Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Newtonian Mechanics / Single particle mechanics

[src: NewtonianMechanics/SingleParticle.lhs] Previous: Usage Table of contents Next: Teeter

```
module NewtonianMechanics.SingleParticle where
```

%< import Calculus.SyntaxTree

```
import           Test.QuickCheck
```

Laws:

- A body remains at rest or in uniform motion unless acted upon by a force.
- If two bodies exert forces on each other, these forces are equal in magnitude and opposite in direction.

We will begin our journey into classical mechanics by studying point particles. A point particle has a mass and it's position is given as vector in three dimensions. Of course it could exist in any number of dimensions but we'll stay in three dimensions since it is more intuitive and easier to understand.

The components of the vector are functions over time that gives the particles position in each dimension, x, y, and z. Since we've already defined vectors and mathmatical functions in previous chapters we won't spend any time on them here and instead just import those two modules.

```
import           Calculus.FunExpr
import           Calculus.DifferentialCalc -- Maybe remove
import           Calculus.IntegralCalc     -- Maybe remove
import           Vector.Vector as V
```

The mass of a particle is just a scalar value so we'll model it using $FunExpr$.

```
type Mass = FunExpr
```

We combine the constructor for vectors in three dimensions with the function expressions defined in the chapter on mathmatical analysis. We'll call this new type `VectorE` to signify that it's a vector of expressions.

```
type VectorE = Vector3 FunExpr
```

Now we are ready to define what the data type for a particle is. As we previously stated a point particle has a mass, and a position given as a vector of function expressions. So our data type is simply:

```
data Particle  = P { pos  :: VectorE -- Position as a function of time, unit m
                   , mass :: Mass    -- Mass, unit kg
                   } deriving Show
```

So now we can create our particles! Let's try it out!

```
particle :: Particle
particle = P (V3 (3 * Id * Id) (2 * Id) 1) 3
```

Let's see what happens when we run this in the interpreter.

```
ghci > particle
P {pos = (((3 * id) * id) x, (2 * id) y, 1 z), mass = 3}
```

We've created our first particle! And as we can see from the print out it's accelerating by $3t^2$ in the x-dimension, has a constant velocity of $2t$ in the y-dimension, is positioned at 1 in the z-dimension, and has a mass of 3.

Velocity & Acceleration

Velocity is defined as the derivative of the position with respect to time. More formally, ($\vec{r}$ denotes the position vector):

$$\vec{v} = \frac{d\vec{r}}{dt}$$

And since the position of our particles are given as vectors we'll do the derivation component-wise. We need not worry about the details of the derivation at all since this is all take care of by the Calculus module, all we need to to is use the constructor `D` for the derivative and apply it to each of the components of the vector. The business of applying something to each component of a vector has also already been taken care of! This was the point of `vmap` to map a function over the components of the vector. So if we combine them we get a rather elegant way of computing the velocity of a particle.

```
velocity :: Particle -> VectorE
velocity = vmap D . pos
```

We can try this out in the interpreter with our newly created particle.

```
ghci > velocity particle
((D ((3 * id) * id)) x, (D (2 * id)) y, (D 1) z)
```

Not very readable but at least we can see that it correctly maps the derivative over the components of the vector.

Acceleration is defined as the derivative of the velocity with respect to time, or the second derivative of the position. More formally:

$$\vec{a} = \frac{d\vec{v}}{dt} = \frac{d^2\vec{r}}{dt^2}$$

Exercise Try to figure out how to define the function for calculating the acceleration of a particle.

▼ **Solution**

We already know how to get the velocity of a particle, so the the only step we need to take is to take the derivative of the velocity.

```
acceleration :: Particle -> VectorE
acceleration = vmap D . velocity
```

Which is the same as:

```haskell
acceleration :: Particle -> VectorE
acceleration = vmap D . vmap D . pos
```

▼ **Trivia**

Those of you familiar with functor laws will probably see that the code for calculating the acceleration could also be written as:

```haskell
acceleration :: Particle -> VectorE
acceleration = vmap (D . D) . pos
```

Forces & Newton's second law

Newton's second law states that

> A body acted upon by a force moves in such a manner that the time rate of change of the momentum equals the force.

This law expresses the relationship between force and momentum and is mathematically defined as follows:

$$\vec{F} = \frac{d\vec{p}}{dt} = \frac{d(m \cdot \vec{v})}{dt}$$

Force is a vector so let's create a type synonym to make this clearer.

```haskell
type Force = VectorE
```

The quantity $m \cdot \vec{v}$ is what we mean when we say momentum. So the law states that the net force on a particle is equal to the rate of change of the momentum with respect to time. And since the definition of acceleration is $\vec{a} = \frac{d\vec{v}}{dt}$ we can write this law in a more familiar form, namely:

$$\vec{F} = m \cdot \vec{a}$$

And thus if the particle is accelerating we can calculate the force that must be acting on it, in code this would be:

```haskell
force :: Particle -> Force
force p = vmap (* m) a
  where
    m = mass p
    a = acceleration p
```

Where the acceleration of particle is found by deriving the velocity of that same particle with respect to t:

Kinetic energy

Kinetic stands for motion, so the kinetic energy of a particle is the energy it gains through movement. Energy is a scalar quantity so we'll model it using a $FunExpr$.

```haskell
type Energy = FunExpr
```

In classical mechanics the kinetic energy of particle is equal to $\frac{1}{2}$ of the product of its mass and the square of its velocity. In mathmatical notation this would be.

$$E_k = mv^2$$

But what does it mean to take the square of speed? We know of an operator to take the square of a number, but velocity is not a number it's a vector. So there seems to be a double meaning to what taking the square of something actually means depending on the type of the expression.

