Modelamiento de dinámica de producción de inconformistas en una sociedad

Isabel Nieto, Leonardo Tovar

Universidad Distrital Francisco José de Caldas
Modelamiento Fisico Computacional 2026

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

1. Solución analítica del problema

- Población total $x(t)$
- $x_i(t)$ representa el número de individuos inconformistas
- proporción fija r del resto de la descendencia
- b tasas constante de nacimiento
- d tasas constante de muerte

Sistema de ecuaciones diferenciales que modela la dinámica del problema

$$\begin{cases} \dfrac{dx}{dt} = (b-d)x \\ \dfrac{dx_i}{dt} = (b-d)x_i + rb(x-x_i) \end{cases} \tag{1}$$

Se define la proporción de inconformistas como

$$p(t) = \frac{x_i(t)}{x(t)} \tag{2}$$

Aplicamos la regla de la derivada del cociente:

$$\frac{dp}{dt} = \frac{\frac{dx_i}{dt}x - x_i\frac{dx}{dt}}{x^2}$$

Sustituyendo las expresiones de las derivadas del sistema original:

$$\frac{dp}{dt} = \frac{\left[(b-d)x_i + rb(x-x_i)\right]x - x_i\left[(b-d)x\right]}{x^2}$$

Al distribuir el término x en el numerador, observamos que los términos que contienen la tasa neta $(b-d)$ se cancelan:

$$\frac{dp}{dt} = \frac{(b-d)x_i x + rb(x-x_i)x - (b-d)x_i x}{x^2}$$

Simplificando la expresión restante:

$$\frac{dp}{dt} = \frac{rb(x-x_i)x}{x^2} = \frac{rb(x-x_i)}{x}$$

Finalmente, como $\frac{x-x_i}{x} = 1 - \frac{x_i}{x} = 1-p$, obtenemos la ecuación diferencial lineal para la proporción:

$$\frac{dp}{dt} = rb(1-p)$$

Separando las variables e integrando en ambos lados:

$$\int \frac{dp}{1-p} = \int rb\,dt$$

Obtenemos la solución logarítmica:

$$-\ln(1-p) = rbt + C$$

Multiplicando por -1 y aplicando la exponencial en ambos miembros:

$$1 - p = e^{-(rbt+C)} = Ae^{-rbt}$$

Donde $A = e^{-C}$ es una constante determinada por las condiciones iniciales. Si definimos $p(0) = p_0$ en el tiempo $t = 0$, entonces $A = 1-p_0$. Finalmente, la solución explícita para la proporción de inconformistas es:

$$p(t) = 1 - (1-p_0)e^{-rbt}$$

Formula iterativa por metodo

1. Método de Euler Explícito

Para deducir el esquema, consideramos la expansión en serie de Taylor de la función $p(t)$ alrededor del punto t_n para un incremento temporal h:

$$p(t_n + h) = p(t_n) + h\frac{dp}{dt}\bigg|_{t_n} + \frac{h^2}{2!}\frac{d^2p}{dt^2}\bigg|_{\xi}$$

donde $\xi \in [t_n, t_{n+1}]$. Al despreciar los términos de orden superior ($\mathcal{O}(h^2)$), obtenemos la aproximación lineal:

$$p(t_{n+1}) \approx p(t_n) + hp'(t_n)$$

Utilizando la notación discreta $p_{n+1} \approx p(t_{n+1})$ y $p_n \approx p(t_n)$, y sustituyendo la pendiente dada por nuestra ecuación diferencial $\frac{dp}{dt} = rb(1-p)$, llegamos a la fórmula iterativa:

$$p_{n+1} = p_n + h\left[rb(1 - p_n)\right]$$

Esta expresión permite calcular el estado futuro del sistema basándose exclusivamente en la información del estado actual.

2. Método de Taylor de Orden 2

Partimos de la expansión de Taylor de la función $p(t)$ hasta el segundo orden:

$$p(t_{n+1}) = p(t_n) + hp'(t_n) + \frac{h^2}{2}p''(t_n) + \mathcal{O}(h^3)$$

De la ecuación diferencial original, sabemos que $p'(t) = rb(1 - p)$. Calculamos la segunda derivada temporal:

$$p''(t) = \frac{d}{dt}[rb(1 - p)] = -rb\frac{dp}{dt} = -(rb)^2(1 - p)$$

Sustituyendo estas expresiones en la serie de Taylor, obtenemos la fórmula iterativa:

$$p_{n+1} = p_n + hrb(1 - p_n) - \frac{h^2(rb)^2}{2}(1 - p_n)$$

3. Método del Trapecio

Para el método del trapecio, integramos la ecuación diferencial $p'(t) = rb(1 - p)$ en el intervalo $[t_n, t_{n+1}]$:

$$\int_{t_n}^{t_{n+1}} \frac{dp}{dt}dt = \int_{t_n}^{t_{n+1}} rb(1 - p(t))dt$$

Aplicando el Teorema Fundamental del Cálculo y aproximando la integral mediante la regla del trapecio:

$$p_{n+1} - p_n = \frac{h}{2}\left[rb(1 - p_n) + rb(1 - p_{n+1})\right]$$

Para obtener la fórmula iterativa, despejamos p_{n+1} mediante los siguientes pasos algebraicos:

Agrupamos términos constantes:

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

Expandimos la expresión para separar las variables en t_{n+1}:

$$p_{n+1} = p_n + hrb - \frac{hrb}{2}p_n - \frac{hrb}{2}p_{n+1}$$

Agrupamos todos los términos con p_{n+1} en el miembro izquierdo:

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

Finalmente, despejamos p_{n+1} para obtener el esquema explícito:

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

Este esquema es incondicionalmente estable y posee un error de truncamiento local de orden $\mathcal{O}(h^3)$.

1.1. Análisis Numérico

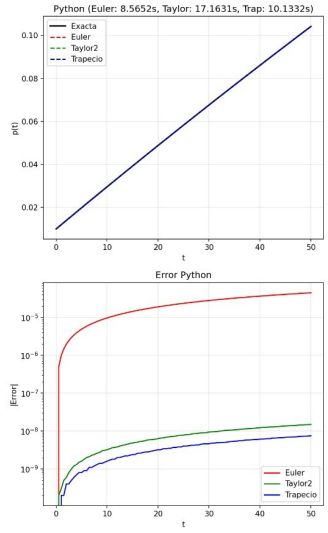

Figura 1: Análisis número de funcion $p(t)$ en Python

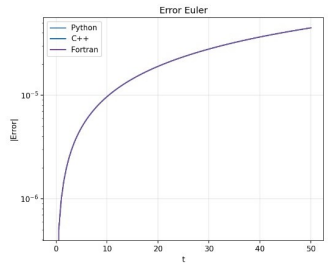

Figura 2: Error de Euler por Lenguaje de programación

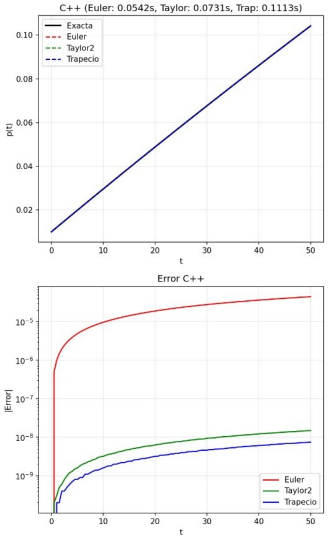

Figura 3: Análisis número de funcion $p(t)$ en C++

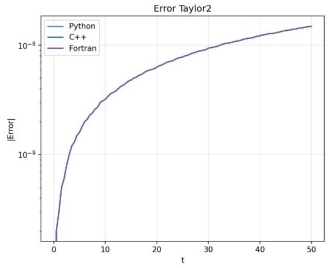

Figura 4: Error de Taylor por Lenguaje de programación

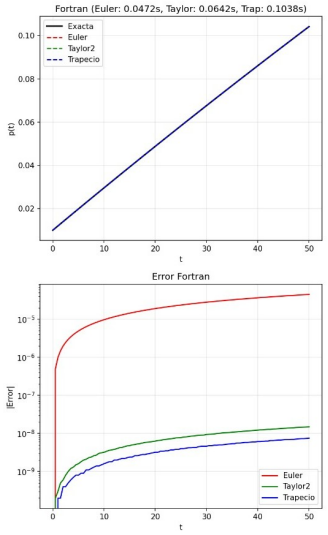

Figura 5: Análisis número de funcion $p(t)$ en Fortran

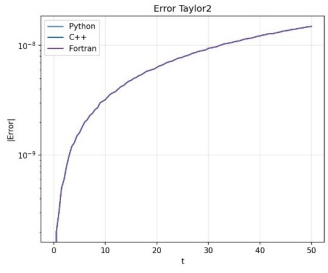

Figura 6: Error de Trapecio por Lenguaje de programación

2. Implementacion de los 3 metodos en Python, C++, Fortran

```python
import numpy as np
import time

# Par metros
b, d, r, p0, t_max = 0.02, 0.015, 0.1, 0.01, 50.0

def f(p): return r * b * (1 - p)
def f_prime(p): return -r * b * f(p)
def exacta(t): return 1 - (1 - p0) * np.exp(-r * b * t)

def euler(p0, h, n):
    p = np.zeros(n + 1)
    p[0] = p0
    start = time.time()
    for i in range(n):
        p[i + 1] = p[i] + h * f(p[i])
    return p, time.time() - start

def taylor2(p0, h, n):
    p = np.zeros(n + 1)
    p[0] = p0
    start = time.time()
    for i in range(n):
        p[i + 1] = p[i] + h * f(p[i]) + (h**2 / 2) * f_prime(p[i])
    return p, time.time() - start

def trapecio(p0, h, n):
    p = np.zeros(n + 1)
    p[0] = p0
    factor = h * r * b / 2
    start = time.time()
    for i in range(n):
        p[i + 1] = (p[i] + (h / 2) * f(p[i]) + factor) / (1 + factor)
    return p, time.time() - start

if __name__ == '__main__':
    # Benchmark con 10^7 pasos
    n_steps_bench = int(1e7)
    h_bench = t_max / n_steps_bench

    print(f"Benchmark Python - 10^7 pasos")

    _, t_euler = euler(p0, h_bench, n_steps_bench)
    print(f"Euler:    {t_euler:.4f} s")

    _, t_taylor = taylor2(p0, h_bench, n_steps_bench)
    print(f"Taylor2:  {t_taylor:.4f} s")

    _, t_trapecio = trapecio(p0, h_bench, n_steps_bench)
    print(f"Trapecio: {t_trapecio:.4f} s")

    # Guardar benchmark
    with open('benchmark_python.dat', 'w') as archivo:
        archivo.write(f"Euler {t_euler:.10f}\n")
```

```
        archivo.write(f"Taylor2 {t_taylor:.10f}\n")
        archivo.write(f"Trapecio {t_trapecio:.10f}\n")

    # Calcular soluciones con h=0.5 para graficar
    n_steps_vis = 100
    h_vis = t_max / n_steps_vis

    p_euler, _ = euler(p0, h_vis, n_steps_vis)
    p_taylor, _ = taylor2(p0, h_vis, n_steps_vis)
    p_trapecio, _ = trapecio(p0, h_vis, n_steps_vis)

    t = np.linspace(0, t_max, n_steps_vis + 1)
    p_exact = exacta(t)

    # Guardar soluciones
    with open('soluciones_python.dat', 'w') as archivo:
        archivo.write("# t Exacta Euler Taylor2 Trapecio\n")
        for i in range(len(t)):
            archivo.write(f"{t[i]:.6f} {p_exact[i]:.10f} {p_euler[i]:.10f} {
                p_taylor[i]:.10f} {p_trapecio[i]:.10f}\n")
```

Listing 1: Implementación en Python

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <chrono>
#include <iomanip>
#include <utility>
using namespace std;
using namespace std::chrono;

const double b = 0.02, d = 0.015, r = 0.1, p0 = 0.01, t_max = 50.0;

inline double f(double p) { return r * b * (1 - p); }
inline double f_prime(double p) { return -r * b * f(p); }
inline double exacta(double t) { return 1 - (1 - p0) * exp(-r * b * t); }

pair<double, double> euler(double p0, double h, long n)
{
    double p = p0;
    auto start = high_resolution_clock::now();
    for (long i = 0; i < n; i++)
        p = p + h * f(p);
    auto end = high_resolution_clock::now();
    return {p, duration<double>(end - start).count()};
}

vector<double> euler_solucion(double p0, double h, long n)
{
    vector<double> p(n + 1);
    p[0] = p0;
    for (long i = 0; i < n; i++)
        p[i + 1] = p[i] + h * f(p[i]);
    return p;
}
```

```cpp

pair<double, double> taylor2(double p0, double h, long n)
{
    double p = p0;
    auto start = high_resolution_clock::now();
    for (long i = 0; i < n; i++)
        p = p + h * f(p) + (h * h / 2.0) * f_prime(p);
    auto end = high_resolution_clock::now();
    return {p, duration<double>(end - start).count()};
}

vector<double> taylor2_solucion(double p0, double h, long n)
{
    vector<double> p(n + 1);
    p[0] = p0;
    for (long i = 0; i < n; i++)
        p[i + 1] = p[i] + h * f(p[i]) + (h * h / 2.0) * f_prime(p[i]);
    return p;
}

pair<double, double> trapecio(double p0, double h, long n)
{
    double p = p0;
    double factor = h * r * b / 2.0;
    auto start = high_resolution_clock::now();
    for (long i = 0; i < n; i++)
        p = (p + (h / 2.0) * f(p) + factor) / (1.0 + factor);
    auto end = high_resolution_clock::now();
    return {p, duration<double>(end - start).count()};
}

vector<double> trapecio_solucion(double p0, double h, long n)
{
    vector<double> p(n + 1);
    p[0] = p0;
    double factor = h * r * b / 2.0;
    for (long i = 0; i < n; i++)
        p[i + 1] = (p[i] + (h / 2.0) * f(p[i]) + factor) / (1.0 + factor);
    return p;
}

int main()
{
    long n_steps = 10000000;
    double h = t_max / n_steps;

    cout << "Benchmark C++ - 10^7 pasos" << endl;

    auto [p1, t_euler] = euler(p0, h, n_steps);
    cout << fixed << setprecision(4) << "Euler:    " << t_euler << " s" << endl
       ;

    auto [p2, t_taylor] = taylor2(p0, h, n_steps);
    cout << "Taylor2:  " << t_taylor << " s" << endl;

    auto [p3, t_trapecio] = trapecio(p0, h, n_steps);
```

```
    cout << "Trapecio: " << t_trapecio << " s" << endl;

    ofstream f("benchmark_cpp.dat");
    f << setprecision(10);
    f << "Euler " << t_euler << endl;
    f << "Taylor2 " << t_taylor << endl;
    f << "Trapecio " << t_trapecio << endl;
    f.close();

    // Calcular soluciones con h=0.5 para graficar
    long n_vis = 100;
    double h_vis = t_max / n_vis;

    auto p_euler = euler_solucion(p0, h_vis, n_vis);
    auto p_taylor = taylor2_solucion(p0, h_vis, n_vis);
    auto p_trapecio_sol = trapecio_solucion(p0, h_vis, n_vis);

    ofstream sol("soluciones_cpp.dat");
    sol << setprecision(10) << fixed;
    sol << "# t Exacta Euler Taylor2 Trapecio" << endl;
    for (long i = 0; i <= n_vis; i++)
    {
        double t = i * h_vis;
        sol << t << " " << exacta(t) << " " << p_euler[i] << " "
            << p_taylor[i] << " " << p_trapecio_sol[i] << endl;
    }
    sol.close();

    if (p1 < 0 || p2 < 0 || p3 < 0)
        return 1;
    return 0;
}
```

Listing 2: Implementación en C++

```
program benchmark
    implicit none
    real(8), parameter :: b=0.02d0, d=0.015d0, r=0.1d0, p0=0.01d0, t_max=50.0d0
    integer, parameter :: n_steps = 10000000
    integer, parameter :: n_vis = 100
    real(8), parameter :: h = t_max / n_steps
    real(8), parameter :: h_vis = t_max / n_vis
    real(8) :: t_euler, t_taylor, t_trapecio
    real(8), allocatable :: p_euler(:), p_taylor(:), p_trapecio(:)
    integer :: unit, i
    real(8) :: t_val

    print *, 'Benchmark Fortran - 10^7 pasos'

    t_euler = euler(p0, h, n_steps)
    print '(A,ES12.4,A)', 'Euler:    ', t_euler, ' s'

    t_taylor = taylor2(p0, h, n_steps)
    print '(A,ES12.4,A)', 'Taylor2:  ', t_taylor, ' s'

    t_trapecio = trapecio(p0, h, n_steps)
    print '(A,ES12.4,A)', 'Trapecio: ', t_trapecio, ' s'
```

