Simulazioni dei moti fisici elementari

In questo progetto si vogliono simulare alcuni moti elementari di punti nel piano e in tre dimensioni sfruttando le equazioni note di fisica.

```
begin
    using Plots
    using LinearAlgebra
    using PlutoUI
end
```

```
v = [5.0, 10.0]
```

```
s₀ = [0.0, 0.0]
```

```
a = [0.0, -9.8]
```

Moto rettilineo uniforme

Il moto rettilineo uniforme è uno dei due moti detti fondamentali (insieme al moto uniformemente accelerato) in quanto da essi si possono ricavare tutti gli altri. In questa sezione andrò a simulare il moto rettilineo uniforme di un punto nel piano sfruttando l'equazione oraria

$$s(t) = s_0 + v_0 t + \frac{1}{2} a t^2$$

dove s_0 è il punto di partenza del punto (e quindi un vettore bidimensionale) e v_0 rappresenta il vettore della velocità, che in questo caso è costante. Per quanto riguarda l'accelerazione, in questo caso è nulla e quindi l'equazione diventa semplicemente

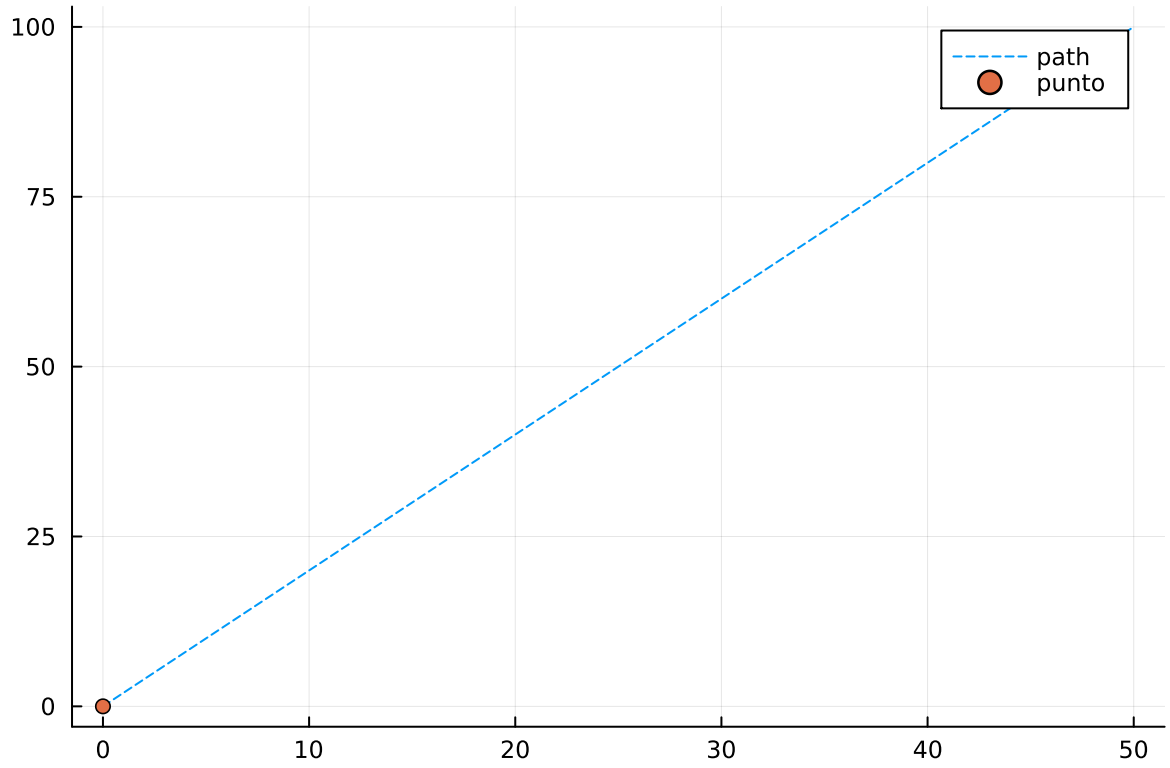
$$s(t) = s_0 + v_0 t$$

In questo caso ho adottato un'**implementazione numerica**, ossia, dato un istante di tempo iniziale t trovo la posizione del punto in ogni istante di tempo successivo ottenendo due vettori `x_coord` e `y_coord` dove `x_coord[i]` e `y_coord[i]` rappresentano le coordinate del punto all'istante i.

```
([0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0,    more ,50.0], [0.0, 1.0, 2.0, 3.0, 4.0, 5
```

```
begin
    # legge
    s(t) = v * t + s₀
    # range temporale
    T = 0:0.1:10
    # coordinate
    coord1 = s.(T)
    x_coord, y_coord = first.(coord1), last.(coord1)
end
```

