1 Vectors and matrices

In vectors we introduced the concept of a vector. A vector mathematically is a geometric object with two attributes a magnitude and a direction. (The direction is undefined in the case the magnitude is 0.) Vectors are typically visualized with an arrow, where the anchoring of the arrow is context dependent and is not particular to a given vector.

Vectors and points are related, but distinct. Let's focus on 3 dimensions. Mathematically, the notation for a point is $p = (x, y, z)$ while the notation for a vector is $\vec{v} = \langle x, y, z \rangle$. The ith component in a vector is referenced by a subscript: v_i. With this, we may write a typical vector as $\vec{v} = \langle v_1, v_2, \ldots, v_n \rangle$ and a vector in $n = 3$ as $\vec{v} = \langle v_1, v_2, v_3 \rangle$. The different grouping notation distinguishes the two objects. As another example, the notation $\{x, y, z\}$ indicates a set. Vectors and points may be *identified* by anchoring the vector at the origin. Set's are quite different from both, as the order of their entries is not unique.

In `Julia`, the notation to define a point and a vector would be identical, using square brackets to group like-type values: `[x, y, z]`. The notation `(x,y,z)` would form a tuple which though similar in many respects, tuples do not have the operations associated with a point or a vector defined for them.

The square bracket constructor has some subtleties:

- `[x,y,z]` calls `vect` and creates a 1-dimensional array
- `[x; y; z]` calls `vcat` to **v**ertically con**cat**enate values together. With simple numbers the two are identical, but not in other cases. (For example, is `A` is a matrix then `[A,A]` is a vector of matrices, `[A;A]` is a matrix combined from the two pieces.
- `[x y z]` calls `hcat` to **h**orizontally con**cat**enate values together. If `x`, `y` are numbers then `[x y]` is *not* a vector, but rather a 2D array with a single row and two columns.
- finally `[w x; y z]` calls `hvcat` to horizontally and vertically concatenate values together to create a container in two dimensions, like a matrix.

(A vector, mathematically, is a one-dimensional collection of numbers, a matrix a two-dimensional *rectangular* collection of numbers, and an array an n-dimensional rectangular-like collection of numbers. In `Julia`, a vector can hold a collection of objects of arbitrary type, though generally all of the same type.)

1.1 Vector addition, scalar multiplication

As seen earlier, vectors have some arithmetic operations defined for them. As a typical use of vectors, mathematically, is to collect the x, y, and z (in 3D) components together, operations like addition and subtraction operate component wise. With this, addition can be visualized geometrically: put the tail of $\vec{v}$ at the tip of $\vec{u}$ and draw a vector from the tail of $\vec{u}$ to the tip of $\vec{v}$ and you have $\vec{u}+\vec{v}$. This is identical by $\vec{v}+\vec{u}$ as vector addition is commutative. Unless $\vec{u}$ and $\vec{v}$ are parallel or one has 0 length, the addition will create a vector with a different direction from the two.

Another operation for vectors is *scalar* multiplication. Geometrically this changes the magnitude, but not the direction of a vector, when the *scalar* is positive. Scalar multiplication is

defined component wise, like addition so the ith component of $c\vec{v}$ is c times the ith component of $\vec{v}$. When the scalar is negative, the direction is "reversed."

To illustrate we first load our package and define two 3D vectors:

```
using CalculusWithJulia
u, v = [1, 2, 3], [4, 3, 2]
```

```
([1, 2, 3], [4, 3, 2])
```

The sum is component-wise summation (`1+4, 2+3, 3+2`):

```
u + v
```

```
3-element Array{Int64,1}:
 5
 5
 5
```

For addition, as the components must pair off, the two vectors being added must be the same dimension.

Scalar multiplication by `2`, say, multiplies each entry by `2`:

```
2 * u
```

```
3-element Array{Int64,1}:
 2
 4
 6
```

1.2 The length and direction of a vector

If a vector $\vec{v} = \langle v_1, v_2, \dots, v_n \rangle$ then the *norm* (also Euclidean norm or length) of $\vec{v}$ is defined by:

$$\|\vec{v}\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}.$$

The definition of a norm leads to a few properties. First, if c is a scalar, $\|c\vec{v}\| = |c|\|\vec{v}\|$ - which says scalar multiplication by c changes the length by $|c|$. (Sometimes, scalar multiplication is described as "scaling by....") The other property is an analog of the triangle inequality, in which for any two vectors $\|\vec{v} + \vec{w}\| \leq \|\vec{v}\| + \|\vec{w}\|$. The right hand side is equal only when the two vectors are parallel and addition is viewed as laying them end to end.

