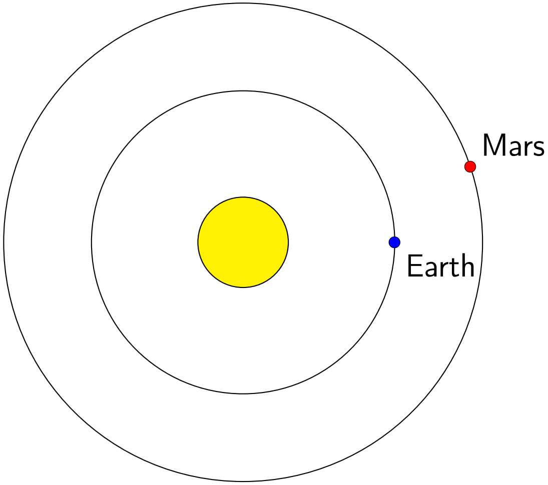
Object Oriented Orbits: a primer on Newtonian physics

Tobi Lehman

2016-03-02 Wed

Elon Musk is right, we need to go to Mars



Before we can do this, we need to simulate the orbits of Earth and Mars.

What we need

Before we can simulate orbits, we need to a few things

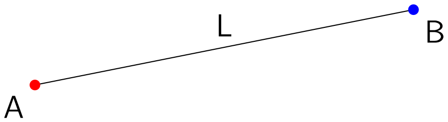
- a model of space to organize the simulated bodies
- a dynamic rule to update the locations of bodies in space

Euclid's axioms

The first complete model of space ever recorded was compiled by Euclid in ancient Greece.

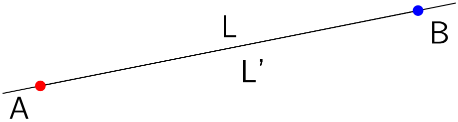
Axiom 1

Between any two points A and B, a line segment L can be drawn



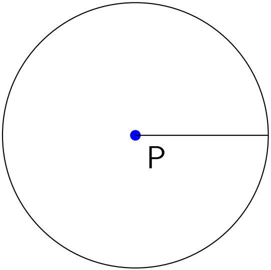
Axiom 2

A line segment L can be extended indefinitely to a larger line segment L', that contains L



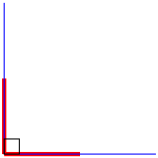
Axiom 3

A circle can be drawn at any point with any radius



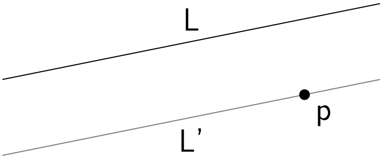
Axiom 4

All right angles are congruent



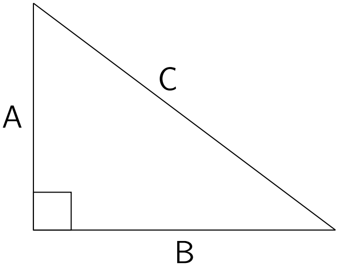
Axiom 5 (The Parallel Postulate)

Given a line L and a point p not on the line, there is exactly one line L' through p that doesn't intersect L



Theorems

From these five axioms, we can deduce many useful things, the most useful for our purposes will be the Pythagorean theorem.



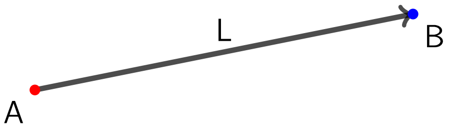
$$A^2 + B^2 = C^2$$

We can use this to compute distance

Axioms 1 and 2 and vectors

Vectors are directed line segments, which can be scaled by real numbers, so axioms 1 and 2 are relevant for vectors

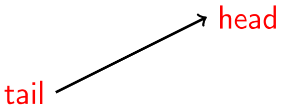
1. Given any two points A and B, a vector $\vec{v}$ exists whose tail is A and head is B
2. Given any vector $\vec{v}$ and any real number c, $c\vec{v}$ extends $\vec{v}$ by a factor of c



