double *euler(int N, double a, double b, double *y0, int dim, double *(*f)(double, double *))
{
    double h = (b - a) / N;
    double *y = malloc((N + 1) * dim * sizeof(double));
    double *dy;


    for (int j = 0; j < dim; j++)
    {
        y[j] = y0[j];
    }

    for (int i = 1; i <= N; i++)
    {
        dy = f(a, &y[(i - 1) * dim]);
        for (int j = 0; j < dim; j++)
        {
            y[i * dim + j] = y[(i - 1) * dim + j] + h * dy[j];
        }
        a += h;
        free(dy);
    }
    return y;
}

double *heun(int N, double a, double b, double *y0, int dim, double *(*f)(double, double *))
{
    double h = (b - a) / N;
    double *y = malloc((N + 1) * dim * sizeof(double));
    double *yDummy = malloc(dim * sizeof(double));
    double *f1, *f2;


    for (int j = 0; j < dim; j++)
    {
        y[j] = y0[j];
    }

    for (int i = 1; i <= N; i++)
```

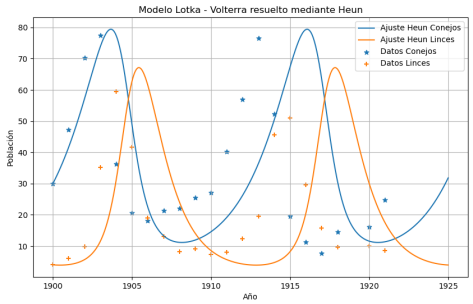
```
    {
        f1 = f(a, &y[(i - 1) * dim]);
        for (int j = 0; j < dim; j++)
        {
            yDummy[j] = y[(i - 1) * dim + j] + h * f1[j];
        }
        f2 = f(a + h, yDummy);

        for (int j = 0; j < dim; j++)
        {
            y[i * dim + j] = y[(i - 1) * dim + j] + (h / 2) * (f1[j] + f2[j]);
        }
        a += h;
        free(f1), free(f2);
    }

    free(yDummy);
    return y;
}

double *RK4(int N, double a, double b, double *y0, int dim, double *(*f)(double, double *))
{
    double h = (b - a) / N;
    double *y = malloc((N + 1) * dim * sizeof(double));
    double *k1, *k2, *k3, *k4, *yTmp;
    yTmp = malloc(dim * sizeof(double));


    for (int j = 0; j < dim; j++)
    {
        y[j] = y0[j];
    }

    for (int i = 1; i <= N; i++)
    {
        k1 = f(a, &y[(i - 1) * dim]);
        for (int j = 0; j < dim; j++)
        {
            yTmp[j] = y[(i - 1) * dim + j] + 0.5 * k1[j];
        }
        k2 = f(a + 0.5 * h, yTmp);

        for (int j = 0; j < dim; j++)
        {
            yTmp[j] = y[(i - 1) * dim + j] + 0.5 * k2[j];
        }
        k3 = f(a + 0.5 * h, yTmp);

        for (int j = 0; j < dim; j++)
        {
            yTmp[j] = y[(i - 1) * dim + j] + k3[j];
        }
        k4 = f(a + h, yTmp);

        for (int j = 0; j < dim; j++)
        {
            y[i * dim + j] = y[(i - 1) * dim + j] + (1.0 / 6.0) * (k1[j] + 2 * k2[j] + 2 * k3
    [j] + k4[j]);
        }

        a += h;
        free(k1);
        free(k2);
        free(k3);
        free(k4);
    }
    free(yTmp);
    return y;
}

double *derivada_parcial_x(double *(*f)(double, double *), double x, double *y, double h, int
     variable)
{
    double yPlus[2], yMinus[2], *fPlus, *fMinus, *derivatives;
    derivatives = malloc(2 * sizeof(double));

    yPlus[0] = y[0];
    yPlus[1] = y[1];
    yMinus[0] = y[0];
```

```
    yMinus[1] = y[1];