A vector with length 1 is called a *unit* vector. Dividing a non-zero vector by its norm will yield a unit vector, a consequence of the first property above. Unit vectors are often written with a "hat:" $\hat{v}$.

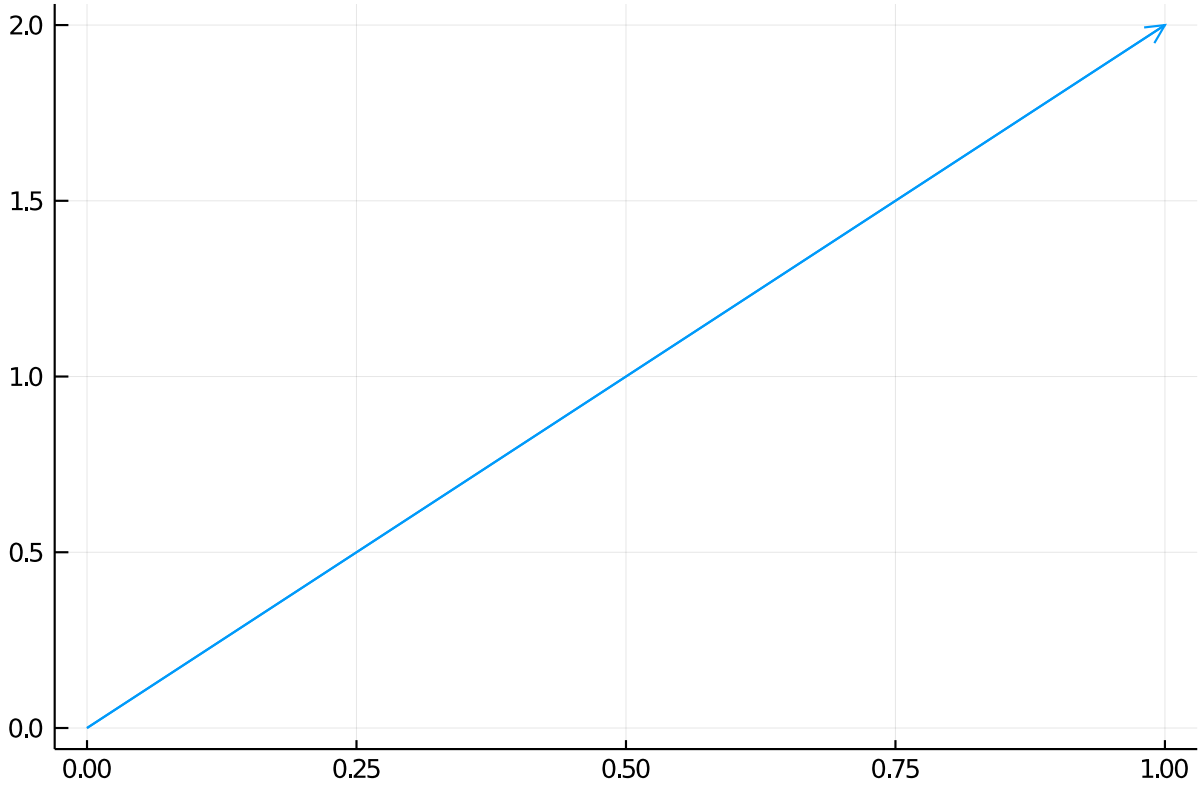
The direction indicated by $\vec{v}$ can be visualized as an angle in 2 or 3 dimensions, but in higher dimensions, visualization is harder. For 2-dimensions, we might associate with a vector, it's unit vector. This in turn may be identified with a point on the unit circle, which from basic trigonometry can be associated with an angle. Something similar, can be done in 3 dimensions, using two angles. However, the "direction" of a vector is best thought of in terms of its associated unit vector. With this, we have a decomposition of a vector $\vec{v}$ into a magnitude scalar and a direction when we write $\vec{v} = \|\vec{v}\| \cdot (\vec{v}/\|\vec{v}\|) = \|\vec{v}\|\hat{v}$.

