**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](#_Toc25165887)

[1.1 Integrador 3](#_Toc25165888)

[1.2 Algoritmo de Verlet 3](#_Toc25165889)

[1.3 Historia 3](#_Toc25165890)

[1.4 Objetivos 4](#_Toc25165891)

[2. Desarrollo y código 5](#_Toc25165892)

[2.1 Menu 5](#_Toc25165893)

[2.2 Main Verlet functions 5](#_Toc25165894)

[2.2.1 Initial Situation 5](#_Toc25165895)

[2.2.2 Verlet integration 6](#_Toc25165896)

[2.2.3 Velocity Verlet 6](#_Toc25165897)

[2.2.4 Stormer Verlet 7](#_Toc25165898)

[2.3 Collision functions 7](#_Toc25165899)

[2.3.1 Check Collision 7](#_Toc25165900)

[2.3.3 Calculate Time 8](#_Toc25165901)

[2.3.4 Calculate Collision Position 9](#_Toc25165902)

[2.4 Acceleration and velocity functions 10](#_Toc25165903)

[2.4.1 Acceleration sum & Drag acceleration 10](#_Toc25165904)

[2.4.2 Terminal Velocity 11](#_Toc25165905)

[2.4.3 Parachutist Acceleration 11](#_Toc25165906)

[2.4.4 Freefall speed 11](#_Toc25165907)

[2.4.5 Freefall acceleration 11](#_Toc25165908)

[2.5 Position functions 12](#_Toc25165909)

[2.5.1 Classical Motion 12](#_Toc25165910)

[2.5.3 Time to position 12](#_Toc25165911)

[4. Conclusiones 13](#_Toc25165912)

1. Introducción

En este documento vamos a describir nuestro integrador de Verlet. Pero, antes de 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 está 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 LoupVerlet, 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 cuáles 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

Antes de explicar el funcionamiento de todas las fórmulas y funciones implementadas en el integrador, creemos que es necesario explicar de dónde adquieren el valor de los componentes de las ecuaciones. Estos valores pueden ser adquiridos a través del menú del usuario, ya que este se los puede proporcionar luego a las funciones o a partir de otras funciones que calculen componentes que necesiten otras funciones para hacer el cálculo de una componente en específico.

## 2.1 Menú

Todo i que en la descripción de cada función se explica cuál es su funcionamiento en verso al menú principal, creemos que es necesario explicar cuál es su funcionamiento y la forma en que lo ve el usuario.

Así pues, lo que hace el menú es, en primer lugar, hacer escoger al usuario entre cuatro opciones: calcular valores específicos, hacer una representación gráfica de la posición de la partícula, simularla en tiempo real o salir del menú y del integrador.

* En el caso de que el usuario escoja la primera opción, el programa le volverá a preguntar si quiere calcular una posición en un momento concreto, el tiempo que tarda la partícula en alcanzar un punto en concreto o la velocidad terminal de la partícula. Entonces, en todas ellas el usuario deberá introducir valores que se piden en las tres opciones, pero habrá algunas preguntas concretas de cada uno, como, por ejemplo, en la primera opción se pide el momento concreto en que se quiere calcular la posición.
* En segundo lugar, tenemos la opción de que se haga una representación gráfica de la partícula. En este caso, se pide al usuario que introduzca una posición x y una posición y, lo mismo con la velocidad, una aceleración, que pude ser la gravedad a secas, o la gravedad teniendo en cuenta el rozamiento, el cuál depende de la densidad de la partícula y la masa de esta entre otros valores. Una vez que todos los valores estén introducidos, el integrador procederá a hacer la representación gráfica del comportamiento de la partícula.
* Por último, en el caso de que el usuario escoja la opción de simulación en tiempo real, se pedirá al usuario exactamente lo mismo que en el caso anterior, pero a la hora de hacer la representación gráfica, esta será en tiempo real, es decir, el comportamiento exacto de la partícula momento a momento.