    if (variable == 0)
    {
        yPlus[0] += h;
        yMinus[0] -= h;
    }
    else if (variable == 1)
    {
        yPlus[1] += h;
        yMinus[1] -= h;
    }
    else
    {
        printf("Error en la variable\n");
        return NULL;
    }

    fPlus = f(x, yPlus);
    fMinus = f(x, yMinus);

    derivatives[0] = (fPlus[0] - fMinus[0]) / (2 * h);
    derivatives[1] = (fPlus[1] - fMinus[1]) / (2 * h);

    free(fPlus);
    free(fMinus);

    return derivatives;
}

double *taylor_segundo_orden(int N, double a, double b, double *y0, int dim, double *(*f)(
    double, double *))
{
    double h = (b - a) / N;
    double *y = malloc((N + 1) * dim * sizeof(double));
    double *derivada_x, *derivada_y, *fy;
    double h_2 = (h * h) / 2;


    for (int j = 0; j < dim; j++)
    {
        y[j] = y0[j];
    }

    for (int i = 1; i <= N; i++)
    {
        fy = f(a, &y[(i - 1) * dim]);
        derivada_x = derivada_parcial_x(f, a, &y[(i - 1) * dim], h, 0);
        derivada_y = derivada_parcial_x(f, a, &y[(i - 1) * dim], h, 1);

        for (int j = 0; j < dim; j++)
        {
            y[i * dim + j] = y[(i - 1) * dim + j] + h * fy[j] + h_2 * (derivada_x[j] +
    derivada_y[j] * fy[j]);
        }