1.3 Visualization of vectors

Vectors may be visualized in 2 or 3 dimensions using `Plots`. In 2 dimensions, the `quiver` function may be used. To graph a vector, it must have its tail placed at a point, so two values are needed.

To plot `u=[1,2]` from `p=[0,0]` we have the following usage:

```
using Plots
gr()        # better arrows than plotly()
quiver([0],[0], quiver=([1],[2]))
```



The cumbersome syntax is typical here. We naturally describe vectors and points using `[a,b,c]` to combine them, but the plotting functions want to plot many such at a time and expect vectors containing just the `x` values, just the `y` values, etc. The above usage looks a bit odd, as these vectors of `x` and `y` values have only one entry. Converting from the one representation to the other requires reshaping the data, which we will do with the following function from the `CalculusWithJulia` package:

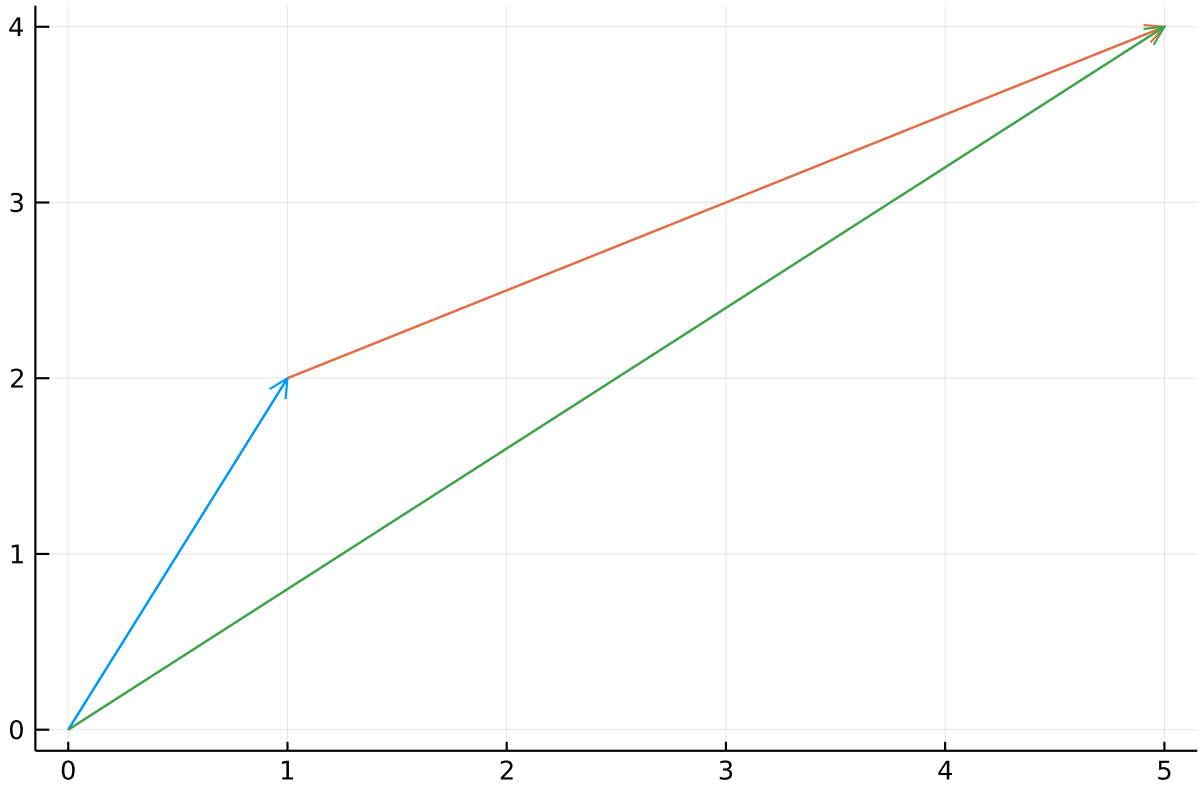
```
unzip(vs) = Tuple(eltype(first(vs))[xyz[j] for xyz in vs] for j in eachindex(first(vs)))
```

This takes a vector of vectors, and returns a tuple containing the `x` values, the `y` values, etc. So if `u=[1,2,3]`, the `unzip([u])` becomes `([1],[2],[3])`. And if `v=[4,5,6]`, then `unzip([u,v])` becomes `([1,4],[2,5],[3,6])`, etc. (The `zip` function in base does essentially the reverse operation.)

