EDO y partículas

Eduardo Graells-Garrido

26 de marzo de 2025

1 Ecuaciones Diferenciales Ordinarias (EDO)

La naturaleza está en constante cambio. Los planetas orbitan, las hojas caen, el agua fluye y las partículas de humo se dispersan. Para crear mundos virtuales convincentes, necesitamos simular esos (y otros) cambios. Las Ecuaciones Diferenciales Ordinarias (EDO) constituyen el lenguaje que nos permite modelar estos sistemas dinámicos, describiendo cómo sus variables cambian en función del tiempo y de las distintas fuerzas naturales y artificiales involucradas (Fig. 1).

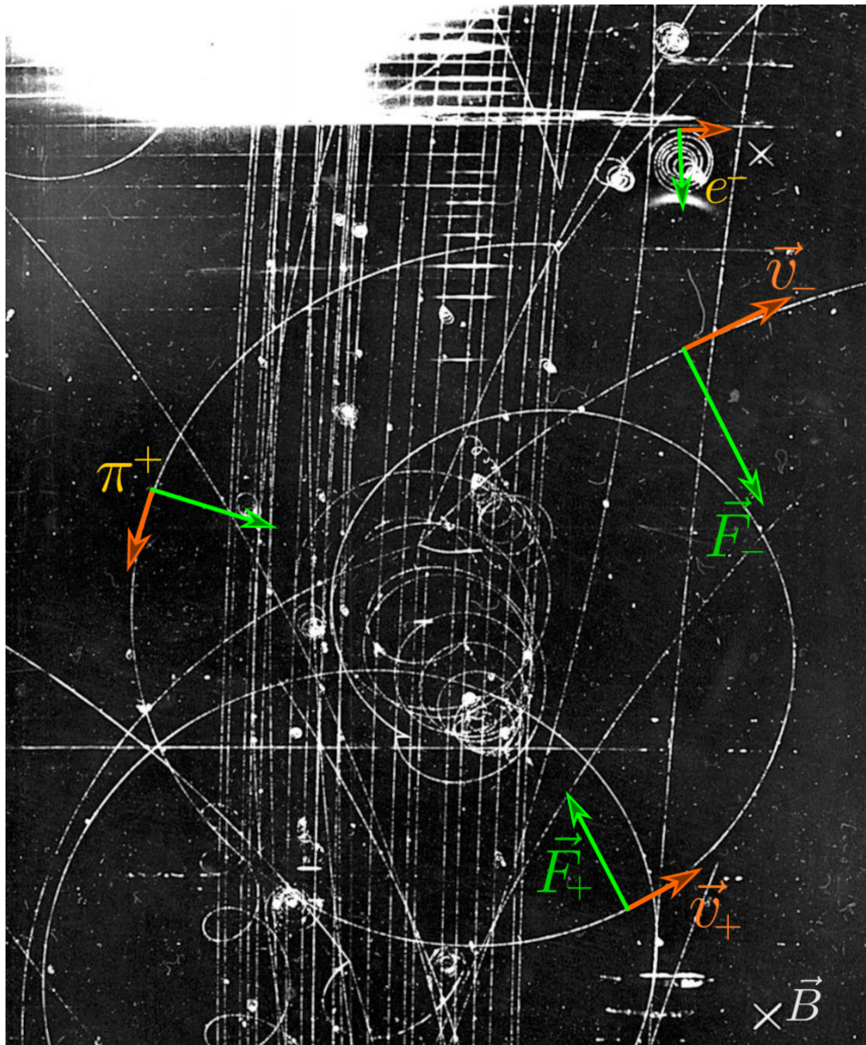


Figura 1. Fuerza de Lorentz. Esta es una fotografía que hizo una persona del Departamento de Energía de los Estados Unidos, y luego alguien que edita Wikipedia anotó las fuerzas involucradas: la Fuerza de Lorentz es la fuerza ejercida por el campo electromagnético que recibe una partícula cargada o una corriente eléctrica. Una imagen tan realista como preciosa. Fuente: Wiki Commons.

Una EDO de primer orden se expresa como:

$$\frac{dy}{dt} = f(t, y).$$

Esta expresión nos indica que la tasa de cambio de y con respecto al tiempo t está determinada por la función f, que depende

tanto del tiempo como del valor actual de y. La EDO en sí misma solo contiene el *cambio del mundo*: para simular es necesario conocer el valor inicial del sistema $y(t_0) = y_0$, lo que permite calcular los estados futuros.

Para sistemas con múltiples variables, usamos:

$$\frac{d\vec{y}}{dt} = \vec{f}(t, \vec{y}).$$

donde $\vec{y}$ es un vector con todas las variables del sistema y $\vec{f}$ describe cómo cambia cada una.

Un ejemplo familiar es el juego *Angry Birds*. Cuando se lanza un pájaro enojado, su trayectoria se calcula con las siguientes ecuaciones:

$$\frac{dx}{dt} = v_x,$$
$$\frac{dy}{dt} = v_y - gt.$$

Este movimiento **tiene solución exacta** porque solo actúa la gravedad en y:

$$x(t) = x_0 + v_x t,$$
$$y(t) = y_0 + v_y t - \frac{1}{2} g t^2.$$

Conociendo estas fórmulas, puedes calcular la posición del pájaro en cualquier instante t. Eso permite añadir elementos a un gráfico como una proyección de la trayectoria que seguiría el ave.

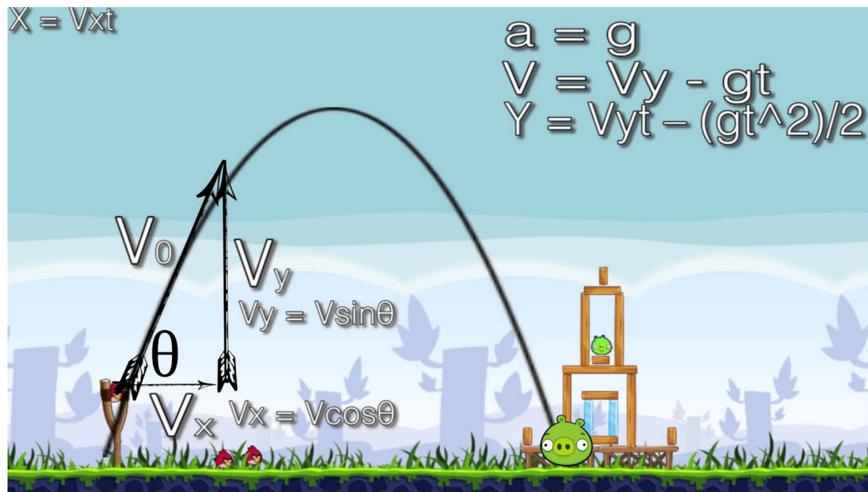


Figura 2. *Angry Birds* (Rovio), donde la solución exacta del movimiento es conocida. Lo que no podremos medir nunca es cuán enojado estaba el pájaro.

Eso sí, la realidad es más compleja y hay muchas más fuerzas que la gravitación. En un *Realistic Angry Birds* consideraríamos:

- **Resistencia del aire**: $F_{aire} = -kv^2$ (depende del cuadrado de la velocidad).

- **Viento variable**: $F_{viento} = f(x, y, t)$ (cambia según posición y tiempo).
- **Efecto Magnus**: Si el pájaro rota, la fuerza depende de la rotación[1].

Al incorporar estas fuerzas al problema, nuestras EDO cambian:

$$\frac{dv_x}{dt} = -kv_x\sqrt{v_x^2 + v_y^2} + \text{viento}_x(x, y, t),$$
$$\frac{dv_y}{dt} = -g - kv_y\sqrt{v_x^2 + v_y^2} + \text{viento}_y(x, y, t).$$

Estas ecuaciones no tienen solución exacta. No existe una fórmula que permita predecir $x(t)$ e $y(t)$. De hecho, la mayoría de las EDO con las que nos encontraremos en la vida profesional carecerán de soluciones analíticas exactas. Para ello utilizaremos la **integración numérica**.

En Computación Gráfica, las EDO que más nos interesan describen el movimiento de objetos bajo fuerzas. La segunda ley de Newton (Fig. 3), $\vec{F} = m\vec{a}$, establece que la aceleración de un objeto es proporcional a la fuerza neta que actúa sobre él. Esto nos da un sistema de dos EDO de primer orden: $\frac{d\vec{r}}{dt} = \vec{v}$ y $\frac{d\vec{v}}{dt} = \vec{F}/m$. Los métodos de integración que veremos operan sobre este sistema, actualizando posición y velocidad en cada paso.

[1]«El efecto Magnus, denominado así en honor al físico y químico alemán Heinrich Gustav Magnus (1802-1870), es el nombre dado al fenómeno físico por el cual la rotación de un objeto afecta a la trayectoria del mismo a través de un fluido, como por ejemplo, el aire. Es producto de varios fenómenos, incluido el principio de Bernoulli y la condición de no deslizamiento del fluido encima de la superficie del objeto». Fuente: Wikipedia.

Figura 3. Isaac Newton. Fuente: Wiki Commons.

Métodos de integración numérica

La integración numérica aproxima soluciones para EDO mediante cálculos iterativos: no se predice la posición de un ave (o de lo que estés simulando) en un tiempo arbitrario mediante una fórmula, sino que calculas su posición en el próximo instante de tiempo (pues sí conoces el cambio: es lo que define la EDO). Y cuando sea ese instante, vuelves a predecir para el instante siguiente. Acumulas pequeños instantes dt (conocido como *paso temporal* o *time step*) hasta llegar al tiempo t de interés.

Exploraremos tres métodos de integración: Euler, Runge-Kutta de 4º orden (RK4) y Verlet, cada uno con características y *trade-offs* de simplicidad, precisión y costo computacional.

Método de Euler

Este método representa la aproximación más simple para resolver EDO de forma numérica. Dada una EDO $\frac{dy}{dt} = f(t, y)$ con condición inicial $y(t_0) = y_0$, la fórmula de iteración o actualización de Euler es:

$$y_{n+1} = y_n + h \cdot f(t_n, y_n),$$

donde h es el tamaño del paso temporal y $t_{n+1} = t_n + h$. Esta fórmula deriva de la aproximación lineal de la serie de Taylor:

$$y(t+h) \approx y(t) + h \cdot \frac{dy}{dt}.$$

En código, un paso de Euler se reduce a una sola línea:

```
def euler_step(f, t, y, h):
    return y + h * f(t, y)
```

El código se evalúa rápidamente (o tan rápido/lento como `f(t, y)`). Sin embargo, esta sencillez tiene un costo: con cada paso, el método acumula error. Tiene error local (por iteración) de orden $O(h^2)$ y error global (por simulación completa) de orden $O(h)$. El error local surge porque Euler trunca la serie de Taylor después del primer término. Al acumular este error a lo largo de $N = T/h$ pasos temporales, el error global se reduce a $O(h)$. El método no conserva propiedades físicas importantes como la energía y el momento, lo que limita su aplicabilidad en simulaciones que requieren fidelidad física.

Método de Runge-Kutta de 4º orden (RK4)

El método RK4 evalúa la derivada en cuatro puntos diferentes para cada paso de integración. Para una EDO $\frac{dy}{dt} = f(t, y)$, la fórmula de RK4 es:

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1), \\ k_3 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2), \\ k_4 &= f(t_n + h, y_n + hk_3), \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

Cada k_i evalúa la función en diferentes posiciones: k_1 al inicio del intervalo, k_2 y k_3 en puntos intermedios, y k_4 al final. El valor final es un promedio ponderado de estas evaluaciones.

Este enfoque aproxima mejor que el método de Euler utilizando el mismo *time step*. RK4 logra error local de orden $O(h^5)$ y error global de orden $O(h^4)$. El promedio ponderado cancela los términos de error hasta h^4, dejando h^5 como el primer término de error significativo. Al reducir el tamaño del paso h a la mitad en cada subevaluación, el error se reduce cerca de 16 veces (2^4). Esto permite que RK4 conserve mejor las propiedades físicas del sistema.

En código, un paso de RK4 explicita las fórmulas anteriores:

```python
def rk4_step(f, t, y, h):
    k1 = f(t, y)
    k2 = f(t + h/2, y + h*k1/2)
    k3 = f(t + h/2, y + h*k2/2)
    k4 = f(t + h, y + h*k3)
    return y + h * (k1 + 2*k2 + 2*k3 + k4) / 6
```

La desventaja principal del método es su mayor costo computacional. Sin embargo, para la mayoría de las aplicaciones en CG, este coste adicional se justifica por la precisión y estabilidad obtenidas.

Método de Verlet

Desarrollado por Loup Verlet en 1967 para simulaciones de dinámica molecular, este método se centra en la integración de la posición sin calcular la velocidad. En Euler y RK4, cada iteración involucra una cadena de cálculos cuando la velocidad también cambia con el tiempo:

- Posición actual (con error acumulado de pasos anteriores).
- Se calcula la aceleración basándonos en esa posición (heredando el error).
- Se calcula una nueva velocidad basándonos en esa aceleración (sumando más error).
- Se calcula una nueva posición basándonos en esa velocidad (sumando aún más error).

El método de Verlet se deriva de la expansión de Taylor de la posición en los puntos $t + h$ y $t - h$:

$$\vec{r}(t+h) = \vec{r}(t) + h\vec{v}(t) + \frac{1}{2}h^2\vec{a}(t) + O(h^3),$$
$$\vec{r}(t-h) = \vec{r}(t) - h\vec{v}(t) + \frac{1}{2}h^2\vec{a}(t) - O(h^3).$$

Sumando estas ecuaciones y despejando, obtenemos:

$$\vec{r}(t+h) = 2\vec{r}(t) - \vec{r}(t-h) + h^2\vec{a}(t) + O(h^4).$$

Esta fórmula ofrece un error local de orden $O(h^4)$ para la posición, incluso superior a RK4, con mayor simplicidad computacional. El método es adecuado cuando se requiere conservación de energía, manteniendo estabilidad incluso para pasos extensos de tiempo. Sin embargo, requiere conocer las posiciones en dos *time steps* y en ocasiones sí necesitamos las velocidades.

Velocity Verlet

El método Velocity Verlet es una variante que calcula de manera explícita tanto velocidades como posiciones. Las fórmulas de actualización son:

$$\vec{r}(t+h) = \vec{r}(t) + h\vec{v}(t) + \frac{1}{2}h^2\vec{a}(t),$$
$$\vec{v}(t+h) = \vec{v}(t) + \frac{h}{2}[\vec{a}(t) + \vec{a}(t+h)].$$

Velocity Verlet destaca por su reversibilidad temporal (simetría en el tiempo). Conserva mejor la energía y ofrece precisión de segundo orden, lo que lo hace ideal para simulaciones físicas. La desventaja es que requiere **dos evaluaciones de fuerzas por paso** (antes y después de mover la posición), lo que duplica el costo de la etapa más cara del ciclo.

Euler simpléctico

Una alternativa más económica es el Euler simpléctico (*symplectic Euler*). A diferencia del Euler explícito, que actualiza posición y velocidad con la misma aceleración, el simpléctico invierte el orden: primero actualiza la velocidad y luego usa esa velocidad nueva para actualizar la posición:

$$\vec{v}_{n+1} = \vec{v}_n + h \cdot \vec{a}_n,$$
$$\vec{x}_{n+1} = \vec{x}_n + h \cdot \vec{v}_{n+1}.$$

Este cambio de orden tiene una consecuencia importante: el método conserva la estructura geométrica del espacio de fases (es *simpléctico*). En la práctica, la energía del sistema oscila en torno al valor correcto en vez de diverger, como ocurre con el Euler explícito. El error local sigue siendo $O(h^2)$, pero el comportamiento a largo plazo es más estable.

La ventaja es que solo necesita **una evaluación de fuerzas por paso**. Para partículas de vida corta (donde el error acumulado no alcanza a ser perceptible), esta reducción a la mitad del costo de fuerzas es un buen compromiso.

Casos de estudio

Para comprender mejor el comportamiento de los métodos de integración numérica, analizaremos tres casos de estudio que ilustran diferentes aspectos de la dinámica de sistemas y el impacto de los métodos de integración (Figura 4). Los tres casos están implementados:

```
uv run python caja_de_juguetes.py edo_case_studies
```

El primer caso es una **EDO cúbica** tiene la siguiente forma:

$$\frac{dy}{dt} = y^3 - y.$$

Esta ecuación se ve simple, pero se trata de una EDO no lineal autónoma (no depende explícitamente del tiempo) con tres puntos de equilibrio: $y = 0, \pm 1$, donde la derivada es cero[2]. Si $|y| < 1$, la solución converge a 0, mientras que si $|y| > 1$, diverge hacia $\pm\infty$. El término cúbico (y^3) representa una retroalimentación positiva que amplifica cualquier desviación, mientras que el término lineal ($-y$) actúa como retroalimentación negativa estabilizadora. Ecuaciones con esta estructura aparecen en modelos de transiciones de fase (donde un material pasa de un estado a otro, como agua a hielo), en dinámica de poblaciones (el efecto Allee, donde una población colapsa si cae por debajo de un umbral crítico) y en redes de regulación genética (donde un gen se activa o desactiva según la concentración de una proteína). Al simular esta ecuación, Euler se desvía de la solución correcta debido a la acumulación de errores, mientras que RK4 mantiene la trayectoria esperada.

[2] El punto $y = 0$ es un punto de equilibrio estable, mientras que $y = -1$ e $y = 1$ son inestables. Un punto de equilibrio inestable es una solución estacionaria de un sistema dinámico donde cualquier perturbación mínima, por pequeña que sea, aleja el sistema de ese punto en lugar de regresar a él. Es como una pelota en la punta de un cerro: representa un estado donde las fuerzas están equilibradas (la derivada es cero), pero este equilibrio es delicado y no perdura ante perturbaciones mínimas. Piensen en eso la próxima vez que envíen alguien a la punta del cerro.

El segundo caso es un sistema de **EDO radial**. si escribimos $\vec{r} = (x, y)$ para la posición, la EDO se expresa como:

$$\frac{d\vec{r}}{dt} = (-y, x).$$

El vector $(-y, x)$ es perpendicular a (x, y) y tiene la misma magnitud, lo que significa que la partícula siempre se mueve perpendicular a su posición radial. Este movimiento perpendicular a la posición radial, con magnitud constante, genera trayectorias circulares. ¡No parece un sistema complicado! De hecho, tiene solución exacta:

$$\begin{aligned} x(t) &= x_0 \cos(t) - y_0 \sin(t), \\ y(t) &= x_0 \sin(t) + y_0 \cos(t), \end{aligned}$$

donde (x_0, y_0) es la condición inicial. El sistema describe un movimiento perpendicular a la posición radial, con magnitud constante: ¡un círculo! Este caso es un test exigente para un integrador numérico: si el método no conserva energía, el radio de la órbita crecerá o decrecerá con el tiempo, delatando el error acumulado. De hecho, al integrar este sistema con el método de Euler, la trayectoria adquiere forma similar a una espiral logarítmica, y la energía del sistema aumenta de forma artificial. En contraste, RK4 mantiene el radio de la órbita casi constante, respetando la física subyacente.

Nuestro tercer caso es un **oscilador no lineal** que surge en circuitos electrónicos con retroalimentación, descrito por la ecuación de van der Pol:

$$\frac{d^2y}{dt^2} - \mu(1 - y^2)\frac{dy}{dt} + y = 0.$$

donde μ es un parámetro positivo que controla la no linealidad del sistema. Esta EDO de segundo orden puede transformarse en un sistema de dos EDO de primer orden:

$$\frac{dy}{dt} = v,$$
$$\frac{dv}{dt} = \mu(1 - y^2)v - y.$$

Desarrollada por Balthasar van der Pol en 1920 al estudiar circuitos con tubos de vacío, esta ecuación presenta un comportamiento llamativo: el término $\mu(1 - y^2)v$ actúa como una amortiguación dependiente de la posición. Cuando $|y| < 1$, tenemos amortiguación negativa (añade energía), y cuando $|y| > 1$, amortiguación positiva (disipa energía). La combinación de efectos opuestos conduce a un ciclo límite: un estado donde las oscilaciones se estabilizan a una amplitud específica, independiente de las condiciones iniciales. Todas las trayectorias son "atraídas" hacia este ciclo característico. Para valores grandes de μ, las oscilaciones adoptan una forma de "relajación", con cambios rápidos seguidos de periodos lentos.

Aunque van der Pol la formuló para circuitos con tubos de vacío, el mismo patrón (amortiguación que cambia de signo según el estado) aparece en contextos muy distintos. En fisiología, el modelo de FitzHugh-Nagumo para el potencial de acción de una neurona es una variante directa del oscilador de van der Pol: la neurona "dispara" cuando acumula suficiente carga y luego se recupera, repitiendo el ciclo de forma autónoma. El mismo principio describe el ritmo cardíaco: las células del nodo sinoauricular se despolarizan y repolarizan siguiendo una dinámica de relajación análoga. En geología, el ciclo de acumulación y liberación de estrés en una falla sísmica (el modelo de *stick-slip*) también se puede modelar con ecuaciones de este tipo. El método RK4 captura la forma correcta de las oscilaciones, mientras que Euler muestra dificultades para mantener la estabilidad del ciclo límite, sobre todo con *time step* grandes.

La Fig. 4 muestra una comparación de estos tres casos utilizando `numpy`.

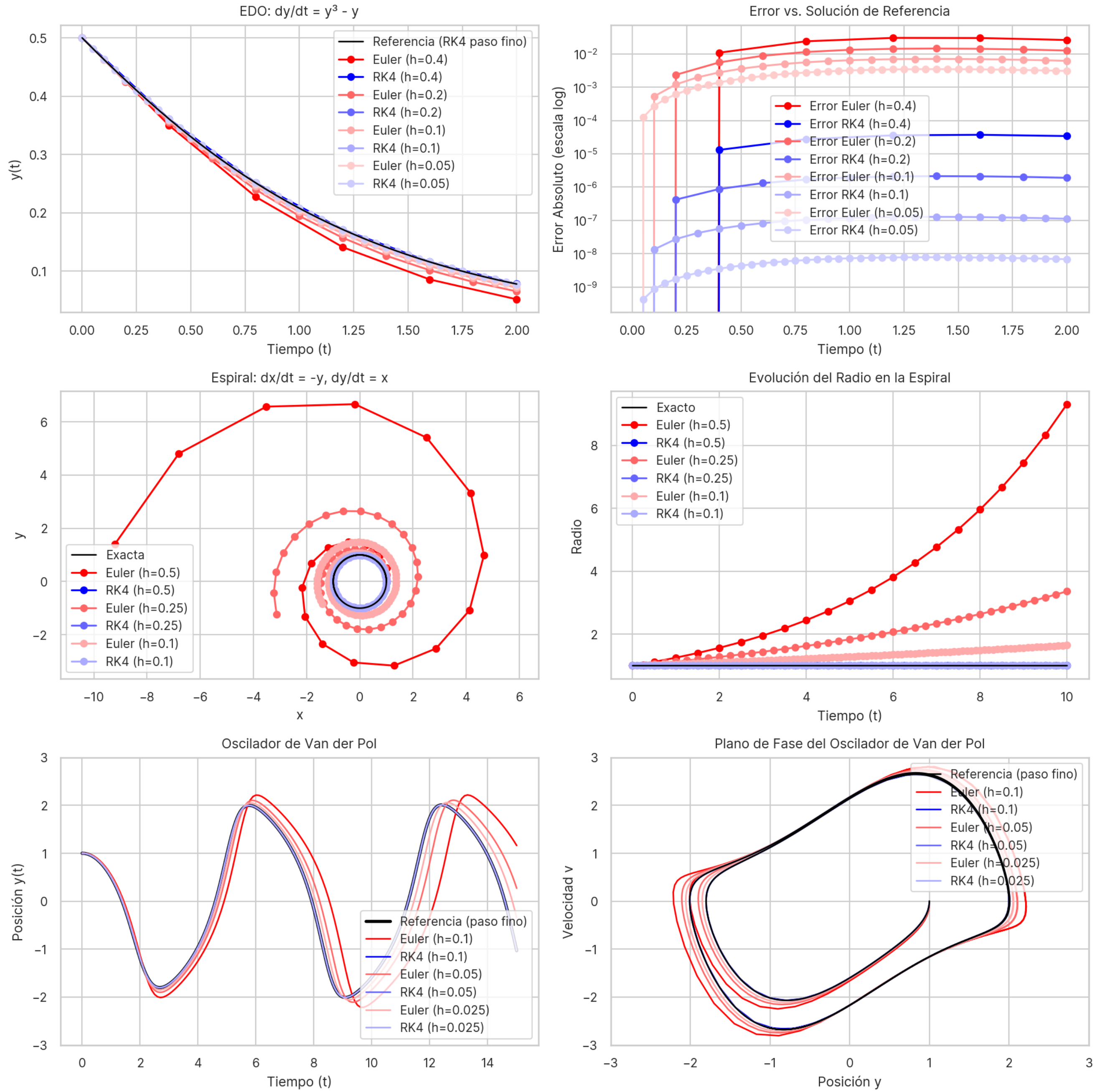


Figura 4. Tres casos de estudio de EDO: una función cúbica (fila superior), una función circular (fila del medio), y un oscilador (fila inferior). Cada línea negra representa al valor analítico de la función. Las líneas azules son integraciones RK4; y las líneas rojas, Euler.

2 De la EDO a la partícula

Ya sabemos resolver EDO y hemos visto cómo se comportan distintos métodos. El siguiente paso es aplicar estas herramientas para modelar una entidad física. En computación gráfica, la abstracción más común es la **partícula**: un objeto puntual con posición $\vec{r}$, velocidad $\vec{v}$, masa m y un tiempo de vida (TTL o *time to live*). Su movimiento sigue el sistema de EDO que introdujimos antes ($d\vec{r}/dt = \vec{v}$, $d\vec{v}/dt = \vec{F}/m$), donde $\vec{F}$ es la fuerza neta que actúa sobre la partícula.

En el archivo `grafica/particle.py` del repositorio del curso, la clase `Particle` encapsula estas propiedades y ofrece un método `update(dt, force_func)` que integra con Velocity Verlet. La función `force_func` recibe la partícula y le aplica fuerzas mediante `apply_force()`.

El ejemplo `flappy_redpanda` implementa un juego al estilo *Flappy Bird* donde un panda rojo debe esquivar obstáculos:

```
uv run python caja_de_juguetes.py flappy_redpanda
```

El panda rojo es una instancia de `Particle` cuya posición vertical está gobernada por la EDO de caída libre:

$$\frac{dy}{dt} = v,$$
$$\frac{dv}{dt} = g,$$

donde g es la aceleración de gravedad (hacia abajo). Es la misma EDO del ejemplo de *Angry Birds*, con la diferencia de que aquí el movimiento horizontal del panda es fijo; solo importa la dinámica vertical (Fig. 5).

Figura 5. *Flappy Red Panda*. El Panda Rojo es el símbolo (¡por lo que entiendo!) del CADCC.

La integración se realiza con Velocity Verlet a través del método `update()` de `Particle`. En cada *frame*, el integrador calcula la nueva posición y velocidad del panda. Al presionar la barra de espacio se aplica un *impulso instantáneo* que reemplaza la velocidad vertical por un valor positivo. Este impulso no es una fuerza (que modificaría la aceleración), sino un cambio directo en la velocidad: una discontinuidad en $v(t)$ que redefine la condición inicial del sistema en ese instante.

Los obstáculos (tubos verdes que se desplazan de derecha a izquierda) no forman parte de la EDO; son elementos geométricos cuya posición se actualiza con velocidad constante. La detección de colisiones compara la posición del panda con los límites de cada tubo y del suelo. Si el panda colisiona, el juego se detiene y permite reiniciar. La detección de colisiones amerita una unidad completa en el curso, pero en este caso, al ser figuras 2D simples, se puede implementar manualmente.

3 Sistemas de partículas

El ejemplo anterior modela una sola entidad. Pero, ¿qué sucede cuando queremos simular humo, fuego o lluvia, donde miles de partículas siguen las mismas leyes físicas con condiciones iniciales distintas? Para eso necesitamos **sistemas de partículas** (Fig. 6).

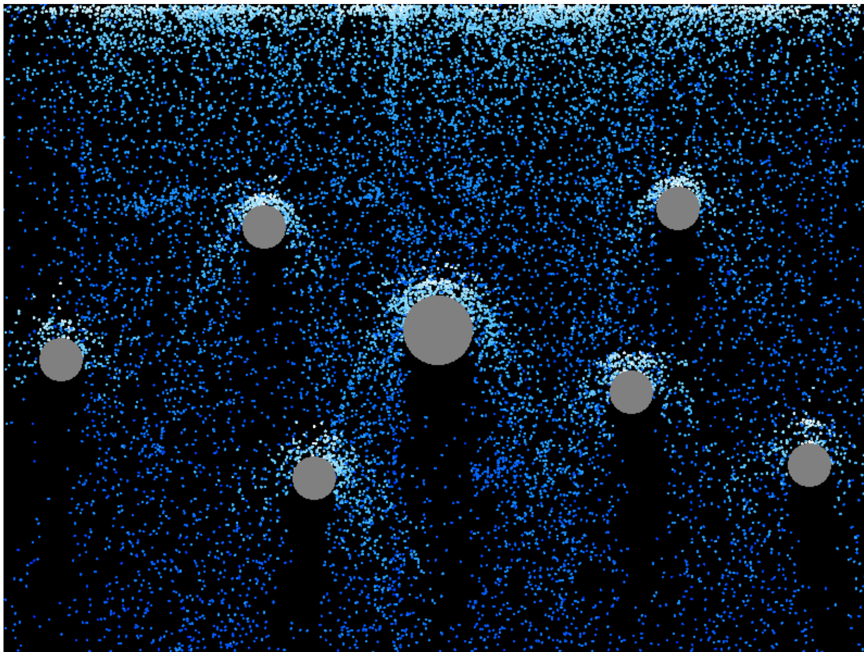


Figura 6. Una simulación de sistema de partículas implementada en WebGL, con cálculos 100% en GPU. Demo: nullprogram.com/webgl-particles. Fuente: github.com/skeeto/webgl-particles.

Un sistema de partículas consiste en *muchos elementos discretos que obedecen las mismas leyes físicas*. Cada partícula posee las

mismas propiedades que ya conocemos (posición, velocidad, masa, TTL), pero ahora las gestionamos en conjunto. En simulaciones físicas permiten recrear órbitas planetarias, modelar la dinámica de fluidos y gases (humo, fuego, agua), simular el comportamiento de materiales como tela o pelo y reproducir fenómenos naturales como lluvia, nieve o polvo. En efectos visuales se usan para generar explosiones, crear estelas de movimiento y rastros, producir efectos atmosféricos e implementar efectos de magia y energía en videojuegos. En visualización científica facilitan la representación de flujos de datos, simulan interacciones moleculares, ilustran la dinámica de poblaciones y ayudan a modelar fenómenos físicos.

Dentro de un sistema de partículas, el ciclo de vida de cada una incluye **generación** mediante un emisor, **simulación** física de su comportamiento, **graficación** y **eliminación** cuando completa su vida útil o sale del área de interés. La Fig. 7 muestra este ciclo. En cada frame, las partículas activas reciben fuerzas (gravedad, viento, turbulencia, contención u otras), el integrador actualiza sus posiciones y velocidades, y el resultado se envía a la GPU para el rendering. Al terminar el frame, se verifica el TTL de cada partícula: si ha expirado, se elimina; si no, vuelve a entrar al ciclo de fuerzas.

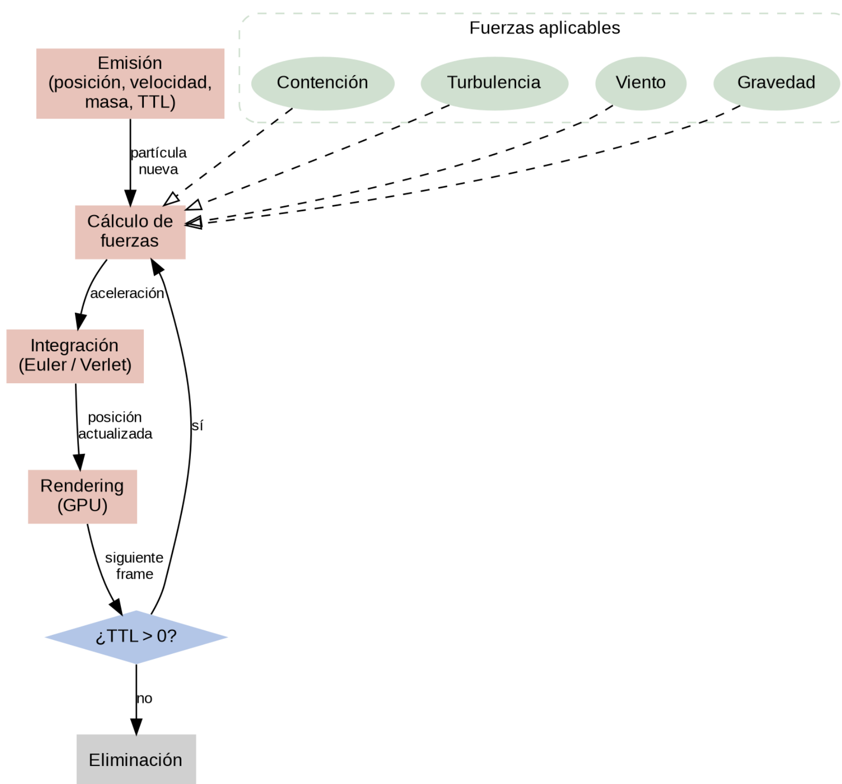


Figura 7. Ciclo de vida de una partícula. Las fuerzas disponibles alimentan el cálculo de aceleración, que el integrador usa para actualizar la posición. Después del rendering, se verifica si la partícula sigue viva. Para sistemas con gran cantidad de partículas a veces Euler resulta suficiente si el *time step* es pequeño. Si las partículas tienen un TTL acotado, la divergencia de Euler no alcanza a ser perceptible. Si se necesita mayor fidelidad, se puede usar Velocity Verlet.

El realismo de un sistema de partículas depende de las fuerzas que actúan sobre cada partícula. Entre las más comunes se encuentran:

- **Fuerzas gravitacionales**: Atraen partículas entre sí siguiendo la ley de Newton ($F = \frac{GMm}{r^2}$).
- **Fuerzas centrales**: Atraen o repelen desde un punto fijo.
- **Fuerzas de vórtice**: Generan movimiento rotacional.
- **Fuerzas de amortiguamiento**: Disipan energía de forma gradual.
- **Fuerzas de contención**: Mantienen partículas dentro de límites definidos.
- **Fuerzas de interacción**: Modelan relaciones entre partículas cercanas.

La combinación de estas fuerzas permite crear una amplia gama de efectos visuales y simulaciones convincentes. Cada partícula sigue el mismo sistema de EDO ($d\vec{r}/dt = \vec{v}$, $d\vec{v}/dt = \vec{F}/m$), pero ahora $\vec{F}$ es la suma de todas las fuerzas que actúan sobre ella.

Ejemplo `particles`

En este ejemplo las partículas se emiten en la posición del *mouse* y caen (Fig. 8), cada una con un tiempo de vida limitado. Se ejecuta así:

```
uv run python caja_de_juguetes.py particles
```

Figura 8. Ejemplo de partículas que siguen al *mouse*.

Al implementar un sistema de partículas hay que decidir cómo se representa el sistema completo. Una primera opción en CPU es representar cada partícula como un objeto independiente, almacenado en una estructura como `deque` (cola de doble extremo) que permite agregar y eliminar con facilidad. Este enfoque (*array of structs*) es intuitivo, pero tiene un costo: iterar sobre miles de objetos en Python es lento, y cada objeto contiene *arrays* pequeños cuyo *overhead* de creación domina sobre la aritmética.

Una alternativa más eficiente es el enfoque *struct of arrays*: en lugar de un objeto por partícula, se mantienen *arrays* NumPy contiguos para cada propiedad (posiciones, velocidades, masas, TTL). Todas las partículas se actualizan con operaciones vectorizadas sobre estos *arrays*, sin *loops* de Python. Se preasigna memoria para un número máximo de partículas y se usa un índice que indica cuántas están activas, lo que evita *allocations* dinámicas en cada *frame*. Las partículas muertas se eliminan compactando los *arrays* con indexación booleana. En efecto, esto es más complejo de implementar que la alternativa anterior, pero la diferencia en desempeño vale la pena[3].

[3]En una implementación en GPU, las posiciones y velocidades se almacenan en texturas y la simulación se realiza en *shaders*. En nuestro caso, la simulación ocurre en CPU y los datos se transfieren a la GPU para el *rendering*.

En el archivo `grafica/particle.py`, la clase `ParticleSystem` implementa el enfoque *struct of arrays* con dos integradores: Velocity Verlet y Euler simpléctico. Las fuerzas se aplican mediante una función que escribe sobre los arrays de aceleración del sistema, sin crear objetos intermedios.

El tamaño y color de cada partícula varía según su tiempo de vida restante. El sistema incorpora las siguientes fuerzas (Fig. 9):

- **Gravedad**: Constante hacia abajo, $(0, -98)$ en unidades del sistema.
- **Viento oscilante**: Fuerza sinusoidal horizontal que varía con el tiempo.
- **Turbulencia aleatoria**: Perturbaciones estocásticas que simulan movimientos impredecibles del aire.
- **Repulsión de bordes**: Mantienen las partículas dentro del área visible.
- **Colisiones con límites**: Rebotes con amortiguación que simulan pérdida de energía.

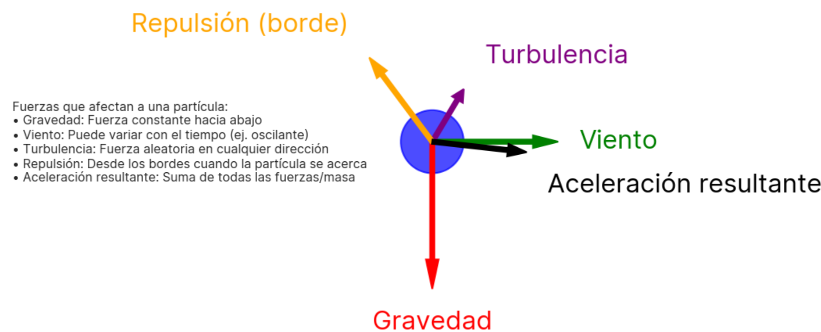


Figura 9. Fuerzas que afectan a una partícula en el ejemplo `particles`.

La combinación de estas fuerzas crea un efecto visual convincente[4]. Cada partícula nace con propiedades individualizadas: posición con leve perturbación gaussiana respecto al punto de emisión, velocidad en dirección aleatoria (distribución angular uniforme), tiempo de vida variable y masa que influye en su respuesta a las fuerzas. Esta variabilidad evita patrones regulares que se perciben como artificiales.

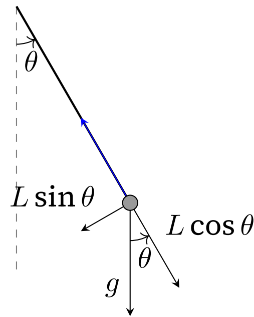
[4] ¿Es realista? ¿O es solo la apariencia? ¿Basta con esto último? Es una de las grandes preguntas que nos haremos en el curso.

4 Ejercicios

Verdadero y falso

1. (Control 1, Otoño 2025) En el método Velocity Verlet para integración numérica, la velocidad se actualiza utilizando solo la aceleración al inicio del paso temporal, similar al método de Euler.
2. (Control 1, Otoño 2025) El método de integración RK4 es siempre la mejor opción para sistemas de partículas en aplicaciones de tiempo real debido a su precisión.
3. (Control 1, Primavera 2025) Para sistemas de partículas con tiempo de vida corto (TTL < 1 segundo), el método de Euler puede ser suficiente porque los errores acumulados no alcanzan a ser perceptibles antes de que la partícula desaparezca.
4. (Control 1, Primavera 2025) El método de Verlet conserva perfectamente la energía en cualquier sistema físico, incluyendo péndulos y órbitas planetarias.

Péndulo (Control 1, Otoño 2025)



Considere un sistema que simula el movimiento de un péndulo simple, descrito por la siguiente ecuación:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin(\theta) = 0$$

donde θ es el ángulo con respecto a la vertical, g es la aceleración gravitacional y L es la longitud del péndulo.

a) Escriba las ecuaciones de actualización para los métodos Euler y Velocity Verlet aplicados a este sistema, expresándolo primero como un sistema de dos ecuaciones de primer orden.

b) En una aplicación de realidad virtual donde se simula un péndulo con el que los usuarios pueden interactuar, el equipo de desarrollo observa que el péndulo aumenta gradualmente su amplitud de oscilación hasta comportarse de manera irreal. Identifique qué método de integración están probablemente utilizando y proponga una solución que resuelva el problema manteniendo un rendimiento adecuado para VR (que requiere alta frecuencia de actualización).

Sistema dinámico *Flappy Bird* (Control 1, Primavera 2025)[5]

En *Flappy Bird*, el pájaro permanece en posición horizontal fija ($x = x_0$) mientras las plataformas se mueven hacia él con velocidad constante $v_{plat} = -2$ m/s. El pájaro cae por gravedad y puede impulsarse verticalmente presionando una tecla.

a) El movimiento vertical del pájaro sigue la ecuación:

$$\frac{d^2y}{dt^2} = -g$$

donde y es la altura, $g = 9.8$ m/s^2, y los impulsos son cambios instantáneos $\Delta v_y = 5$ m/s.

1. Convierta a sistema de primer orden definiendo el vector de estado $\vec{x} = [y, v_y]^T$

[5] En este semestre este control incluyó materia más avanzada, relacionada con *rendering*. La parte (a) se relaciona con esta unidad.

2. Escriba las ecuaciones de actualización con Velocity Verlet (paso temporal Δt):

   - Actualización de posición: $y_{n+1} =$ [complete]
   - Actualización de velocidad: $v_{y,n+1} =$ [complete]

3. ¿Qué ventaja tiene Velocity Verlet sobre Euler para mantener trayectorias parabólicas correctas entre impulsos?

b) Para crear la ilusión de movimiento horizontal, definimos dos sistemas de coordenadas:

Sea el pájaro en posición fija $(x_0, y(t))$ en world space, con $x_0 = 0$ para simplificar. Las plataformas (obstáculos) tienen posiciones $(x_i(t), y_i)$ donde: $x_i(t) = x_{i,inicial} - v_{plat} \cdot t$

con $v_{plat} = 2$ m/s (las plataformas se mueven hacia la izquierda, acercándose al pájaro).

Para mantener al pájaro centrado en pantalla mientras las plataformas se acercan, derive la matriz de vista V usando:

$$V = \text{lookAt}(\vec{eye}, \vec{target}, \vec{up})$$

Determine: 1. $\vec{eye}$ para que la cámara siga horizontalmente al "mundo que se mueve" 2. $\vec{target}$ para mantener el pájaro en el centro visual 3. $\vec{up} = (0, 1, 0)$ (convención estándar)

Exprese los vectores en función del tiempo t y la altura del pájaro $y(t)$.

c) El sprite del pájaro debe rotar según su velocidad vertical. Dado que el pájaro parece avanzar horizontalmente (por el movimiento de las plataformas), definimos una velocidad aparente:

$$\vec{v}_{aparente} = (v_{plat}, v_y(t)) = (2, v_y(t))$$

1. Derive el ángulo de rotación: $\theta = \arctan\left(\frac{v_y(t)}{v_{plat}}\right)$

2. Escriba la matriz de transformación del modelo M (4×4) como producto de matrices elementales:

$$M = T(x_0, y(t), 0) \cdot R_z(\theta)$$

   donde T es traslación y R_z es rotación alrededor del eje Z.

3. Explique por qué enviar M como `uniform mat4` a la GPU es más eficiente que actualizar los vértices del modelo. Mencione el ancho de banda de memoria y la cantidad de datos transferidos.