## 2.2 MainVerletfunctions

### 2.2.1InitialSituation

Esta función ha sido diseñada para el correcto funcionamiento de VerletIntegration, la cual explicaremos en el siguiente punto. Aun así, es necesario empezar a explicarla, para así poder explicar bien cuál es la función de esta. La función VerletIntegration, lo que se hace es calcular la posición que tendrá la partícula en la siguiente iteración mediante la utilización de una posición actual y una posición previa, una aceleración y un tiempo. Así pues, en la primera iteración, la posición actual, la aceleración y el tiempo son proporcionados por el usuario o ya están estipulados por el integrador. En cambio, para poder calcular la posición previa, es necesario utilizar una ecuación aparte, y aquí es donde entra esta función.

Así pues, dado que esta ecuación sí que utiliza la velocidad y sumado a todos los valores que hemos dicho anteriormente, esta función es capaz de calcular la posición previa de la partícula en la primera iteración para poder proporcionarla a la función de VerletIntegration y pueda calcular la nueva posición de la partícula, tanto en la primera iteración como en las siguientes.

2.2.2Verletintegration



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

Para entender cuál es la implementación de código que hemos hecho en esta ecuación, antes de todo es necesario entender qué 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 suma a la multiplicación de la aceleración por el tiempo al cuadrado.

2.2.3 VelocityVerlet



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.

2.2.4 StormerVerlet



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 StormerVerlet. 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.

## 2.3 Collision functions

### 2.3.1 CheckCollision

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.

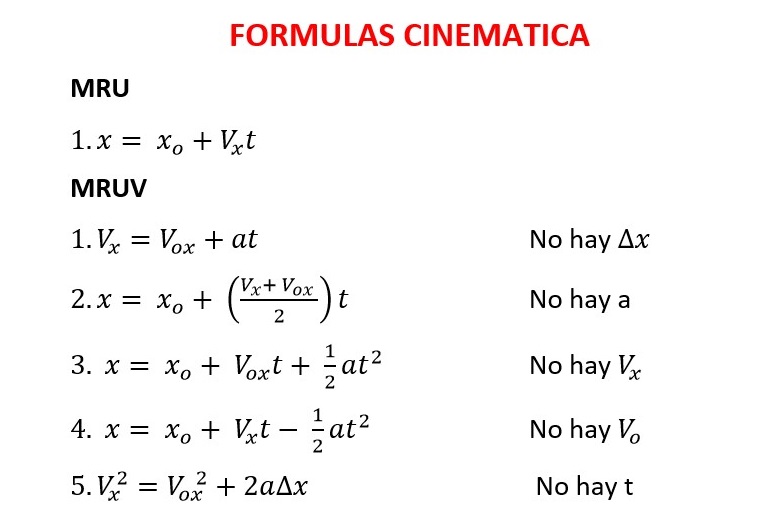
Si cualquiera de estos 4 casos se cumple, entonces el booleano ret se vuelve true y la función retorna ese valor.

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

### 2.3.3 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.

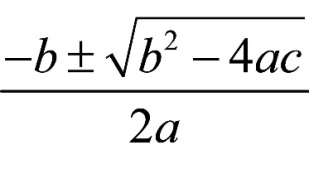
Si la aceleración no es 0, entonces calculamos el tpow, que no es más que aislar el tiempo de la fórmula del MRUA:



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.

### 2.3.4 CalculateCollision 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 CheckCollision.

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 inferior a 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.

Finalmente, calculamos la velocidad de la partícula durante la colisión. Para ello, utilizamos los booleanos col\_x y col\_y. Si col\_x es true, entonces la velocidad final en x será ella misma menos ella misma por 0,9. Y si col\_y es true, pasará lo mismo, pero con la velocidad en y.

## 2.4 Acceleration and velocity functions

### 2.4.1 Acceleration sum & Drag acceleration

No se puede entender la función de Acceleration Sum sin antes calcular la de DragAcceleration. 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 DragAcceleration 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 DragAcceleration.