With `unzip` defined, we can plot a 2-D vector `v` anchored at point `p` through `quiver(unzip([p])..., quiver=unzip([v]))`.

To illustrate, the following defines 3 vectors (the third through addition), then graphs all three, though in different starting points to emphasize the geometric interpretation of vector addition.

```
u = [1, 2]
v = [4, 2]
w = u + v
p = [0,0]
quiver(unzip([p])..., quiver=unzip([u]))
quiver!(unzip([u])..., quiver=unzip([v]))
quiver!(unzip([p])..., quiver=unzip([w]))
```



Plotting a 3-d vector is not supported in all toolkits with `quiver`. A line segment may be substituted and can be produced with `plot(unzip([p,p+v])...)`. To avoid all these details, the `CalculusWithJulia` provides the `arrow!` function to *add* a vector to an existing plot. The function requires a point, `p`, and the vector, `v`:

With this, the above simplifies to:

```
plot(legend=false)
arrow!(p, u)
arrow!(u, v)
arrow!(p, w)
```

The distinction between a point and a vector within `Julia` is only mental. We use the same storage type. Mathematically, we can **identify** a point and a vector, by considering the vector with its tail placed at the origin. In this case, the tip of the arrow is located at the point. But this is only an identification, though a useful one. It allows us to "add" a point and a vector (e.g., writing $P + \vec{v}$) by imagining the point as a vector anchored at the origin.

To see that a unit vector has the same "direction" as the vector, we might draw them with different widths:

```
using LinearAlgebra
v = [2, 3]
u = v / norm(v)
p = [0, 0]
plot(legend=false)
arrow!(p, v)
arrow!(p, u, linewidth=5)
```

The `norm` function is in the standard library, `LinearAlgebra`, which must be loaded first through the command `using LinearAlgebra`. (Though here it is redundant, as that package is loaded when the `CalculusWithJulia` package is loaded.)

1.4 Aside: review of `Julia`'s use of dots to work with containers

`Julia` makes use of the dot, ".", in a few ways to simplify usage when containers, such as vectors, are involved:

- **Splatting**. The use of three dots, "`...`", to "splat" the values from a container like a vector (or tuple) into *arguments* of a function can be very convenient. It was used above in the definition for the `arrow!` function: essentially `quiver!(unzip([p])..., quiver=unzip([v]))`. The `quiver` function expects 2 (or 3) arguments describing the `xs` and `ys` (and sometimes `zs`). The `unzip` function returns these in a container, so splatting is used to turn the values in the container into distinct arguments of the function. Whereas the `quiver` argument expects a tuple of vectors, so no splatting is used for that part of the definition. Another use of splatting we will see is with functions of vectors. These can be defined in terms of the vector's components or the vector as a whole, as below:

```
f(x,y,z) = x^2 + y^2 + z^2
f(v) = v[1]^2 + v[2]^2 + v[3]^2
```

```
f (generic function with 2 methods)
```

The first uses the components and is arguably, much easier to read. The second uses indexing in the function body to access the components. Both uses have their merits. If a function is easier to write in terms of its components, but an interface expects a vector of components as it argument, then splatting can be useful, to go from one style to another, similar to this:

```
g(x,y,z) =  x^2 + y^2 + z^2
g(v) = g(v...)
```

```
g (generic function with 2 methods)
```

The splatting will mean `g(v)` eventually calls `g(x,y,z)` through `Julia`'s multiple dispatch machinery when `v = [x,y,z]`.

(The three dots can also appear in the definition of the arguments to a function, but there the usage is not splatting but rather a specification of a variable number of arguments.)

- **Broadcasting**. For a univariate function, `f`, and vector, `xs`, the call `f.(xs)` *broadcasts* `f` over each value of `xs` and returns a container holding all the values. This is a compact alternative to a comprehension when a function is defined. The `map` function is similar. When `f` depends on more than one value, broadcasting can still be used: `f.(xs, ys)` will broadcast `f` over values formed from *both* `xs` and `ys`. The `map` function is similar, but broadcasting has the extra feature of attempting to match up the shapes of `xs` and `ys` when they are not identical. (See the help page for `broadcast` for more details.)