$t =$



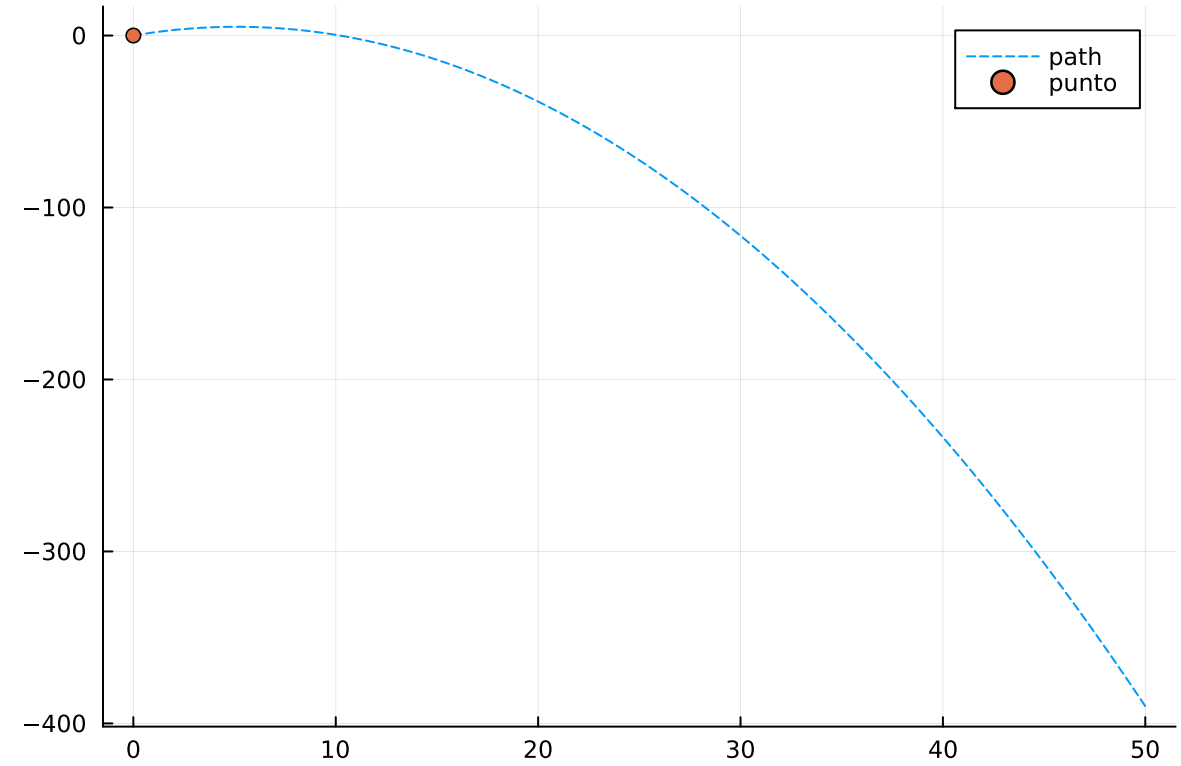

```
begin
    plot(x_coord, y_coord, ls=:dash, label="path")
    scatter!([x_coord[t]], [y_coord[t]], label="punto")
end
```

Moto rettilineo uniformemente accelerato

In questa sezione andrò ad analizzare il moto uniformemente accelerato di un punto nel piano. Le operazioni effettuate per simulare tale moto e la relativa legge oraria sono analoghe al moto rettilineo uniforme con l'unica differenza che il vettore dell'accelerazione non è nullo.

```
([0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0,  more ,50.0], [0.0, 0.951, 1.804, 2.559,
```

```
begin
    s₂(t) = (0.5).* a * t^2 + v*t + s₀
    T₂ = 0:0.1:10
    coord2 = s₂.(T₂)
    x_coord2, y_coord2 = first.(coord2), last.(coord2)
end
```




```
begin
    plot(x_coord2, y_coord2, ls=:dash, label="path")
    scatter!([x_coord2[t₂]], [y_coord2[t₂]], label="punto")
end
```

Moto orbitale

L'ultimo moto che andremo a simulare è il moto orbitale. In questo caso viene utilizzato un **approccio iterativo** sfruttando l'equazione della forza

$$F = Ma$$

e della forza di attrazione gravitazionale

$$F_g = G\frac{Mm}{d^2}$$

dove G è una costante gravitazionale, M la massa del punto fisso, m la massa del punto orbitante e d la distanza tra i due punti.

Viene introdotta la struct "Point" che definisce la struttura di un punto con le coordinate x, y e z reali (e quindi viene definito in tre dimensioni di default), un vettore velocità (`vel`) e un valore `m` reale che rappresenta la massa.

Successivamente definiamo il costruttore `Point` per semplificare la creazione di un punto con un controllo sulla dimensione del vettore velocità e, nel caso vengano inserite solo due coordinate su tre, la terza viene automaticamente impostata a 0.

Per aggiornare la posizione di un punto iterativamente viene utilizzata la funzione `update_pos` che somma al vettore posizione il vettore velocità e ottiene la nuova posizione.

La funzione `apply_force` aggiorna il vettore velocità, sfruttando la prima equazione vista sopra, aggiungendo la forza di attrazione diviso la massa alla velocità del punto.

La forza di attrazione viene calcolata dalla funzione `attracts`, che sfrutta la seconda equazione vista sopra, che, dopo un opportuno controllo sulla distanza (se prossima allo zero la velocità si annulla), calcola la forza di attrazione (e la sua direzione) del punto fisso sul punto mobile.

Infine andiamo ad applicare tutte le funzioni definite in precedenza in un'unica funzione `simulation` che prende in input due punti e il numero di iterazioni desiderate e restituisce i vettori delle coordinate x, y, e z.

```
δt = 0.01
```

```
attracts (generic function with 1 method)
```

```
begin
    mutable struct Point
        x::Real
        y::Real
        z::Real
        vel::AbstractVector
        m::Real
        function Point(x, y, z, vel, m)
            @assert length(vel) ≤ 3 "Velocity size error."
            new_vel = [0.0, 0.0, 0.0]
            for i=1:length(vel)
                new_vel[i] = vel[i]
            end
            return new(x, y, z, new_vel, m)
        end
    end

    Point(x, y, vel, m) = Point(x, y, 0.0, vel, m)
    Point(x, y, m::Real) = Point(x, y, 0.0, zeros(3), m)

    function update_pos!(p::Point)

        #Per evitare che la posizione del punto faccia salti 'troppo grandi' tra
        un'iterazione e l'altra, moltiplico per un valore δt arbitrariamente
        piccolo in modo tale da poter gestire la precisione dei valori e quindi la
        fluidità del grafico

        p.x += p.vel[1] * δt
        p.y += p.vel[2] * δt
        p.z += p.vel[3] * δt
    end

    function apply_force!(p::Point, f)
        if f == nothing
            p.vel .= 0
        else
            p.vel .+= (f/p.m)
        end
    end

    function dist(p1::Point, p2::Point)
        √((p1.x - p2.x)^2 + (p1.y - p2.y)^2 + (p1.z - p2.z)^2)
    end

    function attracts(A::Point, B::Point; G=1, ϵ=1.0)::Union{AbstractVector,
    Nothing}
        d = dist(A, B)
        if d < ϵ
            return nothing
        end

        # il vettore 'dir' rappresenta la direzione della forza applicata al punto
        mobile dal punto fisso ad ogni iterazione

        dir = [A.x - B.x, A.y - B.y, A.z - B.z]
        normalize!(dir)
        dir .* (A.m * B.m / (d^2)) .* G
```

```
    end

end
```

```
([5.0, 5.02941, 5.05822, 5.08643, 5.11404, 5.14104, 5.16742, 5.19319, 5.21834,    more ,-
```

```
begin
    function simulation(P::Point, O::Point; iter::Int)
        x_coord = [P.x]
        y_coord = [P.y]
        z_coord = [P.z]
        for _=1:iter
            f = attracts(O, P; G=0.1, ϵ=0.1)
            apply_force!(P, f)
            update_pos!(P)
            append!(x_coord, P.x)
            append!(y_coord, P.y)
            append!(z_coord, P.z)
        end
        x_coord, y_coord, z_coord
    end

    O = Point(.0, .0, .0, [.0, .0, .0], 400)
    P = Point(5.0, 10.0, 10.0, [3.0, -3.0, 1.5], 10)
    n = 1000
    x_coord3, y_coord3, z_coord3 = simulation(P, O, iter=n)

end
```

t = 1