        a += h;
        free(fy);
        free(derivada_x);
        free(derivada_y);
    }
    return y;
}
```

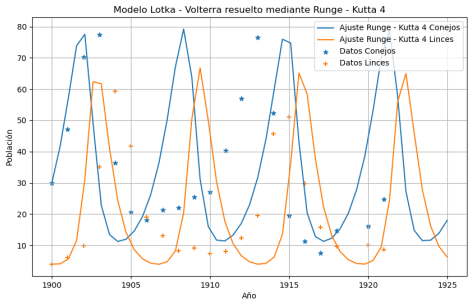
Con estas funciones se resolvieron el sistema de ecuaciones con los siguientes resultados puestos en gráficas



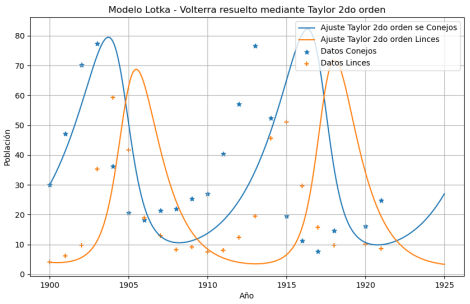
(a) Método de Euler, $N = 1000$



(b) Método de Heun, $N = 1000$



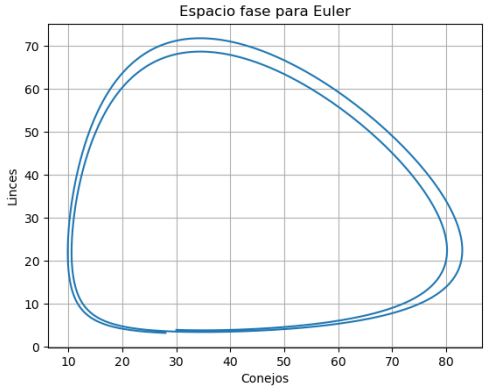
(c) Método de Runge - Kutta 4, $N = 48$



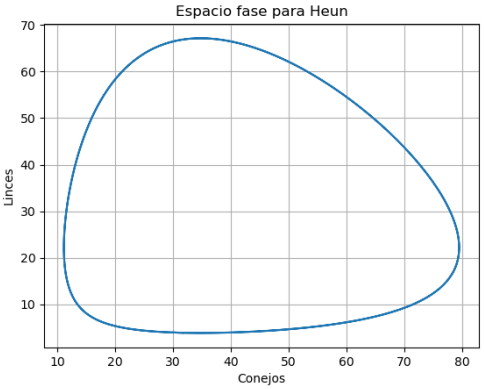
(d) Método de Taylor de 2do orden, $N = 1000$

Cabe resaltar que para el método de Runge - Kutta 4 al hacer N mayor la solución tiende a ajustarse mucho más a los datos que tenemos pero con una mayor oscilación.

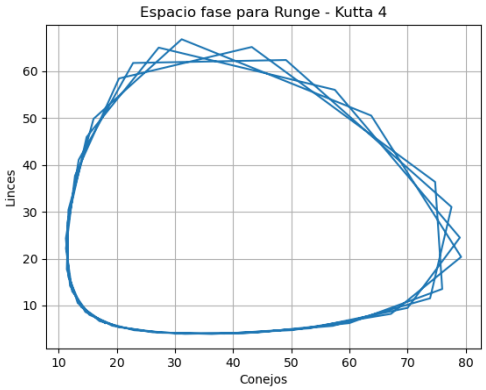
c) El espacio fase de cada solución se puede ver a continuación



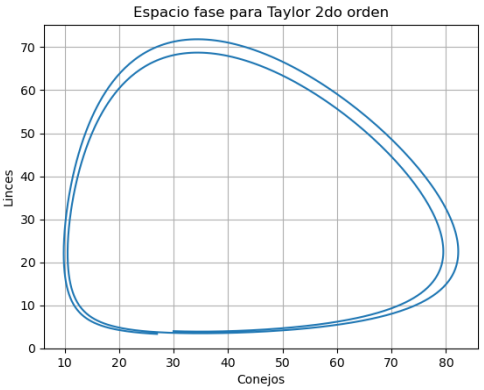
(a) Espacio fase Euler, $N = 1000$



(b) Espacio fase Heun, $N = 1000$



(c) Espacio fase Runge - Kutta 4, $N = 48$



(d) Espacio fase Taylor de 2do orden, $N = 1000$

4. Conclusiones

Dentro de los métodos para resolver ecuaciones diferenciales podemos notar que existen aplicaciones variadas a estas, desde comparar el ajuste de datos al modelo teórico y viceversa, hasta la realización de poblemas simples como el cálculo de integrales. Notamos a su vez que para conseguir modelos suaves existen diferentes aproximación dependiendo del método; el método de Runge - Kutta 4 obtiene resultados muy buenos en apenas unas cuantas iteraciones para ecuaciones diferenciales sencillas, pero para el caso del sistema Lotka-Volterra el método presentó fuertes oscilaciones para un numéro N de pasos más grande.

Resulta necesario, pues, tener bien en claro las motivaciones y necesidades de estudio del sistema al momento de escoger el método: por ejemplo, si se desea estudiar un sistema dentro de un rango pequeño y sin la necesidad de mucha precisión el método de Euler es la mejor alternativa, mientras que si se desea estudiar un sistema sin muchas oscilaciones y necesitando una mayor precisión el método Runge - Kutta 4 resulta la mejor opción. Otros métodos como los de Heun y Taylor de segundo orden son muy buenas alternativas cuando se desconoce las propiedades del sistema (o si estas son muy específicas) y se requiera de un exactitud mayor.

5. Referencias

1. Arévalo Ovalle, D., Bernal Yermanos, M. Á., & Posada Restrepo, J. A. (2021). Métodos numéricos con Python.
2. Kong, Q., Siauw, T., & Bayen, A. (2020). Python programming and numerical methods: A guide for engineers and scientists. Academic Press.
3. Richard. L. Burden y J. Douglas Faires, Análisis Numérico, 7a Edición, Editorial Thomson Learning, 2002.
4. Samuel S M Wong, Computational Methods in Physics and Enginering, Ed. World Scientific, 3rd Edition, 1997.
5. Teukolsky, S. A., Flannery, B. P., Press, W. H., & Vetterling, W. T. (1992). Numerical recipes in C. SMR, 693(1), 59-70.
6. William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes: The Art of Scientific Computing, 3rd Edition, Cambridge University Press, 2007