For example, if `xs` is a vector and `ys` a scalar, then the value in `ys` is repeated many times to match up with the values of `xs`. Or if `xs` and `ys` have different dimensions, the values of one will be repeated. Consider this:

```
xs = ys = [0, 1]
f(x,y) = x + y
f.(xs, ys)
```

```
2-element Array{Int64,1}:
 0
 2
```

This matches `xs` and `ys` to pass `(0,0)` and then `(1,1)` to `f`, returning `0` and `2`. Now consider

```
xs = [0, 1]; ys = [0 1]  # xs is a column vector, ys a row vector
f.(xs, ys)
```

```
2×@*(2 Array(*@{Int64,2}:
 0  1
 1  2
```

The two dimensions are different so for each value of `xs` the vector of `ys` is broadcast. This returns a matrix now.

At times using the "apply" notation: `x |> f`, in place of using `f(x)` is useful, as it can move the wrapping function to the right of the expression. To broadcast, `.|>` is available.

At times the automatic broadcasting is not as desired. A case involving "pairs" will come up where we want to broadcast the pair as a whole, not the two sides.

1.5 The dot product

There is no concept of multiplying two vectors, or for that matter dividing two vectors. However, there are two operations between vectors that are somewhat similar to multiplication, these being the dot product and the cross product. Each has an algebraic definition, but their geometric properties are what motivate their usage. We begin by discussing the dot product.

The dot product between two vectors can be viewed algebraically in terms of the following product. If $\vec{v} = \langle v_1, v_2, \ldots, v_n \rangle$ and $\vec{w} = \langle w_1, w_2, \ldots, w_n \rangle$, then the *dot product* of $\vec{v}$ and $\vec{w}$ is defined by:

$$\vec{v} \cdot \vec{w} = v_1 w_1 + v_2 w_2 + \cdots + v_n w_n.$$

From this, we can see the relationship between the norm, or Euclidean length of a vector: $\vec{v} \cdot \vec{v} = \|\vec{v}\|^2$. We can also see that the dot product is commutative, that is $\vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{v}$.

The dot product has an important geometrical interpolation. Two (non-parallel) vectors will lie in the same "plane", even in higher dimensions. Within this plane, there will be an angle between them within $[0, \pi]$. Call this angle θ. (This means the angle between the two vectors is the same regardless of their order of consideration.) Then

$$\vec{v} \cdot \vec{w} = \|\vec{v}\| \|\vec{w}\| \cos(\theta).$$

If we denoted $\hat{v} = \vec{v}/\|\vec{v}\|$, the unit vector in the direction of $\vec{v}$, then by dividing, we see that $\cos(\theta) = \hat{v} \cdot \hat{w}$. That is the angle does not depend on the magnitude of the vectors involved.

The dot product is computed in `Julia` by the `dot` function, which is in the `LinearAlgebra` package of the standard library. This must be loaded (as above) before its use either directly or through the `CalculusWithJulia` package:

```
u = [1, 2]
v = [2, 1]
dot(u, v)
```

4

> **note**
>
> Note
>
> In `Julia`, the unicode operator entered by `\cdot[tab]` can also be used to mirror the math notation:

```
u ⋅ v   # u \cdot[tab] v
```

4

Continuing, to find the angle between $\vec{u}$ and $\vec{v}$, we might do this:

```
ctheta = dot(u/norm(u), v/norm(v))
acos(ctheta)
```

0 . 6 4 3 5 0 1 1 0 8 7 9 3 2 8 4 5

The cosine of $\pi/2$ is 0, so two vectors which are at right angles to each other will have a dot product of 0:

```
u = [1, 2]
v = [2, -1]
u ⋅ v
```

0

In two dimensions, we learn that a perpendicular line to a line with slope m will have slope $-1/m$. From a 2-dimensional vector, say $\vec{u} = \langle u_1, u_2 \rangle$ the slope is u_2/u_1 so a perpendicular vector to $\vec{u}$ will be $\langle u_2, -u_1 \rangle$, as above. For higher dimensions, where the angle is harder to visualize, the dot product defines perpendicularness, or *orthogonality*.

