**INTEGRADOR**

**DE**

**VERLET**

**Física II**

**MISIÓN #1**

Autores:

Marc Pagès

Arnau Falgueras

Marc Ramos

Roger Promera

Pol Camacho

Carlos Redolar

Pablo Galve

Josep Sànchez

Silvino Medina

ÍNDICE

[1. Introducción 3](#_Toc24996351)

[1.1 Integrador 3](#_Toc24996352)

[1.2 Algoritmo de Verlet 3](#_Toc24996353)

[1.3 Historia 3](#_Toc24996354)

[1.4 Objetivos 4](#_Toc24996355)

[2. Desarrollo y código 5](#_Toc24996356)

[2.1 Verlet integration 5](#_Toc24996357)

[2.2 Velocity Verlet 5](#_Toc24996358)

[2.3 Stormer Verlet 6](#_Toc24996359)

[2.4 Acceleration sum & Drag acceleration 7](#_Toc24996360)

[2.5 Terminal Velocity 7](#_Toc24996361)

[2.6 Check Collision 8](#_Toc24996362)

[2.7 Calculate Collision Position 8](#_Toc24996363)

[2.8 Calculate Time 9](#_Toc24996364)

[4. Conclusiones 11](#_Toc24996365)

1. Introducción

Antes que nada, para ponernos en contexto es necesario hacer una introducción para saber qué es el algoritmo de Verlet y qué es un integrador, para así poder entender qué significa cada concepto y el funcionamiento en su totalidad.

1.1 Integrador

Un integrador es un dispositivo que en su salida realiza la operación matemática de integración, es decir, una generalización de la suma de infinitos sumandos, infinitesimalmente pequeños.

1.2 Algoritmo de Verlet

El algoritmo de Verlet es un procedimiento para la integración numérica de ecuaciones diferenciales ordinarias de segundo orden con valores iniciales conocidos. Es especialmente utilizado en las situaciones en que la expresión de la segunda derivada solo es función de las variables, ya sea dependiente o independiente, sin participar la primera derivada.

A parte del algoritmo base de Verlet, existe otro basado en la velocidad, el cual es considerado mejor que el algoritmo original. Es bastante parecido al algoritmo Leapfrog, el cual esta basado en actualizar posiciones y velocidades en puntos del tiempo intercalados, escalonados de tal manera que se saltan los unos a los otros. Pero, a diferencia de este otro algoritmo, la velocidad y la posición son calculados con el mismo valor que la variable de tiempo.

1.3 Historia

La primera persona en presentar la primera versión de la denominada Integración de Verlet fue el matemático francés Loup Verlet, que lo hizo en el año 1967 y se caracterizaba a la vez por su simplicidad, exactitud y estabilidad. La siguiente versión, propuesta en 1985, fue apodada como “Algoritmo de Verlet con velocidad”. Esta consistía en lo mismo que la anterior, con ligeras correcciones en la integración y con mejoras en la precisión y la estabilidad de las soluciones.

1.4 Objetivos

El objetivo principal del proyecto es implementar un integrador basado en el algoritmo de Verlet, el cual tenga un óptimo funcionamiento y que no solo esté basado en cálculos matemáticos y valores, sino que también disponga de una parte gráfica que proyecte todos los cálculos de una forma más visual y fácil de entender.

A parte, no queremos introducir solo las ecuaciones básicas del algoritmo de Verlet, sino ir más allá e implementar las máximas ecuaciones posibles para que el integrador sea lo más completo posible.

Además, queremos que el usuario, a través de la consola, pueda determinar cuales son los valores de las diferentes unidades de las fórmulas, para que el resultado que se refleje de forma gráfica sea el que la propia persona haya introducido en el menú inicial.

2. Desarrollo y código

2.2 Verlet integration



Esta es la ecuación base de la integración de Verlet, la cual hemos utilizado en primer lugar y tiene como unidades la posición, la aceleración y el tiempo.

Para entender cuál es la implementación de código que hemos hecho de esta ecuación, antes de todo, es necesario entender que es lo que hace la ecuación en sí, los valores necesarios para calcularla y cuál es el resultado.

Antes que nada, decir que es una ecuación que lo que hace es calcular la siguiente posición de una partícula, partiendo de una posición actual sumado a la resta de la posición actual y la posición previa. Entonces se multiplica todo está por la aceleración y por el tiempo al cuadrado.

La forma en la cual lo hemos implementado nosotros es con un menú inicial, el cual proporcionara la información necesaria a la ecuación para que se pueda desarrollar. Lo que hace el menú es preguntar al usuario por los valores que quiere introducir, los cuales son la posición actual de la partícula, tanto el valor x como él y, la aceleración, que puede escoger que sea la gravedad (a escoger por el usuario), o que también se tenga en cuenta el rozamiento del aire, etc. En cuanto al tiempo, es un valor el cual está estipulado por nosotros. Entonces, en el caso de que sea la primera iteración de la partícula el valor de la posición previa es 0, pero en el caso de que sea la segunda o más la posición de la primera iteración pasa a ser la posición previa de la segunda. Así pues, en el gráfico se va mostrando cual es la posición de la partícula en cada iteración hasta que llegue a un límite estipulado de loops.

2.3 Velocity Verlet



En este caso, nos encontramos delante de una ecuación parecida a la base del algoritmo de Verlet, pero en este caso, basado en la velocidad.

Lo que hace esta ecuación es calcular la nueva velocidad de una partícula en concreto, mediante el uso de una velocidad actual, una aceleración y un tiempo. Así pues, lo que hace la ecuación es coger la velocidad actual de la partícula, sumarla a la suma de la mitad de la aceleración previa y la aceleración actual y todo ello multiplicado por el tiempo.

En cuanto a la implementación que hemos hecho nosotros en nuestro código, se utiliza para cálculos de futuras ecuaciones, pero la utilización más importante es para el calculo de la velocidad de la partícula en base a la información dada por el usuario. Así pues, al usuario se la pide la velocidad que quiere que tenga la partícula y la aceleración, que puede ser tanto la gravedad como la gravedad, la fuerza de rozamiento del aire, la densidad de la partícula, etc. Así pues, lo que hace el código es ir haciendo iteraciones de la velocidad de la partícula partiendo de la inicial dada por el usuario y a partir de la primera, la propia función va tomando los nuevos valores en función del resultado de la primera iteración.

2.4 Stormer Verlet



A parte de la ecuación anterior, la cual es utilizada para calcular la velocidad de la particular, hay otras formas para poder obtener esta velocidad, como es en este caso la ecuación de Stormer Verlet. El proceso de obtención de la velocidad por parte de esta ecuación es muy sencillo, ya que se basa en la utilización de la ecuación base de Verlet para obtener la posición de la partícula en cada iteración, para así luego poder calcular la velocidad en cada iteración.



Así pues, lo que hace es obtener la posición con la utilización de la ecuación principal, restarla de la posición anterior de la partícula y dividirlo del tiempo transcurrido.

La forma en la cual lo hemos implementado en nuestro código es mediante el mismo método que la ecuación base, en la que el usuario introduce la posición y la aceleración y la ecuación por si sola calcula la posición de la partícula. Una vez calculada la posición se le resta la posición anterior de la partícula que, en el caso de que sea la primera iteración será 0, y si es la segunda o más, será la posición actual de la iteración anterior. Finalmente, se divide el resultado de esta resta al tiempo.

## 2.5 Acceleration sum & Drag acceleration

No se puede entender la función de Acceleration Sum sin antes calcular la de Drag Acceleration. Aquí asumimos que la aceleración de la partícula no es constante y varía según una serie de factores que veremos a continuación.



En este caso, con la Drag Acceleration calculamos la desaceleración de una partícula cuando existe una fuerza de rozamiento (pongamos por caso, con el aire) contraria al movimiento de la partícula. Esta aceleración, que puede ser en el eje x o y, depende de la densidad, del coeficiente de rozamiento, el área de la partícula, la velocidad (en x o y) y la masa de la partícula.

Por lo tanto, el resultado es la multiplicación de la densidad por el coeficiente de rozamiento por el área por ½, todo ello multiplicado por la velocidad al cuadrado (dependiendo de si queremos coger la aceleración en x o y tendremos que utilizar la velocidad en x o y) y todo lo anterior dividido entre la masa.

En el menú inicial le preguntamos al usuario si quiere tener una única aceleración, la gravedad. Si la respuesta es NO, entonces le pedimos que dé valores de gravedad, de coeficiente aerodinámico, de densidad y de masa. La velocidad ya está definida por el usuario anteriormente.

Cabe decir que esta desaceleración es en dirección opuesta al movimiento de la partícula, y por lo tanto se deberá restar a la aceleración final como veremos a continuación.

Con esto ya podemos calcular la Acceleration Sum, que no es más que el cálculo de la aceleración total en un momento determinado. Para ello, mediante un iPoint, creamos la suma de aceleraciones en x y en y, y las inicializamos en 0. Después, igualamos la suma a la aceleración de ese momento de la partícula, y para la suma en y, se le añade la gravedad. Finalmente, la suma total de la aceleración será ella misma menos la Drag Acceleration.

## 2.6 Terminal Velocity

Esta función retorna la velocidad de la partícula en el momento en que la aceleración es constante. Cuando esta partícula cae con una gravedad y una aceleración, la velocidad aumenta. Pero llega un momento en el que la Drag Acceleration y el peso de la partícula (su masa) se igualan, y la velocidad es constante.

## 2.7 Check Collision

Esta función solamente comprueba si existe una colisión entre la partícula y un objeto dentro del mundo desde todas las direcciones posibles (por arriba, por abajo, por la derecha y por la izquierda.

Empezamos creando un booleano llamado ret, y le asignamos el valor false. Este se volverá true cuando exista la colisión, y seguirá falso cuando no la haya.

Para que exista la colisión, se deben cumplir 4 condiciones que están incluidas dentro del if: La primera es si la posición x de la partícula más su radio es igual o superior a la posición x del objeto. La segunda, si la posición x de la partícula menos su radio es igual o menor a la posición x del objeto más su anchura. La tercera, si la posición y de la partícula menos su radio es igual o menor a la posición y del objeto más su altura. Y la cuarta, si la posición y de la partícula más su radio es igual o mayor a la posición y del objeto.

Si cualquiera de estos 4 casos se cumple, entonces el booleano ret se vuelve true y la función retorna ese valor.

A continuación, existe otra función de Check Collision, pero con la diferencia de que retorna todas las posibles colisiones que puedan existir entre la partícula y el objeto.

## 2.8 Calculate Collision Position

Una vez sabemos que se ha producido una colisión, entonces utilizamos esta función para saber qué posición ocupa la partícula justo antes de que esta colisione. Para eso, necesitaremos una variable float llamada time, la función Calculate Time (que se explicará más adelante) y dos booleanos (col\_x y col\_y) que representan las colisiones en x y en y.

Empezamos igualando la variable time a 0 y los booleanos a false, ya que solo serán true si se cumplen 4 condiciones de colisión muy parecidas a las de la función Check Collision.

Si la posición x previa de la partícula (no la actual) más su radio es más pequeña que la posición x del otro objeto, entonces igualamos la variable time a la función Calculate Time, donde le pasamos la posición previa de la partícula, su velocidad en x, su aceleración en x y la posición x del objeto. El booleano col\_x se iguala a true.

Si la posición x previa de la partícula menos su radio es superior a la posición x del objeto más su anchura, igualamos la variable time a la función Calculate Time y le pasamos las mismas variables que antes, pero añadiendo la anchura del objeto. El booleano col\_x se iguala a true.

Si la posición y previa de la partícula más su radio es inferiora la posición y del otro objeto, entonces igualamos la variable time a la función Calculate Time y le pasamos las mismas variables que en la primera condición, pero en el eje y. Ahora el booleano col\_y se iguala a true.

Por último, si la posición y previa de la partícula menos su radio es superior a la posición y del objeto más su altura, entonces igualamos la variable time a la función Calculate Time, y le pasamos las mismas variables que en la segunda condición, pero en el eje y y cambiando la anchura por la altura. El booleano col y se iguala a true.

Una vez calculado el time, ya podemos calcular con exactitud la posición de la partícula en el momento del choque. Y lo hacemos basándonos en esta fórmula de cinemática:

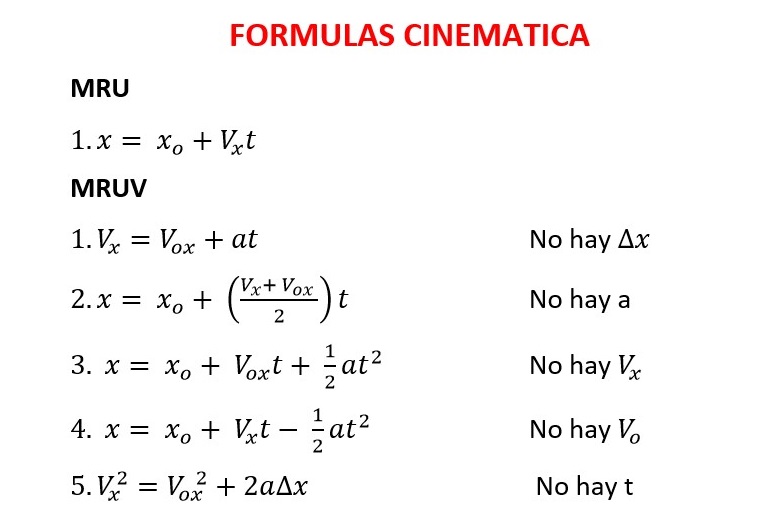


Donde x es la posición en el choque, x0 es la posición previa a la colisión, Vx es la velocidad de la partícula, t es la variable time previamente calculada, y a es la aceleración de la partícula.

## 2.9 Calculate Time

Esta función, como hemos dicho anteriormente, se utiliza siempre que la partícula colisiona. Con ella calculamos el tiempo total de colisión desde que esta se produce hasta que la partícula deja de colisionar. Para ello se usa la posición previa del cuerpo cuando colisiona, la posición nueva cuando deja de colisionar, la velocidad en ese intervalo de tiempo y la aceleración.

Definimos 4 variables float de tiempo: time, time1, time2 y tpow.

Si la aceleración es 0, aislamos el tiempo de la fórmula de cinemática:

Y retornamos la variable time.

Si la aceleración no es 0, entonces calculamos el tpow, que no es más que aislar el tiempo de la fórmula del MRUA:



Como al aislar el tiempo utilizamos una ecuación de segundo grado, el tiempo que aislado de esta manera:

=tpow, siendo b la velocidad de la partícula, a la aceleración y c la resta de la posición inicial menos la final.

Como hay 2 soluciones posibles, el tiempo final se calcula con 2 variables, que son el time 1 y el time 2 respectivamente.

Siendo tpow, como hemos dicho antes, la solución de la raíz cuadrada, el time 1 será la solución positiva de toda esta ecuación, mientras que time 2 será la negativa.

Finalmente, ponemos 2 condiciones. Si el time 1 es más grande que 0, este representa el tiempo total que ha durado la colisión, y se iguala a time. Por el contrario, si time 2 es superior a 0, será este el que se igualará a time. Si ninguno de los 2 es superior a 0, significa que el tiempo de colisión será 0.

4. Conclusiones