More concrete what we actually mean when we say `the square of vector'' is to take the dot product of the vector with itself. This may seem fairly straightforward and obvious on paper but when we start working with types this overloading of the word`square'' looks a bit strange. For numbers it has the type

```haskell
square :: Num a => a -> a
```

but for vectors it has the type

```haskell
square :: Num a => Vector a -> a
```

same name, vastly different meanings. If we were to encode this double meaning of the function we would have to create a type class which would encode this ability to be squared and then make Num and Vector instances of this class.

But enough yammering about types and double meanings! Let's actually define the function for taking the square of a vector and move on with our lives.

```
square :: VectorE -> FunExpr
square v = dotProd v v
```

Pretty straightforward. So with this out of the way we can now define the function for calculating the kinetic energy of a particle, again this is almost the same as just writing down the mathmatical formula with the added benefit of types.

```
kineticEnergy :: Particle -> Energy
kineticEnergy p = Const 0.5 * m * v2
  where
    m  = mass p
    v  = velocity p
    v2 = square v
```

Potential energy

In classical mechanics potential energy is defined as the energy possed by a particle because of its position relative to other particles, its electrical charge and other factors. This means that to actually calculate the potential energy of a particle we'd have to take into account all other particles that populate the system no matter how far apart they are.

Potential energy near Earth

Thankfully if we're close to the Earths gravitational field it's ok to simplify this problem and only take into account the Earths gravitational pull since all other factors are negliable in comparison.

The potential energy for particles affected by gravity is defined with mathmatical notation as:

$$E_p = m \cdot g \cdot h$$

where m is the mass of the particle, h its height, and g is the acceleration due to gravity (9.82 m/s^2).

```
potentialEnergy :: Particle -> Energy
potentialEnergy p = m * g * h
  where
    m          = mass p
    g          = Const 9.82
    (V3 _ _ h) = pos p
```

Work

If a constant force $\vec{F}$ is applied to a particle that moves from position $\vec{r_1}$ to $\vec{r_2}$ then the *work* done by the force is defined as the dot product of the force and the vector of displacement. In mathmatical notation this is written as

$$W = \vec{F} \cdot \Delta\vec{r}$$

where $\Delta\vec{r} = \vec{r_2} - \vec{r_1}$.

The work-energy theorem states that for a particle of constant mass *m*, the total work *W* done on the particle as it moves from position r_1 to r_2 is equal to the change in kinetic energy E_k of the particle:

$$W = \Delta E_K = E_{k,2} - E_{k,1} = \frac{1}{2}m(\vec{v_2}^2 - \vec{v_1}^2)$$

Let's codify this theorem:

PS: This used to work just fine, but it no longer does since the switch to FunExpr. Problem probably lies somewhere in SyntaxTree

```
prop_WorkEnergyTheorem :: Mass -> VectorE -> VectorE -> IO Bool
prop_WorkEnergyTheorem m v1 v2 = prettyEqual deltaEnergy (kineticEnergy
  displacedParticle)
  where
    particle1 = P v1 m -- | Two particles with the same mass
    particle2 = P v2 m -- | But different position vector
    -- |          E_k,2                      - E_k,1
```

```
    deltaEnergy = kineticEnergy particle2 - kineticEnergy particle1
    displacedParticle = P (v2 - v1) m


-- Test values
v1 = V3 (3 :* Id) (2 :* Id) (1 :* Id)
v2 = V3 0 0 (5 :* Id)
v3 = V3 0 (3 :* Id) 0 :: VectorE
v4 = V3 2 2 2 :: VectorE
m  = 5
p1 = P v1 m
p2 = P v2 m
dE = kineticEnergy p2 - kineticEnergy p1
p3 = P (v2 - v1) m
```

Law of universal gravitation

Newton's law of universal gravitation states that a particle attracts every other particle in the universe with a force that is directly proportional to the product of their masses, and is inversely proportional to the square of the distance between their centers.

This means that every particle with mass attracts every other particle with mass by a force pointing along the line intersecting both points.

There is an equation for calculating the magnitude of this force which states with math what we stated in words above:

$$F = G\frac{m_1 m_2}{r^2}$$

Where *F* is the maginitude of the force, *m1* and *m2* are the masses of the objects interacting, *r* is the distance between the centers of the masses and *G* is the gravitational constant.

The gravitational constant has been finely approximated through experiments and we can state it in our code like this:

```
type Constant = FunExpr

gravConst :: Constant
gravConst = 6.674 * (10 ** (-11))
```

Now we can codify the law of universal gravitation using our definition of particles.

```
lawOfUniversalGravitation :: Particle -> Particle -> FunExpr
lawOfUniversalGravitation p1 p2 = gravConst * ((m_1 * m_2) / r2)
  where
    m_1 = mass p1
    m_2 = mass p2
    r2 = square $ pos p2 - pos p1
```

If a particles position is defined as a vector representing its displacement from some origin O, then its heigh should be x. Or maybe it should be the magnitude of the vector, if the gravitational force originates from O. Hmmm

This seems weird since I don't know what the frame of reference is…

```
potentialEnergy :: Particle -> Energy
potentialEnergy p = undefined
  where
    m          = mass p
    (V3 x _ _) = pos p
```