For example, these two vectors are orthogonal, as their dot product is 0, even though we can't readily visualize them:

```
u = [1, 2, 3, 4, 5]
v = [-30, 4, 3, 2, 1]
u ⋅ v
```

0

Projection From right triangle trigonometry, we learn that $\cos(\theta) = \text{adjacent/hypotenuse}$. If we use a vector, $\vec{h}$ for the hypotenuse, and $\vec{a} = \langle 1, 0 \rangle$, we have this picture:

```
h = [2, 3]
a = [1, 0]  # unit vector
h_hat = h / norm(h)
theta = acos(h_hat ⋅ a)

plot(legend=false)
arrow!([0,0], h)
arrow!([0,0], norm(h) * cos(theta) * a)
arrow!([0,0], a, linewidth=3)
```

We used vectors to find the angle made by `h`, and from there, using the length of the hypotenuse is `norm(h)`, we can identify the length of the adjacent side, it being the length of the hypotenuse times the cosine of θ. Geometrically, we call the vector `norm(h) * cos(theta) * a` the *projection* of $\vec{h}$ onto $\vec{a}$, the word coming from the shadow $\vec{h}$ would cast on the direction of $\vec{a}$ were there light coming perpendicular to $\vec{a}$.

The projection can be made for any pair of vectors, and in any dimension $n > 1$. The projection of $\vec{u}$ on $\vec{v}$ would be a vector of length $\vec{u}$ (the hypotenuse) times the cosine of the angle in the direction of $\vec{v}$. In dot-product notation:

$$proj_{\vec{v}}(\vec{u}) = \|\vec{u}\| \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|\|\vec{v}\|} \frac{\vec{v}}{\|\vec{v}\|}.$$

This can simplify. After cancelling, and expressing norms in terms of dot products, we have:

$$proj_{\vec{v}}(\vec{u}) = \frac{\vec{u} \cdot \vec{v}}{\vec{v} \cdot \vec{v}} \vec{v} = (\vec{u} \cdot \hat{v})\hat{v},$$

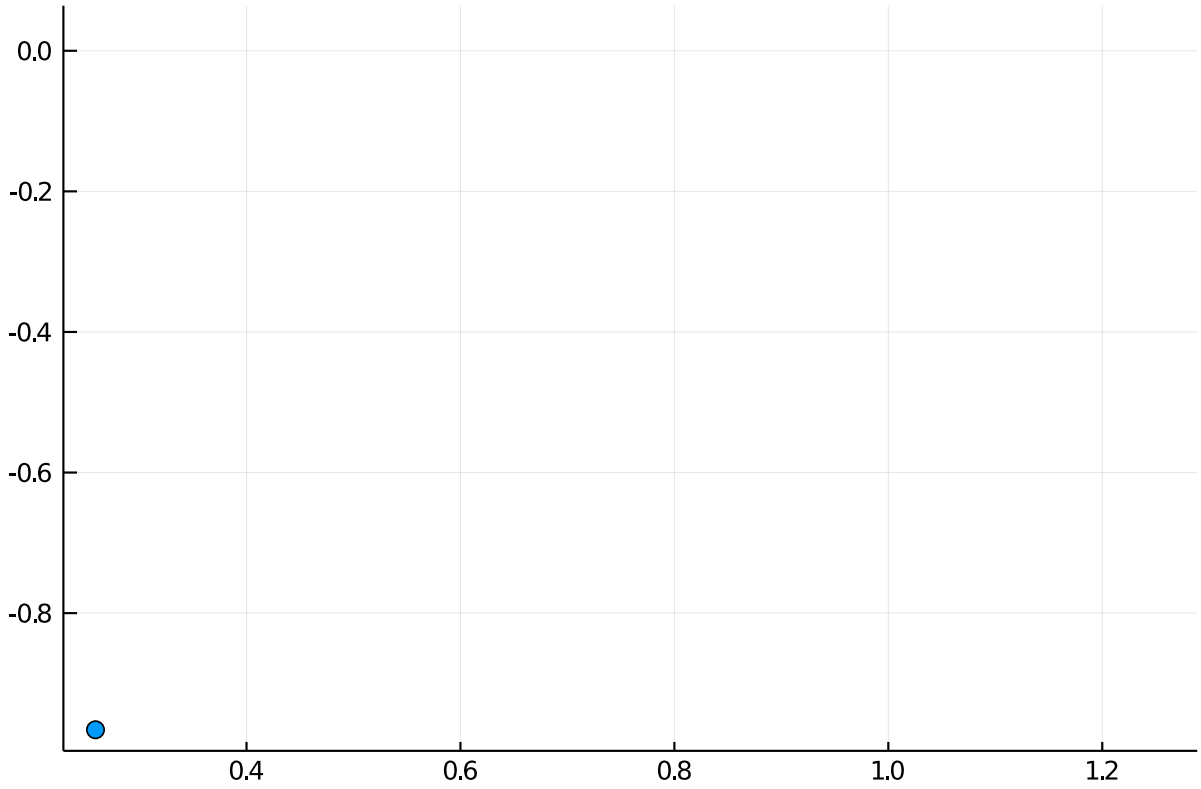
where $\hat{v}$ is the unit vector in the direction of $\vec{v}$.