### 2.4.2 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 DragAcceleration y el peso de la partícula (su masa) se igualan, y la velocidad es constante.

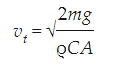
### 2.4.3 ParachutistAcceleration

En el caso que la opción que elija el usuario en base a la aceleración sea únicamente la gravedad esta función no tiene sentida ser aplicada, pero en caso de que elija que existan fuerzas de rozamiento, densidad, área y demás si que se puede aplicar esta función.

Para calcular la aceleración, lo que hace la ecuación es multiplicar la masa negativa por la gravedad y sumarlo a dos veces la gravedad y a una constante. Finalmente, se divide todo esto a la masa de la partícula.

Todos los valores de la ecuación son dados por el usuario cuando complete el menú inicial de integrador.

### 2.4.4 Freefallspeed



El objetivo de esta función es retornar la velocidad de la partícula en el caso que el usuario elija que como aceleración existan fuerzas de rozamiento, la densidad del aire, el área de la partícula, etc. Así pues, para calcular la velocidad se multiplica dos por la gravedad y por la masa. Entonces, se divide todo esto a la multiplicación de la densidad del aire, el área y la fricción del aire. Finalmente, se eleva al cuadrado toda la ecuación.

### 

### 2.4.5 Freefallacceleration

El objetivo de esta función es muy parecido que el de la función anterior, ya que lo que hace es devolver la aceleración de la partícula en el caso de que el usuario lo desee.

Las componentes que necesita la ecuación para poder calcular la aceleración son la masa, la gravedad y la fricción del aire. En primer lugar, se multiplica la masa por la gravedad y se le resta la masa por la gravedad y por la fricción del aire. Finalmente, se divide todo ello por la masa de la partícula.

## 2.5 Position functions

### 2.5.1 ClassicalMotion

Xn+1 = X + V\*dt + ½ a\*dt^2

En el caso de la función ClassicalMotion, su funcionamiento es prácticamente el mismo que en la función base de Verlet, la única diferencia es que en este caso sí que se utiliza la velocidad para calcular la nueva posición de la partícula, y en consecuencia, no necesita otra función que le calcule la posición previa. Así pues, lo que hace es esta ecuación es, por decirlo de alguna manera, sustituir la posición previa por la velocidad, para así poder calcular la posición.

Su funcionamiento pasa por diferentes valores, que son: la posición actual de la partícula, la velocidad y la aceleración, sin olvidarnos del tiempo que es establecido por nosotros. El calculo se realiza de la siguiente manera:

En primer lugar, se suma la posición actual a la velocidad multiplicada del tiempo y seguidamente, se multiplica a la multiplicación de la mitad de la aceleración y el tiempo al cuadrado.

### 2.5.3 Time to position

El objetivo de esta función es devolver cuanto tiempo tarda la partícula en llegar a una posición en concreto. Para ello utiliza una función (ClassicalMotion)explicada anteriormente para poder adquirir variables necesarias para el cálculo.

El funcionamiento de la función es el siguiente:

* Lo primera que hace la función es calcular algunas variables que son necesarias para el calcula del tiempo. Dos de ellas, la aceleración y la posición inicial son dadas por el usuario en el menú, pero en el caso de la posición siguiente se necesaita hacer el calcular con la función ClassicalMotion, la cual hemos explicado anteriormente.
* En el momento en el que el usuario elige si quiere realizar el cálculo en el axis x o en él y, la función acceda a un if o a otro. En ambos se realizan los mismos cálculos, pero cada uno en sus respectivos ejes.
* Así pues, los cálculos realizados por la función son, en primer lugar, un bucle el cual va iterando hasta que llegue al máximo de las iteraciones posibles (cuando ya ha llegado a la posición). En este bucle, en el caso que todavía no se haya llegado al máximo de las iteraciones se ejecuta una commutación, es decir, se atribuye la posición de la partícula a una variable que se guarda. Seguidamente, mediante la función Verletintegration, se calcula la nueva posición de la partícula y se le atribuye este valor a la posición de la partícula. Finalmente, se atribuye el valor inicial guardado a la posición previa para que en la siguiente iteración se pueda realizar el cálculo de la función VerletIntegration otra vez.

4. Ejemplos

En este apartado vamos a mostrar algunos ejemplos de cómo se grafica la partícula en cada uno de los casos que podemos elegir en el menú inicial, el cual es rellenado por el usuario.

## 4.1 Calculatespecificvalues

* Valores introducidos

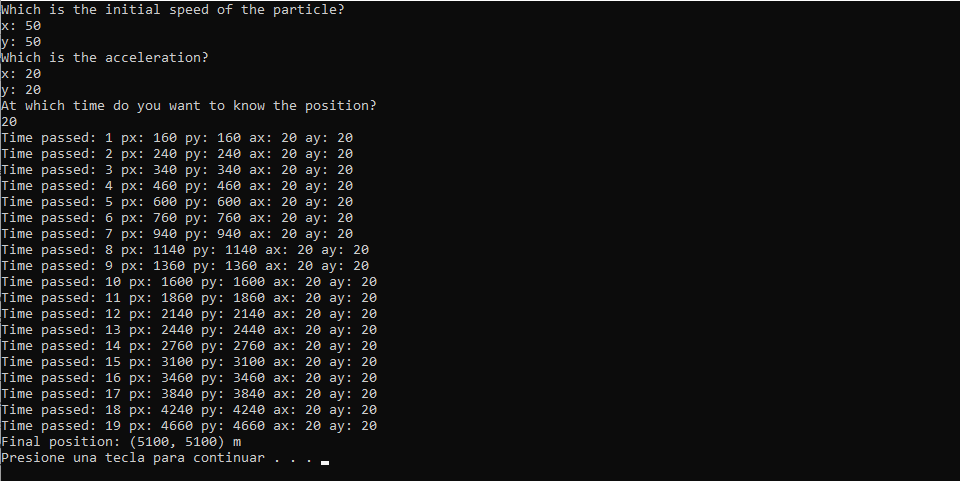
