Learn Physics by Programming in Haskell

Scott N. Walck
Department of Physics
Lebanon Valley College
Annville, Pennsylvania, USA

May 25, 2014

Physics 261: Intro to Computational Physics

- Prereq: 1 year of intro physics, 1 semester of calculus
- No previous programming experience expected
- Goal is to deepen understanding of physics by expressing physics in a new language
- Spend about seven weeks learning a subset of Haskell
- Code for later parts of the course:
  `cabal install learn-physics`

Types and higher-order functions help you learn physics

- expose the structure of Newtonian mechanics
- clarify and organize ideas in electromagnetic theory

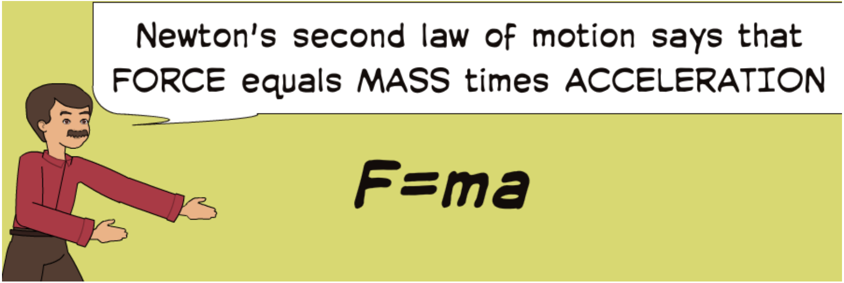
A type for 3-dimensional vectors

```
data Vec = Vec { xComp :: Double
               , yComp :: Double
               , zComp :: Double
               } deriving (Eq)

(^+^) :: Vec -> Vec -> Vec
Vec ax ay az ^+^ Vec bx by bz
    = Vec (ax+bx) (ay+by) (az+bz)

(*^) :: Double -> Vec -> Vec
c *^ Vec ax ay az = Vec (c*ax) (c*ay) (c*az)
```

Function types clarify our thinking

Function	Description	Type
(^+^)	vector addition	Vec -> Vec -> Vec
(^-^)	vector subtraction	Vec -> Vec -> Vec
(*^)	scalar multiplication	Double -> Vec -> Vec
(^*)	scalar multiplication	Vec -> Double -> Vec
(^/)	scalar division	Vec -> Double -> Vec
(<.>)	dot product	Vec -> Vec -> Double
(><)	cross product	Vec -> Vec -> Vec
magnitude	magnitude	Vec -> Double
zeroV	zero vector	Vec
iHat	unit vector	Vec
negateV	vector negation	Vec -> Vec
xComp	vector component	Vec -> Double
sumV	vector sum	[Vec] -> Vec

What could be simpler?



Newton's Second Law is a Differential Equation

Even for a single object,

$$F = ma$$

is shorthand for

$$F_{\mathrm{net}}\left(t, x, \frac{dx}{dt}\right) = m\frac{d^2x}{dt^2} \qquad \text{in one dimension,}$$

or

$$\vec{F}_{\mathrm{net}}\left(t, \vec{r}, \frac{d\vec{r}}{dt}\right) = m\frac{d^2\vec{r}}{dt^2} \qquad \text{in three dimensions.}$$

For multiple particles, Newton's 2nd law is a set of coupled differential equations.

Euler Method for Newton's Second Law

The second-order differential equation

$$\frac{d^2\vec{r}}{dt^2} = \vec{a}\left(t, \vec{r}, \frac{d\vec{r}}{dt}\right)$$

has the following state update rule.

Over a short time Δt,

$$(t, \vec{r}, \vec{v}) \rightarrow (t', \vec{r}', \vec{v}')$$

where

$$\begin{aligned} t' &= t + \Delta t \\ \vec{r}' &= \vec{r} + \vec{v}\Delta t \\ \vec{v}' &= \vec{v} + \vec{a}(t, \vec{r}, \vec{v})\Delta t. \end{aligned}$$

Mechanics of One Object in Three Dimensions

```
type Time         = Double
type Displacement = Vec
type Velocity     = Vec
type State        = (Time, Displacement, Velocity)

type AccelerationFunction = State -> Vec

eulerStep :: AccelerationFunction
          -> Double -> State -> State
eulerStep a dt (t,r,v) = (t',r',v')
    where
      t' = t + dt
      r' = r ^+^ v ^* dt
      v' = v ^+^ a(t,r,v) ^* dt
```

Different problems have different acceleration functions

Satellite orbiting the Earth:

$$\vec{F} = -\frac{GMm}{r^2}\hat{r} \qquad \vec{a} = -\frac{GM}{r^2}\hat{r}$$

```
satellite :: AccelerationFunction
satellite (t,r,v)
    = 6.67e-11 * 5.98e24 / magnitude r ^ 2 *^ u
      where
        u = negateV r ^/ magnitude r
```

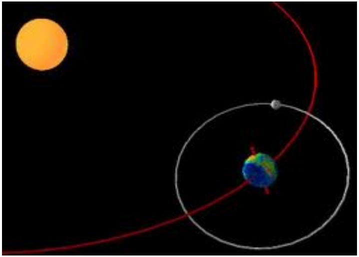
Different problems have different acceleration functions

Damped, driven harmonic oscillator:

```
dampedDrivenOsc :: Double   -- damping constant
                -> Double   -- drive amplitude
                -> Double   -- drive frequency
                -> AccelerationFunction
dampedDrivenOsc beta driveAmp omega (t,r,v)
    = (forceDamp ^+^ forceDrive ^+^ forceSpring) ^/ mass
      where
        forceDamp   = (-beta) *^ v
        forceDrive  = driveAmp * cos (omega * t) *^ iHat
        forceSpring = (-k) *^ r
        mass        = 1
        k           = 1  -- spring constant
```

Multiple Particles

```
type SystemState = (Time, [(Displacement, Velocity)])
type SystemAccFunc = SystemState -> [Vec]
```



Example: Elastic string is modelled as a collection of 100 masses connected by springs.

Structure of Mechanics

1. Choose a *type* to represent the state space for the problem.

```
type State       = (Time, Vec, Vec)
type SystemState = (Time, [(Vec, Vec)])
```

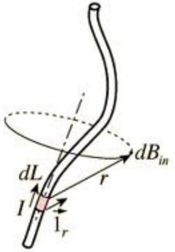
2. Describe how the state changes in time.

```
type AccelerationFunction = State -> Vec
eulerStep :: AccelerationFunction
          -> Double -> State -> State
```

3. Give an initial state for the system.

```
initialState :: State
```

Magnetic Field produced by a Wire



Magnetic field at $\vec{r}$ produced by a current I flowing along a curve C is

$$\vec{B}(\vec{r}) = \frac{\mu_0 I}{4\pi} \int_C \frac{d\vec{l}' \times (\vec{r} - \vec{r}')}{|\vec{r} - \vec{r}'|^3}. \qquad \text{(Biot-Savart law)}$$

Data types for curve, scalar field, vector field

```
data Curve
    = Curve { curveFunc          :: Double -> Position
            , startingCurveParam :: Double
            , endingCurveParam   :: Double }

loopCurve :: Curve
loopCurve = Curve (\phi -> cyl 1 phi 0) 0 (2*pi)

type ScalarField = Position -> Double
type VectorField = Position -> Vec
type Field v     = Position -> v
```

Integration is a Higher-Order Function

A general purpose "crossed line integral"

$$\int_C \vec{F}(\vec{r}') \times dl'$$

```haskell
-- | Calculates integral vf x dl over curve.
crossedLineIntegral
    :: Int          -- ^ number of intervals
    -> VectorField  -- ^ vector field
    -> Curve        -- ^ curve to integrate over
    -> Vec          -- ^ vector result
```

Type signature clarifies purpose

$$\vec{B}(\vec{r}) = \frac{\mu_0 I}{4\pi} \int_C \frac{d\vec{l'} \times (\vec{r} - \vec{r'})}{|\vec{r} - \vec{r'}|^3} \qquad \text{(Biot-Savart law)}$$

```
bFieldFromLineCurrent
    :: Current        -- ^ current (in Amps)
    -> Curve          -- ^ geometry of the line current
    -> VectorField    -- ^ magnetic field (in Tesla)
bFieldFromLineCurrent i c r
    = k *^ crossedLineIntegral 1000 integrand c
      where
        k = 1e-7  -- mu0 / (4 * pi)
        integrand r' = (-i) *^ d ^/ magnitude d ** 3
            where
              d = displacement r' r
```

Thanks for Listening!

```haskell
{-# OPTIONS_GHC -Wall #-}

module Main where

main :: IO ()
main = putStrLn "Types and higher-order functions" >>
       putStrLn "help you learn physics."
```