Example A pendulum, a bob on a string, swings back and forth due to the force of gravity. When the bob is displaced from rest by an angle θ, then the tension force of the string on the bob is directed along the string and has magnitude given by the *projection* of the force due to gravity.

A force diagram is a useful visualization device of physics to illustrate the applied forces involved in a scenario. In this case the bob has two forces acting on it: a force due to tension in the string of unknown magnitude, but in the direction of the string; and a force due to gravity. The latter is in the downward direction and has magnitude mg, $g = 9.8 m/sec^2$ being the gravitational constant.

```
theta = pi/12
mass, gravity = 1/9.8, 9.8

l = [-sin(theta), cos(theta)]
p = -l
Fg = [0, -mass*gravity]
plot(legend=false)
arrow!(p, l)
arrow!(p, Fg)
scatter!(p[1:1], p[2:2], markersize=5)
```



The magnitude of the tension force is exactly that of the force of gravity projected onto $\vec{l}$, as the bob is not accelerating in that direction. The component of the gravity force in the perpendicular direction is the part of the gravitational force that causes acceleration in the pendulum. Here we find the projection onto $\vec{l}$ and visualize the two components of the gravitational force.

```
plot(legend=false, aspect_ratio=:equal)
arrow!(p, l)
arrow!(p, Fg)
scatter!(p[1:1], p[2:2], markersize=5)

proj = (Fg ⋅ l) / (l ⋅ l) * l   # force of gravity in direction of tension
porth = Fg - proj               # force of gravity perpendicular to tension

arrow!(p, proj)
arrow!(p, porth, linewidth=3)
```

Example Starting with three vectors, we can create three orthogonal vectors using projection and subtraction. The creation of `porth` above is the pattern we will exploit.

Let's begin with three vectors in R^3:

```
u = [1, 2, 3]
v = [1, 1, 2]
w = [1, 2, 4]
```

```
3-element Array{Int64,1}:
 1
 2
 4
```

We can find a vector from `v` orthogonal to `u` using:

```
unit_vec(u) = u / norm(u)
projection(u, v) = (u ⋅ unit_vec(v)) * unit_vec(v)

vorth = v - projection(v, u)
worth = w - projection(w, u) - projection(w, vorth)
```

```
3-element Array{Float64,1}:
 -0.33333333333333265
 -0.3333333333333336
  0.33333333333333354
```

We can verify the orthogonality through:

```
u ⋅ vorth, u ⋅ worth, vorth ⋅ worth
```

```
(-3.3306690738754696e-16, 8.881784197001252e-16, 3.677613769070831e-16)
```

This only works when the three vectors do not all lie in the same plane. In general, this is the beginnings of the Gram-Schmidt process for creating *orthogonal* vectors from a collection of vectors.

Algebraic properties The dot product is similar to multiplication, but different, as it is an operation defined between vectors of the same dimension. However, many algebraic properties carry over:

- commutative: $\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{u}$
- scalar multiplication: $(c\vec{u}) \cdot \vec{v} = c(\vec{u} \cdot \vec{v})$.
- distributive $\vec{u} \cdot (\vec{v} + \vec{w}) = \vec{u} \cdot \vec{v} + \vec{u} \cdot \vec{w}$

The last two can be combined: $\vec{u} \cdot (s\vec{v} + t\vec{w}) = s(\vec{u} \cdot \vec{v}) + t(\vec{u} \cdot \vec{w})$.

But associative does not make sense, as $(\vec{u} \cdot \vec{v}) \cdot \vec{w}$ does not make sense as two dot products: the result of the first is not a vector, but a scalar.

1.6 Matrices

Algebraically, the dot product of two vectors - pair off by components, multiply these, then add - is a common operation. Take for example, the general equation of a line, or a plane: