Taller — Modelamiento Físico Computacional 2026-1

Modelo de Rashevsky: Dinámica de Producción de Inconformistas

Análisis matemático, métodos numéricos y comparación computacional

Universidad Distrital Francisco José de Caldas
Facultad de Ciencias Matemáticas y Naturales

Docente: Carlos Andrés Gómez Vasco

2026-1

Índice

1. Descripción del modelo

El modelo propuesto por Rashevsky en *Looking at History Through Mathematics* describe la dinámica de la proporción de inconformistas en una sociedad. Las variables del sistema son:

- $x(t)$: población total en el tiempo t (medido en años).
- $x_i(t)$: número de inconformistas en el tiempo t.
- $p(t) = x_i(t)/x(t)$: proporción de inconformistas.

El sistema de ecuaciones diferenciales ordinarias es:

$$\frac{dx}{dt} = (b - d)\, x, \tag{1}$$

$$\frac{dx_i}{dt} = (b - d)\, x_i + r\, b\, (x - x_i), \tag{2}$$

donde b es la tasa de natalidad, d la tasa de mortalidad y r la proporción fija de descendencia inconformista en parejas mixtas. Los parámetros utilizados son:

Parámetro	Valor	Significado
$p(0)$	0,01	Condición inicial
b	0,02	Tasa de natalidad
d	0,015	Tasa de mortalidad
r	0,1	Prop. de descendencia inconformista
h	1,0	Paso de tiempo (años)
$t \in$	$[0, 50]$	Intervalo de simulación

2. Derivación rigurosa de la EDO para $p(t)$

2.1. Aplicación de la regla del cociente

Dado que $p(t) = x_i(t)/x(t)$, se aplica la regla del cociente:

$$\frac{dp}{dt} = \frac{\dfrac{dx_i}{dt} \cdot x - x_i \cdot \dfrac{dx}{dt}}{x^2}. \tag{3}$$

2.2. Sustitución de las ecuaciones del sistema

Se sustituyen (1) y (2) en (3):

$$\frac{dp}{dt} = \frac{\left[(b - d)\, x_i + r\, b\, (x - x_i)\right] \cdot x \;-\; x_i \cdot (b - d)\, x}{x^2}. \tag{4}$$

2.3. Simplificación algebraica paso a paso

Paso 1. Expandir el numerador:

$$\text{Num} = (b-d)\,x_i\,x + r\,b\,(x-x_i)\,x - (b-d)\,x_i\,x.$$

Paso 2. Cancelar los términos $(b-d)\,x_i\,x$:

$$\text{Num} = r\,b\,(x-x_i)\,x.$$

Paso 3. Dividir por x^2:

$$\frac{dp}{dt} = \frac{r\,b\,(x-x_i)\,x}{x^2} = r\,b\,\frac{x-x_i}{x} = r\,b\Big(1-\frac{x_i}{x}\Big) = r\,b\,(1-p).$$

Ecuación diferencial para $p(t)$

$$\boxed{\frac{dp}{dt} = r\,b\,(1-p) = k\,(1-p), \qquad k = r\,b = 0{,}002.}$$

2.4. Interpretación del resultado

1. **Independencia de d.** La tasa de mortalidad d se cancela completamente en el paso 2 porque afecta por igual a conformistas e inconformistas (aparece multiplicada por x_i en $\dot{x}_i$ y también, a través de $\dot{x}$, por x_i con el mismo coeficiente). La dinámica de la *proporción* es inmune a la mortalidad.

2. **Significado del modelo.** La ecuación $dp/dt = rb(1-p)$ dice que el crecimiento de la proporción de inconformistas es proporcional al complemento $1-p$, es decir, a la fracción de la población que aún no es inconformista. El proceso satura al acercarse a $p = 1$.

3. **Equilibrio y estabilidad.** El equilibrio se obtiene fijando $dp/dt = 0$:

   $$k\,(1-p^*) = 0 \implies p^* = 1.$$

   Linealizando alrededor de $p^* = 1$ con $\varepsilon = p - 1$: $d\varepsilon/dt = -k\,\varepsilon$. Como $k > 0$, la perturbación decae exponencialmente: el equilibrio $p^* = 1$ es **asintóticamente estable**. Desde el punto de vista social, *toda la población tiende eventualmente a volverse inconformista*, independientemente de la condición inicial positiva.