Posición x e y: 100, 100

Velocidad inicial x e y: 50, 50

Aceleración x e y: 20, 20

Tiempo: 20

* Gráfico



## 4.2 Do graphicalpresentation of the position

* Valores introducidos

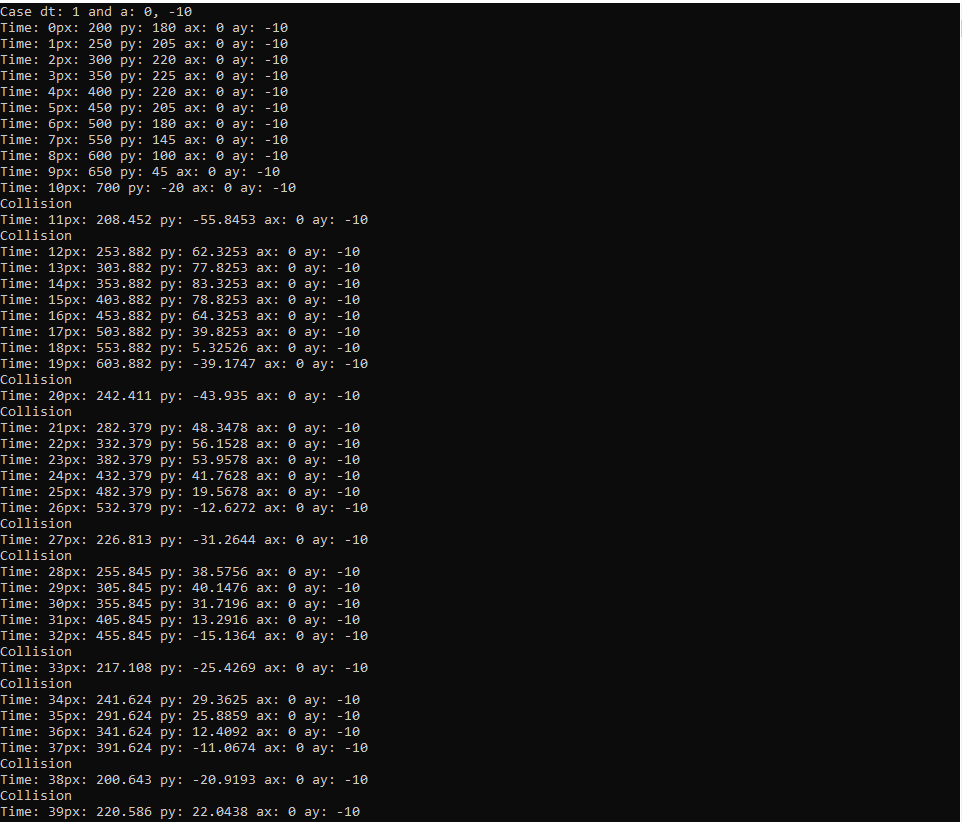
Posición x e y: 100, 100

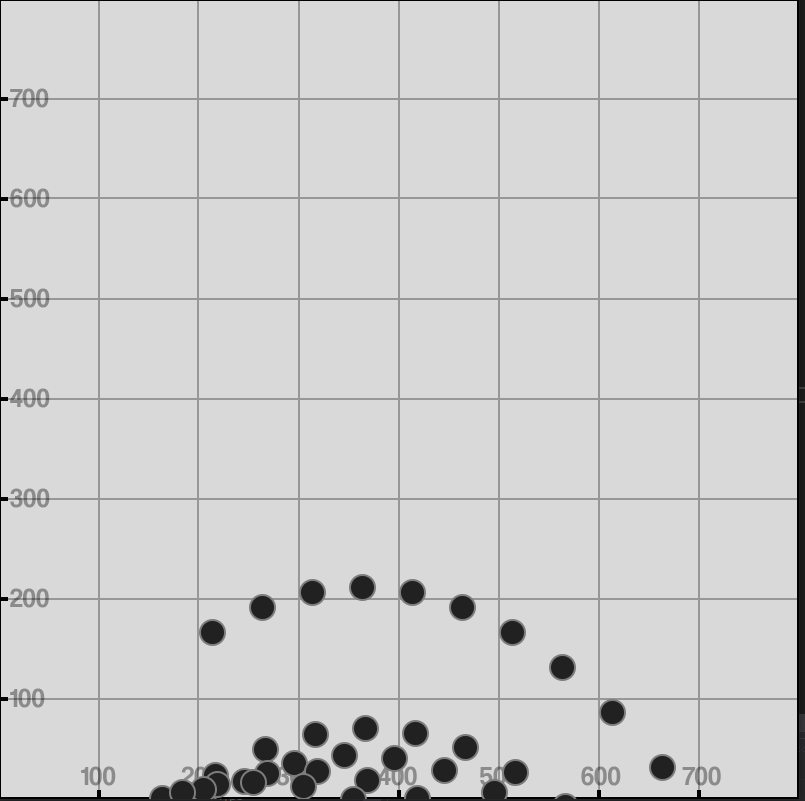