Some terminology

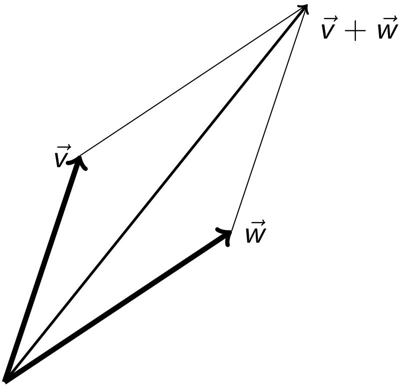
We call the initial point of a vector its tail

The final point of the vector is called its head



Vectors can be added

Given any two vectors $\vec{v}$ and $\vec{w}$ with the same tail, their sum $\vec{v} + \vec{w}$ can be visualized using a parallelogram:

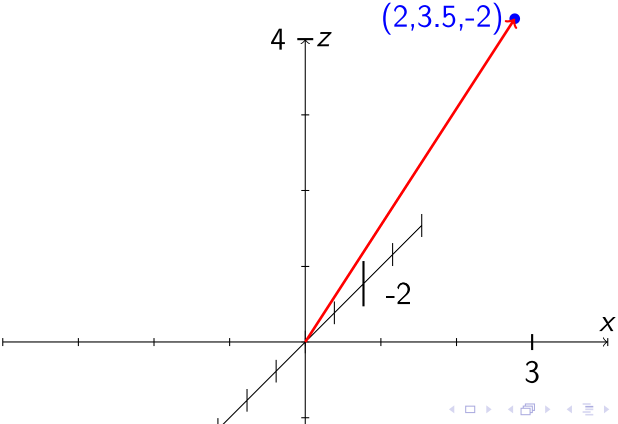


This uses axiom 5, and this operation is commutative

Vectors and coordinate systems

Given a coordinate system, we can represent vectors using pairs (2D) or triples (3D) of real numbers:

There is a special point, $\vec{0}$ which is just the origin.



Vectors have a 'dot product'

Given any two vectors $\vec{v} = (v_1, v_2, v_3)$ and $\vec{w} = (w_1, w_2, w_3)$

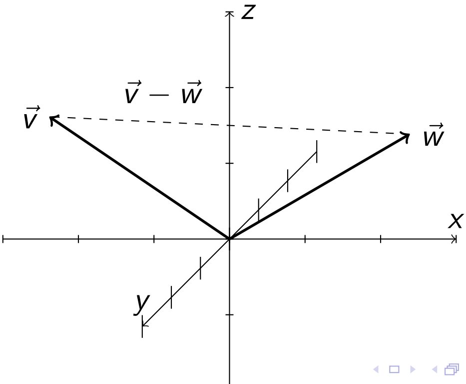
their dot product $\vec{v} \cdot \vec{w} = v_1 w_1 + v_2 w_2 + v_3 w_3$

Useful fact: $\vec{v} \cdot \vec{w} = |v||w|cos(\theta)$

That also implies that $\sqrt{\vec{v} \cdot \vec{v}}$ is the length of the vector

Distance between vectors

We are using vectors to represent points in space, so we will compute the distance between the points V and W by computing $\sqrt{(\vec{v} - \vec{w}) \cdot (\vec{v} - \vec{w})}$. This dot product magic just follows from the Pythagorean theorem.



Vectors in Ruby (components)

Now that we have a model of space, we can start writing some ruby code

- a Vector has components (the coordinates)

```ruby
class Vector
  attr_reader :components

  def initialize(components)
    @components = components
  end
end
```

Vectors in Ruby (algebra)

- a Vector can be added to another vector
- a Vector can be multiplied by a scalar

```
class Vector
  def +(vector)
    sums = components.zip(vector.components).
                          map {|(vi,wi)| vi+wi }
    Vector.new(sum)
  end

  def *(scalar)
    Vector.new(components.map{|c| scalar*c })
  end
end
```

Vectors in Ruby (equality and dot product)

- we can compare two vectors for equality
- we can take the dot product of two vectors and get the scalar

```
class Vector
  def ==(vector)
    components == vector.components
  end

  def dot(vector)
    pairs = components.zip(vector.components)
    pairs.map {|(vi,wi)| vi*wi }.
          inject(&:+)
  end
end
```

Time

Now we have a decent model of space, we can move on to the dynamic rule, it will be a way to update the state of the bodies in space over time.

Relation between position, time and velocity

We can represent the path a body takes using a function $\vec{x}(t)$.

The velocity is then just the rate of change of position with respect to time

$$\vec{v}(t) = \frac{d\vec{x}}{dt}$$

Relation between velocity and acceleration

Similarly, the acceleration is the rate of change of velocity with respect to time

$$\vec{a}(t) = \frac{d\vec{v}}{dt}$$

Newton's 1st Law states that

Bodies travel in straight lines with constant velocity
unless a force is acting on it

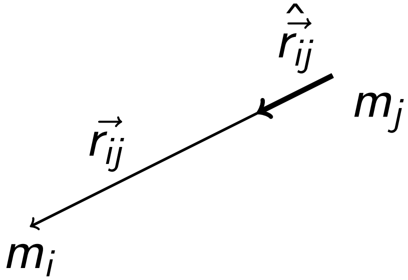
$$\vec{x}(t) = \underbrace{\vec{x}_0}_{\text{initial position}} + \underbrace{\vec{v}_0}_{\text{initial velocity}} t$$

Newton's 2nd Law states that

The vector sum of forces acting on a body is its acceleration times its mass

$$\underbrace{\sum_j \vec{F}_{ij}}_{\text{sum of all forces acting on the i-th body}} = m_i \vec{a}_i$$

Newton's Law of Universal Gravitation



$$\vec{F_{ij}} = \left(G \frac{m_i m_j}{|\vec{r_{ij}}|^2} \right) \hat{\vec{r_{ij}}}$$

Bodies in Ruby

the Body class should have a read-only mass

along with a position and a velocity

```ruby
class Body
  attr_reader :mass
  attr_accessor :position, :velocity

  def initialize(mass:, position:, velocity:)
    @mass = mass
    @position = Vector.new(position)
    @velocity = Vector.new(velocity)
  end
end
```

Forces on Bodies in Ruby

Bodies have a method to compute the gravitational force acting on it from another Body.

```
class Body
  def force_from(body)
    rvec = body.position - position
    r = rvec.norm
    rhat = rvec * (1/r)
    rhat * (Newtonian.G * mass * body.mass / r**2)
  end
end
```

the Universe

It's very big
- Douglas Adams

the Universe in Ruby

The final class will be Universe, it organizes all the bodies

```
class Universe
  attr_reader :dimensions, :bodies

  def initialize(dimensions:, bodies:)
    @dimensions = dimensions
    @bodies = bodies
  end
end
```

it also has a number of dimensions, we can use this to make sure the bodies are all in the same kind of space

the Enumerable Universe

Since force is computed pairwise, we create an iterator for pairs of distinct objects

```
class Universe
  def each_pair
    bodies.each do |body_i|
      bodies.each do |body_j|
        next if body_i == body_j
        yield [body_i, body_j]
      end
    end
  end
end
```

The evolve method

Finally, we can define the main simulation loop

```
class Universe
  def evolve(dt)
    each_pair do |(body_i, body_j)|
      force_ij = body_i.force_from(body_j)
      f_over_m = force_ij * (1.0/body_i.mass)
      body_i.velocity += f_over_m * dt
      body_i.position += body_i.velocity * dt
    end
  end
end
```

Binary Star system

Our first application is going to be simulating a binary star system, with two equal-mass stars