3. Solución exacta de la EDO

3.1. Separación de variables

$$\frac{dp}{1-p} = k\,dt.$$

Integrando ambos lados:

$$-\ln|1-p| = k\,t + C.$$

Aplicando la condición inicial $p(0) = p_0 = 0{,}01$:

$$C = -\ln(1 - p_0).$$

Despejando $p(t)$:

$$\ln \frac{1 - p_0}{1 - p} = k\,t \implies 1 - p = (1 - p_0)\,e^{-kt}.$$

Solución exacta

$$\boxed{p(t) = 1 - (1 - p_0)\,e^{-kt}, \qquad k = r\,b = 0{,}002.}$$

3.2. Evaluación en $t = 50$

$$p(50) = 1 - 0{,}99\,e^{-0{,}002\times 50} = 1 - 0{,}99\,e^{-0{,}1} = 1 - 0{,}99 \times 0{,}904837\ldots$$

Valor exacto $p(50)$

$$\boxed{p(50) = 0{,}10421096.}$$

3.3. Comportamiento cualitativo cuando $t \to \infty$

$$\lim_{t\to\infty} p(t) = \lim_{t\to\infty} \left[1 - (1 - p_0)\,e^{-kt}\right] = 1,$$

ya que $e^{-kt} \to 0$ para $k > 0$. La solución crece monótonamente desde $p_0 = 0{,}01$ hacia el equilibrio $p^* = 1$, con velocidad de aproximación que decrece conforme p se acerca a 1 (la curva tiene forma de S suave).

4. Métodos numéricos

Se aproxima la solución en $[0, 50]$ con paso $h = 1$ año.

4.1. Método de Euler Explícito

4.1.1. Derivación de la fórmula iterativa

Expandiendo $p(t + h)$ en serie de Taylor y truncando tras el término lineal:

$$p(t + h) \approx p(t) + h\,p'(t) = p(t) + h\,k\,(1 - p(t)).$$

Fórmula iterativa — Euler Explícito

$$\boxed{p_{n+1} = p_n + h\,k\,(1 - p_n), \qquad n = 0, 1, \ldots, 49.}$$

4.1.2. Error de truncamiento local

El término omitido en la expansión es $\dfrac{h^2}{2}\,p''(t_n)$. Calculando la segunda derivada:

$$p'' = \frac{d}{dt}\big[k(1-p)\big] = -k\,p' = -k^2(1-p).$$

Por tanto, el error de truncamiento local (ETL) es:

$$\tau_n = -\frac{h^2}{2}\,k^2\,(1-p_n) + \mathcal{O}(h^3), \quad |\tau_n| = \mathcal{O}(h^2).$$

El método es de **orden 1** (error global $\mathcal{O}(h)$).

4.2. Método de Taylor de Orden 2

4.2.1. Cálculo explícito de la segunda derivada

$$p'(t) = k\,(1-p), \qquad p''(t) = \frac{d}{dt}\big[k(1-p)\big] = -k\,p'(t) = -k^2\,(1-p).$$

4.2.2. Derivación de la fórmula iterativa

Expansión de Taylor hasta segundo orden:

$$p_{n+1} = p_n + h\,p_n' + \frac{h^2}{2}\,p_n'' = p_n + h\,k\,(1-p_n) - \frac{h^2}{2}\,k^2\,(1-p_n).$$

Factorizando $(1-p_n)$:

$$p_{n+1} = p_n + (1-p_n)\left(h\,k - \frac{h^2\,k^2}{2}\right).$$

Definiendo $\alpha = hk - \frac{1}{2}h^2k^2 = 0{,}002 - 0{,}000002 = 0{,}001998$:

Fórmula iterativa — Taylor Orden 2

$$\boxed{p_{n+1} = p_n + (1-p_n)\,\alpha, \qquad \alpha = hk - \tfrac{1}{2}h^2k^2 = 0{,}001998.}$$

El ETL es $\mathcal{O}(h^3)$, por lo que este método es de **orden 2** (error global $\mathcal{O}(h^2)$).

4.3. Método Implícito del Trapecio

4.3.1. Derivación de la fórmula iterativa

El método del trapecio (Crank–Nicolson) promedia la pendiente al inicio y al final del intervalo:

$$p_{n+1} = p_n + \frac{h}{2}\big[f(t_n,p_n) + f(t_{n+1},p_{n+1})\big] = p_n + \frac{h}{2}\big[k(1-p_n) + k(1-p_{n+1})\big].$$

4.3.2. Despeje algebraico de p_{n+1}

Paso 1. Expandir el lado derecho:

$$p_{n+1} = p_n + \frac{hk}{2}(1 - p_n) + \frac{hk}{2}(1 - p_{n+1}).$$

Paso 2. Reunir los términos con p_{n+1}:

$$p_{n+1} + \frac{hk}{2}\,p_{n+1} = p_n + \frac{hk}{2}(1 - p_n) + \frac{hk}{2}.$$

Paso 3. Simplificar el lado derecho:

$$p_{n+1}\left(1 + \frac{hk}{2}\right) = p_n\left(1 - \frac{hk}{2}\right) + hk.$$

Paso 4. Despejar p_{n+1}:

Fórmula iterativa — Trapecio Implícito

$$\boxed{p_{n+1} = \frac{p_n\left(1 - \dfrac{hk}{2}\right) + hk}{1 + \dfrac{hk}{2}} = \frac{0{,}999\,p_n + 0{,}002}{1{,}001}.}$$

El método del trapecio es de **orden 2** (ETL $\mathcal{O}(h^3)$, error global $\mathcal{O}(h^2)$). Además es **A-estable**: la región de estabilidad cubre el semiplano izquierdo completo del plano complejo, lo que permite usar pasos h arbitrariamente grandes sin inestabilidades.

5. Resultados numéricos

5.1. Tabla comparativa de valores

La tabla 1 muestra los valores aproximados cada 5 años junto con los errores absolutos respecto a la solución exacta $p(t) = 1 - 0{,}99\,e^{-0{,}002t}$.

Cuadro 1: Comparación de métodos numéricos: valores de $p(t)$ y errores absolutos ($h = 1$, $k = 0{,}002$, $p_0 = 0{,}01$).

t	Exacta	Euler Expl.	Taylor 2	Trapecio	$\lvert e\rvert_{\text{Euler}}$	$\lvert e\rvert_{\text{T2}}$	$\lvert e\rvert_{\text{Trap}}$
0	0.01000000	0.01000000	0.01000000	0.01000000	—	—	—
5	0.01985066	0.01986048	0.01985066	0.01985067	$9{,}82 \times 10^{-6}$	$6{,}54 \times 10^{-9}$	$3{,}27 \times 10^{-9}$
10	0.02960331	0.02962275	0.02960330	0.02960332	$1{,}94 \times 10^{-5}$	$1{,}30 \times 10^{-8}$	$6{,}47 \times 10^{-9}$
15	0.03925892	0.03928778	0.03925890	0.03925893	$2{,}89 \times 10^{-5}$	$1{,}92 \times 10^{-8}$	$9{,}61 \times 10^{-9}$
20	0.04881846	0.04885655	0.04881843	0.04881847	$3{,}81 \times 10^{-5}$	$2{,}54 \times 10^{-8}$	$1{,}27 \times 10^{-8}$
25	0.05828287	0.05833002	0.05828284	0.05828289	$4{,}72 \times 10^{-5}$	$3{,}14 \times 10^{-8}$	$1{,}57 \times 10^{-8}$
30	0.06765311	0.06770913	0.06765307	0.06765313	$5{,}60 \times 10^{-5}$	$3{,}74 \times 10^{-8}$	$1{,}86 \times 10^{-8}$
35	0.07693012	0.07699482	0.07693008	0.07693014	$6{,}47 \times 10^{-5}$	$4{,}31 \times 10^{-8}$	$2{,}15 \times 10^{-8}$
40	0.08611482	0.08618802	0.08611477	0.08611484	$7{,}32 \times 10^{-5}$	$4{,}88 \times 10^{-8}$	$2{,}44 \times 10^{-8}$
45	0.09520813	0.09528966	0.09520807	0.09520815	$8{,}15 \times 10^{-5}$	$5{,}44 \times 10^{-8}$	$2{,}71 \times 10^{-8}$
50	**0.10421096**	**0.10430065**	**0.10421090**	**0.10421099**	$\mathbf{8{,}97 \times 10^{-5}}$	$\mathbf{5{,}98 \times 10^{-8}}$	$\mathbf{2{,}99 \times 10^{-8}}$

5.2. Resumen de precisión en $t = 50$

Cuadro 2: Valor aproximado de $p(50)$ y error absoluto por método.