```
23
24      if (t_euler < 0.0d0 .or. t_taylor < 0.0d0 .or. t_trapecio < 0.0d0) stop
25
26      open(newunit=unit, file='benchmark_fortran.dat', status='replace')
27      write(unit, '(A,F20.10)') 'Euler ', t_euler
28      write(unit, '(A,F20.10)') 'Taylor2 ', t_taylor
29      write(unit, '(A,F20.10)') 'Trapecio ', t_trapecio
30      close(unit)
31
32      ! Calcular soluciones para graficar
33      allocate(p_euler(0:n_vis), p_taylor(0:n_vis), p_trapecio(0:n_vis))
34
35      call euler_solucion(p0, h_vis, n_vis, p_euler)
36      call taylor2_solucion(p0, h_vis, n_vis, p_taylor)
37      call trapecio_solucion(p0, h_vis, n_vis, p_trapecio)
38
39      open(newunit=unit, file='soluciones_fortran.dat', status='replace')
40      write(unit, '(A)') '# t Exacta Euler Taylor2 Trapecio'
41      do i = 0, n_vis
42          t_val = i * h_vis
43          write(unit, '(5F20.10)') t_val, exacta(t_val), p_euler(i), &
44                                   p_taylor(i), p_trapecio(i)
45      end do
46      close(unit)
47
48      deallocate(p_euler, p_taylor, p_trapecio)
49
50  contains
51
52      pure function f(p) result(fp)
53          real(8), intent(in) :: p
54          real(8) :: fp
55          fp = r * b * (1.0d0 - p)
56      end function
57
58      pure function f_prime(p) result(fpp)
59          real(8), intent(in) :: p
60          real(8) :: fpp
61          fpp = -r * b * f(p)
62      end function
63
64      function euler(p_init, h, n) result(tiempo)
65          real(8), intent(in) :: p_init, h
66          integer, intent(in) :: n
67          real(8) :: tiempo, p, t1, t2
68          integer :: i
69          p = p_init
70          call cpu_time(t1)
71          do i = 1, n
72              p = p + h * f(p)
73          end do
74          call cpu_time(t2)
75          tiempo = t2 - t1
76          if (p < 0.0d0 .or. p > 2.0d0) print *, 'guard:', p
77      end function
78
```

```
79      function taylor2(p_init, h, n) result(tiempo)
80          real(8), intent(in) :: p_init, h
81          integer, intent(in) :: n
82          real(8) :: tiempo, p, t1, t2
83          integer :: i
84          p = p_init
85          call cpu_time(t1)
86          do i = 1, n
87              p = p + h * f(p) + (h * h / 2.0d0) * f_prime(p)
88          end do
89          call cpu_time(t2)
90          tiempo = t2 - t1
91          if (p < 0.0d0 .or. p > 2.0d0) print *, 'guard:', p
92      end function
93
94      function trapecio(p_init, h, n) result(tiempo)
95          real(8), intent(in) :: p_init, h
96          integer, intent(in) :: n
97          real(8) :: tiempo, p, factor, t1, t2
98          integer :: i
99          factor = h * r * b / 2.0d0
100         p = p_init
101         call cpu_time(t1)
102         do i = 1, n
103             p = (p + (h / 2.0d0) * f(p) + factor) / (1.0d0 + factor)
104         end do
105         call cpu_time(t2)
106         tiempo = t2 - t1
107         if (p < 0.0d0 .or. p > 2.0d0) print *, 'guard:', p
108     end function
109
110     pure function exacta(t) result(p)
111         real(8), intent(in) :: t
112         real(8) :: p
113         p = 1.0d0 - (1.0d0 - p0) * exp(-r * b * t)
114     end function
115
116     subroutine euler_solucion(p_init, h, n, sol)
117         real(8), intent(in) :: p_init, h
118         integer, intent(in) :: n
119         real(8), intent(out) :: sol(0:n)
120         real(8) :: p
121         integer :: i
122         p = p_init
123         sol(0) = p
124         do i = 1, n
125             p = p + h * f(p)
126             sol(i) = p
127         end do
128     end subroutine
129
130     subroutine taylor2_solucion(p_init, h, n, sol)
131         real(8), intent(in) :: p_init, h
132         integer, intent(in) :: n
133         real(8), intent(out) :: sol(0:n)
134         real(8) :: p
```

```
        integer :: i
        p = p_init
        sol(0) = p
        do i = 1, n
            p = p + h * f(p) + (h * h / 2.0d0) * f_prime(p)
            sol(i) = p
        end do
    end subroutine

    subroutine trapecio_solucion(p_init, h, n, sol)
        real(8), intent(in) :: p_init, h
        integer, intent(in) :: n
        real(8), intent(out) :: sol(0:n)
        real(8) :: p, factor
        integer :: i
        p = p_init
        factor = h * r * b / 2.0d0
        sol(0) = p
        do i = 1, n
            p = (p + (h / 2.0d0) * f(p) + factor) / (1.0d0 + factor)
            sol(i) = p
        end do
    end subroutine

end program
```

Listing 3: Implementación en Fortran