Velocidad inicial x e y: 50, 50

Gravedad: -10

Masa: 5

* Gráfico





## 4.3 Simulate in real time

* Valores introducidos

Posición x e y: 100, 100

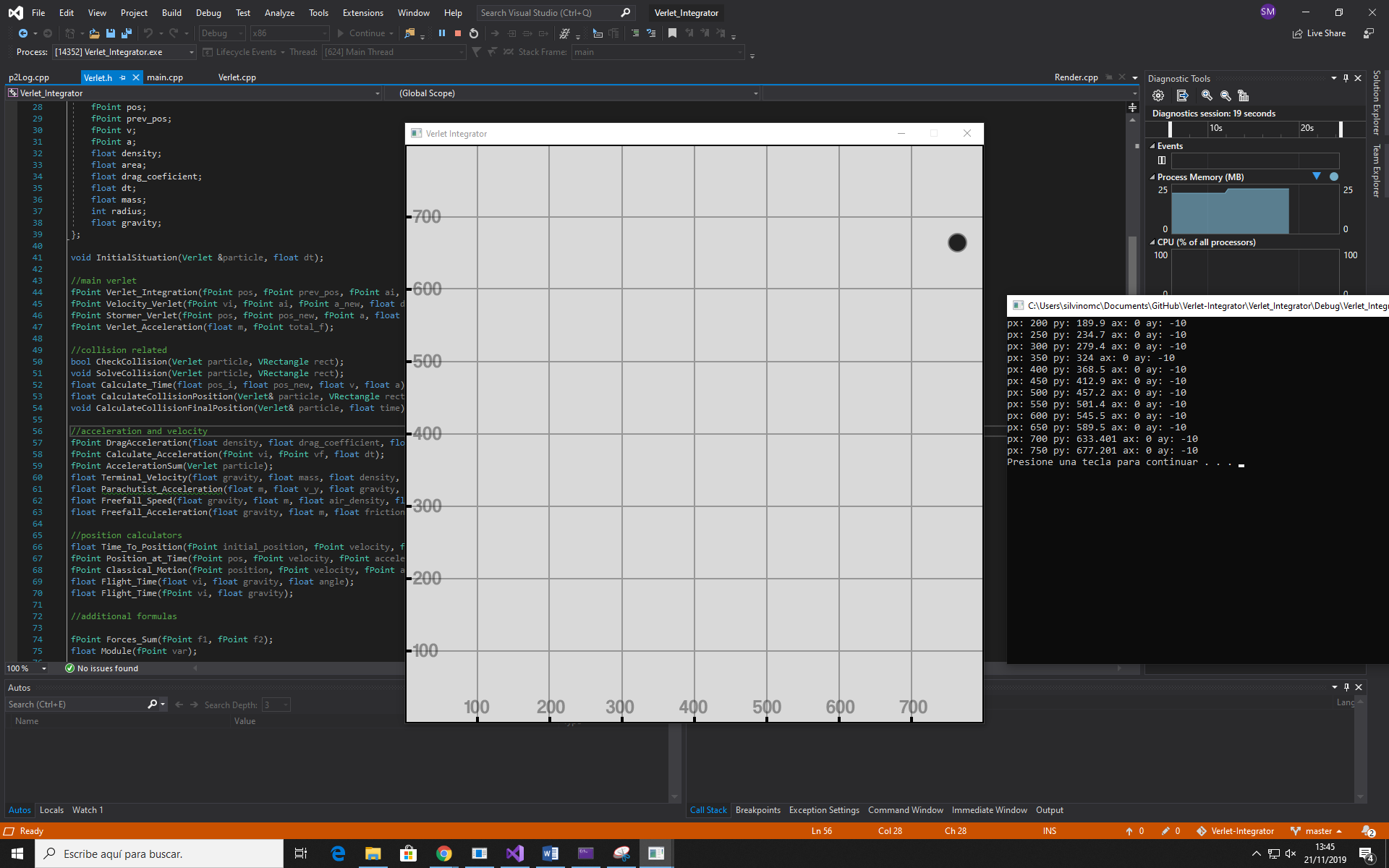
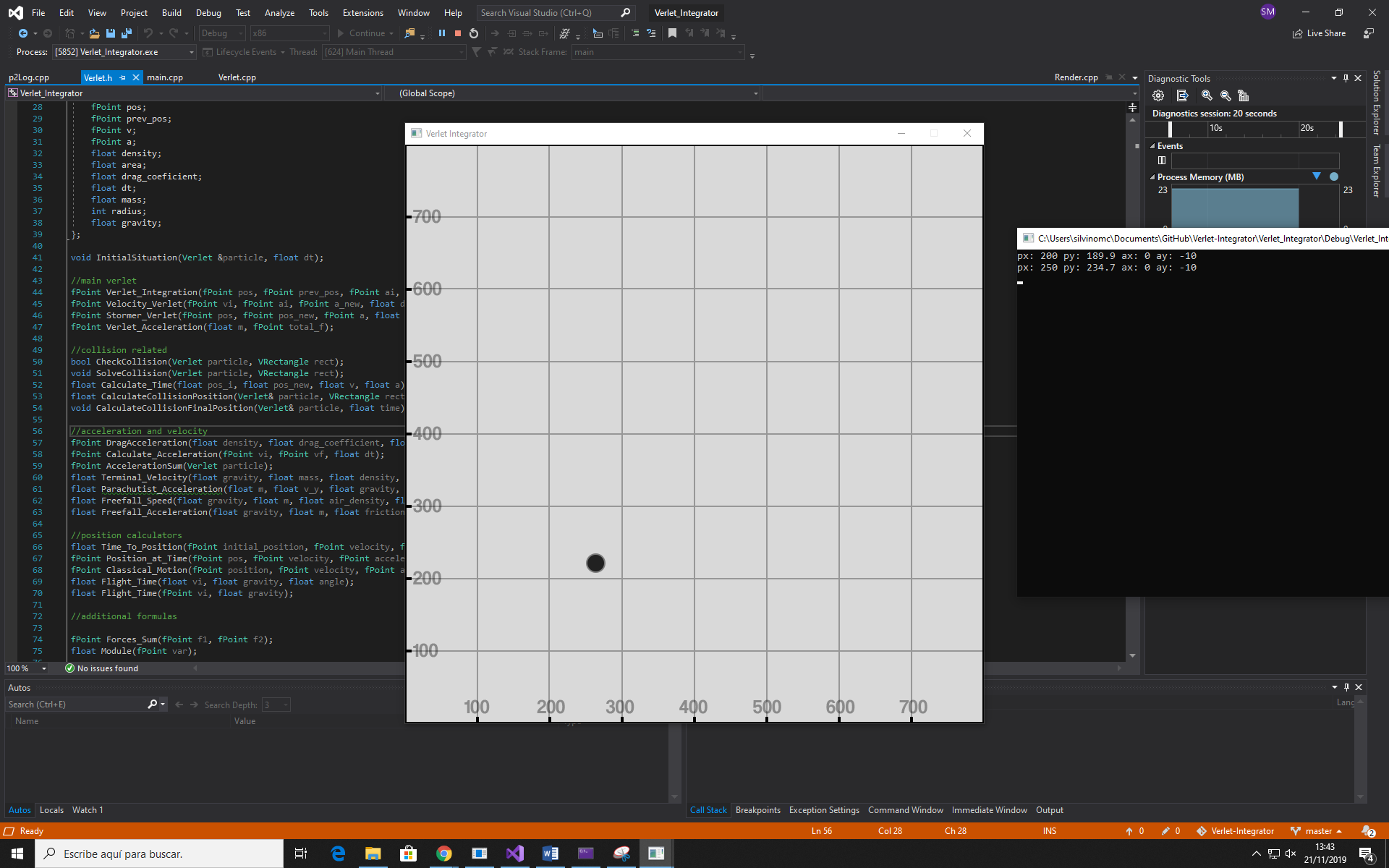
Velocidad inicial x e y: 50, 50

Gravedad: -10

Masa: 2

* Gráfico

En este caso, para poder comprobar lo que realmente hace esta opción del menú es necesario ver la compilación del código en acción. Pero, para poder mostrarlo un poco en este documento, hemos introducido la posición inicial de la partícula en base a unos valores y su posición final



# 5. Conclusiones