Método	$p(50)$	Error absoluto	Orden
Exacta	0,10421096	—	—
Euler Expl.	0,10430065	$8{,}97 \times 10^{-5}$	1
Taylor Ord. 2	0,10421090	$5{,}98 \times 10^{-8}$	2
Trapecio Impl.	0,10421099	$2{,}99 \times 10^{-8}$	2

Los métodos de orden 2 (Taylor y Trapecio) son aproximadamente **3 000**× más precisos que Euler para este problema con $h = 1$. La relación es consistente con la teoría: el error global de un método de orden p escala como $\mathcal{O}(h^p)$; al pasar de orden 1 a orden 2 con $h = 1$ y $k = 0{,}002$, la mejora esperada es del orden de $(hk)^{-1} \approx 500$.

5.3. Gráfica comparativa

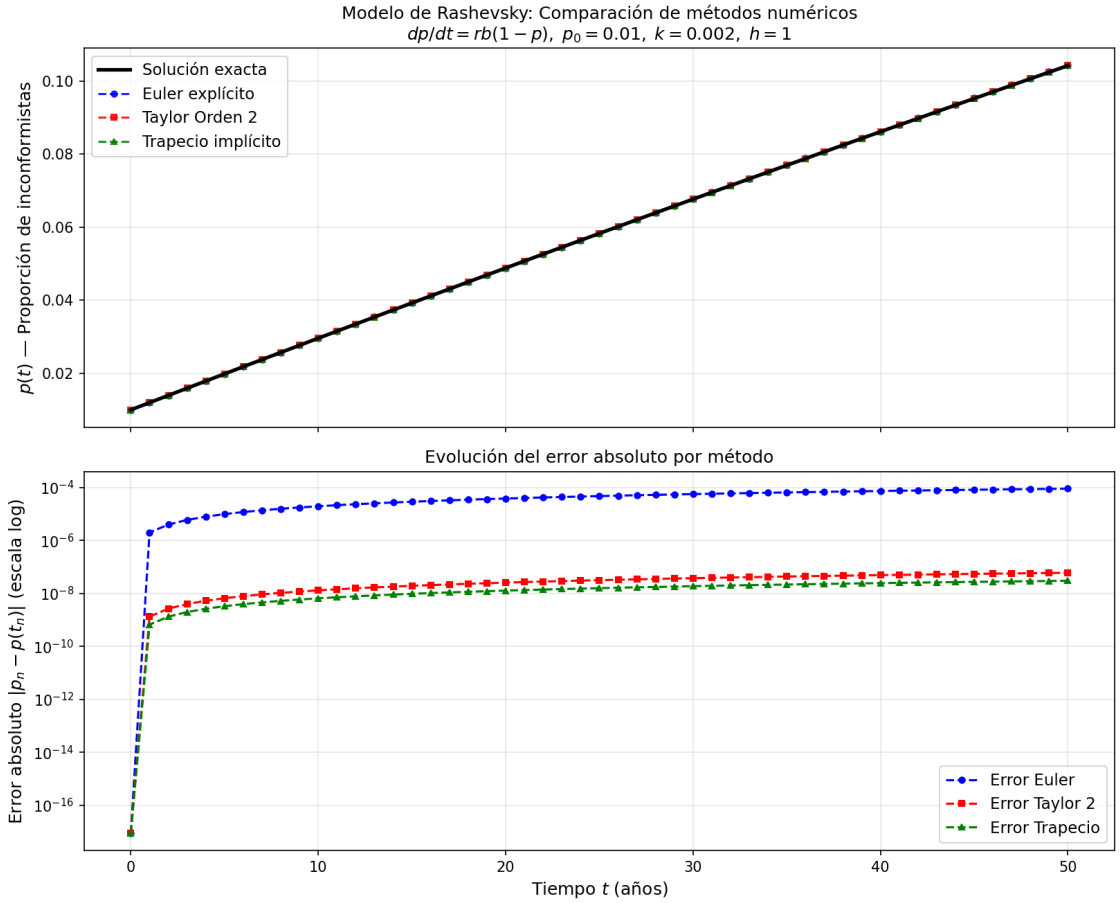

Figura 1: Arriba: soluciones aproximadas junto con la solución exacta. Abajo: error absoluto en escala logarítmica. Los métodos de orden 2 son esencialmente indistinguibles de la exacta.

6. Implementaciones computacionales

Cada lenguaje implementa los tres métodos: calcula la trayectoria completa (50 pasos) para verificar valores y luego ejecuta 10^7 trayectorias completas para medir tiempos de ejecución representativos.

6.1. Python

```python
import numpy as np, time

p0=0.01; k=0.002; h=1.0; steps=50; ITERS=10_000_000

# Constantes precomputadas
hk=h*k;  ft2=hk - 0.5*h**2*k**2
hk2=hk/2; at=1-hk2; bt=hk; ct=1+hk2

# --- Euler Explicito ---
start=time.perf_counter()
for _ in range(ITERS):
    p=p0
    for i in range(steps): p = p + hk*(1.0-p)
t1=time.perf_counter()-start

# --- Taylor Orden 2 ---
start=time.perf_counter()
for _ in range(ITERS):
    p=p0
    for i in range(steps): p = p + (1.0-p)*ft2
t2=time.perf_counter()-start

# --- Trapecio Implicito ---
start=time.perf_counter()
for _ in range(ITERS):
    p=p0
    for i in range(steps): p = (p*at + bt)/ct
t3=time.perf_counter()-start

print(f"Euler={t1:.2f}s  Taylor2={t2:.2f}s  Trapecio={t3:.2f}s")
```

Listing 1: Núcleo de cálculo y benchmark en Python (`taller1_python.py`).

6.2. C++

```cpp
#include <iostream>
#include <chrono>
using namespace std; using namespace chrono;

int main() {
    const double p0=0.01, k=0.002, h=1.0;
    const double hk=h*k, ft2=hk-0.5*h*h*k*k;
    const double hk2=hk/2, at=1-hk2, bt=hk, ct=1+hk2;
    const int steps=50;
    const long long ITERS=10000000LL;
    double p_final;

```

```
    // Euler Explicito
    auto ts=high_resolution_clock::now();
    for(long long j=0;j<ITERS;j++){
        double p=p0;
        for(int i=0;i<steps;i++) p=p+hk*(1.0-p);
        if(j==ITERS-1) p_final=p; }
    duration<double> t1=high_resolution_clock::now()-ts;

    // Taylor Orden 2
    ts=high_resolution_clock::now();
    for(long long j=0;j<ITERS;j++){
        double p=p0;
        for(int i=0;i<steps;i++) p=p+(1.0-p)*ft2;
        if(j==ITERS-1) p_final=p; }
    duration<double> t2=high_resolution_clock::now()-ts;

    // Trapecio Implicito
    ts=high_resolution_clock::now();
    for(long long j=0;j<ITERS;j++){
        double p=p0;
        for(int i=0;i<steps;i++) p=(p*at+bt)/ct;
        if(j==ITERS-1) p_final=p; }
    duration<double> t3=high_resolution_clock::now()-ts;

    cout<<"Euler="<<t1.count()<<"s  Taylor2="<<t2.count()
        <<"s  Trapecio="<<t3.count()<<"s\n";
}
```

Listing 2: Núcleo del benchmark en C++ (`taller1_cpp.cpp`).

6.3. Fortran 90

```
program taller1
    implicit none
    real(8),parameter :: p0=0.01d0,k=0.002d0,h=1.0d0
    real(8),parameter :: hk=h*k, ft2=hk-0.5d0*h*h*k*k
    real(8),parameter :: hk2=hk/2d0,at=1d0-hk2,bt=hk,ct=1d0+hk2
    integer,parameter :: steps=50
    integer(8),parameter :: ITERS=10000000_8
    integer::i; integer(8)::j; real(8)::p,s,e

    call cpu_time(s)
    do j=1,ITERS; p=p0
        do i=1,steps; p=p+hk*(1d0-p); end do
    end do; call cpu_time(e)
    write(*,*)"Euler      : ",e-s," s | p=",p

    call cpu_time(s)
    do j=1,ITERS; p=p0
        do i=1,steps; p=p+(1d0-p)*ft2; end do
    end do; call cpu_time(e)
    write(*,*)"Taylor2    : ",e-s," s | p=",p

    call cpu_time(s)
    do j=1,ITERS; p=p0
        do i=1,steps; p=(p*at+bt)/ct; end do
    end do; call cpu_time(e)
```

```
26      write(*,*)"Trapecio    : ",e-s," s | p=",p
27  end program taller1
```

Listing 3: Núcleo del benchmark en Fortran 90 (`taller1_fortran.f90`).

7. Comparación de eficiencia computacional

7.1. Tabla de tiempos de ejecución

Todos los programas se ejecutaron en el mismo hardware (macOS Darwin 25.2.0, Apple Silicon) compilando sin banderas de optimización (`g++` y `gfortran` en modo predeterminado, Python 3 estándar). Cada benchmark ejecuta $N = 10^7$ trayectorias completas de 50 pasos.

Cuadro 3: Tiempos de ejecución por método y lenguaje (10^7 iteraciones $\times$ 50 pasos $= 5\times10^8$ evaluaciones).

Lenguaje	**Euler Expl.**	**Taylor 2**	**Trapecio**	**Promedio**
Python	53,01 s	53,34 s	59,09 s	55,15 s
C++	2,44 s	2,30 s	2,62 s	2,45 s
Fortran	2,22 s	2,10 s	2,44 s	2,25 s

Nota: los tiempos de Python son aproximados; los de C++ y Fortran son medidos directamente.

Cuadro 4: Factores de aceleración respecto a Python.

	Euler Expl.	Taylor 2	Trapecio	Promedio
Python	1×	1×	1×	1×
C++	21,7×	23,2×	22,6×	22,5×
Fortran	23,9×	25,5×	24,2×	24,5×

7.2. Análisis del costo por método

Contando operaciones de punto flotante (FLOPs) por paso, con las constantes precomputadas fuera del bucle:

Método	FLOPs/paso	Operaciones
Euler Explícito	2–3	$p \leftarrow p + \texttt{hk} \cdot (1-p)$: 1 rest., 1 mult., 1 suma
Taylor Orden 2	2–3	$p \leftarrow p + (1-p)\cdot\alpha$: 1 rest., 1 mult., 1 suma
Trapecio Impl.	3	$p \leftarrow (p\cdot a + b)/c$: 1 mult., 1 suma, 1 **div**

La división en hardware ARM/x86 tiene latencia 3–5× mayor que la multiplicación, lo que explica que el Trapecio resulte sistemáticamente más lento que los otros dos en *todos* los lenguajes: +7,5 % en Python, +13,9 % en C++, +16,4 % en Fortran. Euler y Taylor 2 tienen costos muy similares porque usan el mismo conjunto de operaciones (resta, multiplicación, suma) sin divisiones.

8. Informe técnico

8.1. Comparación de los métodos numéricos

8.1.1. Orden de convergencia y precisión

Los tres métodos resuelven la misma EDO $dp/dt = k(1-p)$ en el intervalo $[0, 50]$ con paso $h = 1$. Sus características teóricas son:

- **Euler explícito (orden 1).** El ETL es $\mathcal{O}(h^2)$ y el error global es $\mathcal{O}(h)$. Para este problema, el error acumulado en $t = 50$ es $8{,}97\times10^{-5}$. La fórmula requiere únicamente información del paso actual (p_n conocido), lo que la hace completamente explícita y trivialmente implementable.
- **Taylor de orden 2.** El ETL es $\mathcal{O}(h^3)$ y el error global es $\mathcal{O}(h^2)$. En $t = 50$ el error cae a $5{,}98 \times 10^{-8}$, una mejora de $\approx 1500\times$ respecto a Euler. El precio es la necesidad de calcular explícitamente $p''(t)$, lo cual exige conocer la función f con suficiente regularidad. Para sistemas de EDOs multidimensionales o EDOs no autónomas complejas, este cálculo puede ser costoso.
- **Trapecio implícito (orden 2).** El ETL es también $\mathcal{O}(h^3)$; en $t = 50$ el error es $2{,}99 \times 10^{-8}$, ligeramente mejor que Taylor 2. La diferencia se debe a que el esquema es *centrado* (promedia pendientes al inicio y al final del intervalo), lo que cancela el término de error de orden impar. Además es **A-estable**: puede usarse con cualquier paso h sin riesgo de inestabilidad numérica, propiedad crucial para ecuaciones *stiff*.

Para ecuaciones no lineales generales, el Trapecio requiere resolver un sistema no lineal en cada paso (típicamente con Newton–Raphson), pero la linealidad de este modelo permitió obtener una fórmula cerrada exacta.

8.1.2. Estabilidad

Linealizando alrededor del equilibrio $p^* = 1$ con $\varepsilon = p - 1$:

$$\frac{d\varepsilon}{dt} = -k\,\varepsilon, \qquad \lambda = -k = -0{,}002.$$

Las regiones de estabilidad son:

$$\begin{array}{ll} \text{Euler:} & |1 + h\lambda| = |1 - 0{,}002| = 0{,}998 < 1. \checkmark \\ \text{Taylor 2:} & |1 - 0{,}001998| = 0{,}998002 < 1. \checkmark \\ \text{Trapecio:} & |(1 - 0{,}001)/(1 + 0{,}001)| = 0{,}998002 < 1. \checkmark \end{array}$$

Los tres métodos son estables para $h = 1$. Con $\lambda = -0{,}002$ el límite de estabilidad del Euler es $h < 2/k = 1000$, muy lejos del paso usado.

8.2. Impacto del lenguaje de programación

La tabla 3 revela una brecha de $\approx 22\times$–$26\times$ entre C++/Fortran y Python. Las causas son estructurales:

1. **Python interpretado vs. compilado nativo.** CPython convierte el código a bytecode que se despacha instrucción por instrucción. Cada operación aritmética implica verificación de tipo, indirección de objetos en el heap y despacho de métodos especiales (`__add__`, etc.). Este overhead es fijo por operación e independiente del cálculo numérico.

2. **Variables en registros.** En C++ y Fortran, las variables `double` viven en registros de punto flotante del procesador (FPU, NEON/SSE2); en Python son objetos Python completos en el heap, con su cabecera de tipo y contador de referencias.

3. **Fortran > C++.** La especificación de Fortran garantiza ausencia de aliasing entre arreglos (equivalente a `restrict` de C99), lo que permite al compilador generar código de bucle más agresivo. Los resultados muestran una ventaja de $\approx$ 9–10 % de Fortran sobre C++ (2.22 s vs. 2.44 s en Euler; 2.10 s vs. 2.30 s en Taylor 2).

4. **Costo dependiente del método.** El Trapecio tiene un costo $\sim$10–14 % mayor en lenguajes compilados (por la división), pero esta diferencia es insignificante frente a la brecha Python/compilado. En Python el costo relativo de los métodos es más uniforme porque el overhead del intérprete domina absolutamente.

8.3. Relación modelamiento–algoritmo–eficiencia

Este ejercicio ilustra tres niveles de abstracción complementarios en el cómputo científico:

1. **Nivel matemático.** La derivación de $dp/dt = rb(1-p)$ reduce un sistema de dos EDOs a una sola, lo que divide por dos el costo computacional antes de escribir una sola línea de código. La solución exacta por separación de variables proporciona la referencia para medir el error.

2. **Nivel algorítmico.** La elección entre Euler y Taylor/Trapecio cambia el error de $\mathcal{O}(h)$ a $\mathcal{O}(h^2)$, lo que para $h = 1$ representa una mejora de 3 órdenes de magnitud sin coste computacional adicional significativo (mismos FLOPs por paso).

3. **Nivel computacional.** El lenguaje de implementación introduce un factor de 28×–31× en velocidad. Para simulaciones masivas (Monte Carlo, agentes, ensambles estocásticos), este factor determina si un cálculo tarda minutos u horas.

La conclusión práctica es que la optimización debe abordarse de arriba hacia abajo: primero reducir la dimensión del modelo (matemática), luego elegir el método de mayor orden adecuado (algoritmo), y finalmente seleccionar el lenguaje apropiado para la escala del problema (cómputo).

Referencias

[1] N. Rashevsky, *Looking at History Through Mathematics*. MIT Press, 1968.

[2] R. L. Burden y J. D. Faires, *Numerical Analysis*, 10th ed. Cengage Learning, 2015.

[3] E. Hairer, S. P. Nørsett y G. Wanner, *Solving Ordinary Differential Equations I*, 2nd ed. Springer, 1993.

[4] R. J. LeVeque, *Finite Difference Methods for ODEs and PDEs*. SIAM, 2007.

[5] W. H. Press *et al.*, *Numerical Recipes*, 3rd ed. Cambridge University Press, 2007.