The line `res = h` isn't finished yet, but this is enough code to test. Once I finished and tested `error_func`, I would modify `duck` to use `fzero`.

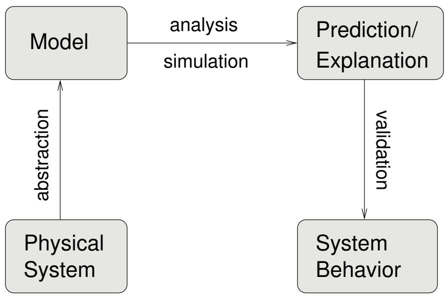
For this problem I might only need two functions, but if there were more, I could write and test them one at a time, and then combine them into a working program.

8.2 Physical modeling

Most of the examples so far have been about mathematics; Exercise 7.2, the "duck problem," is the first example we have seen of a physical system. If you didn't work on this exercise, you should at least go back and read it.

This book is supposed to be about **physical modeling**, so it might be a good idea to explain what that is. Physical modeling is a process for making predictions about physical systems and explaining their behavior. A **physical system** is something in the real world that we are interested in, like a duck.

The following figure shows the steps of this process:



A **model** is a simplified description of a physical system. The process of building a model is called **abstraction**. In this context, "abstract" is the opposite of "realistic;" an abstract model bears little direct resemblance to the physical system it models, in the same way that abstract art does not directly depict objects in the real world. A realistic model is one that includes more details and corresponds more directly to the real world.

Abstraction involves making justified decisions about which factors to include in the model and which factors can be simplified or ignored. For example, in the duck problem, we took into account the density of the duck and the buoyancy of water, but we ignored the buoyancy of the duck due to displacement of air and the dynamic effect of paddling feet. We also simplified the geometry of the duck by assuming that the underwater parts of a duck are similar to a segment of a sphere. And we used coarse estimates of the size and weight of the duck.

Some of these decisions are justifiable. The density of the duck is much higher than the density of air, so the effect of buoyancy in air is probably small. Other decisions, like the spherical geometry, are harder to justify, but very helpful. The actual geometry of a duck is complicated; the sphere model makes it possible to generate an approximate answer without making detailed measurements of real ducks.

A more realistic model is not necessarily better. Models are useful because they can be analyzed mathematically and simulated computationally. Models that are too realistic might be difficult to simulate and impossible to analyze.

A model is successful if it is good enough for its purpose. If we only need a rough idea of the fraction of a duck that lies below the surface, the sphere model is good enough. If we need a more precise answer (for some reason) we might need a more realistic model.

Checking whether a model is good enough is called **validation**. The strongest form of validation is to make a measurement of an actual physical system and compare it to the prediction of a model.

If that is infeasible, there are weaker forms of validation. One is to compare multiple models of the same system. If they are inconsistent, that is an indication that (at least) one of them is wrong, and the size of the discrepancy is a hint about the reliability of their predictions.

We have only seen one physical model so far, so parts of this discussion may not be clear yet. We will come back to these topics later, but first we should learn more about vectors.

8.3 Vectors as input variables

Since many of the built-in functions take vectors as arguments, it should come as no surprise that you can write functions that take vectors. Here's a simple (silly) example:

```
function res = display_vector(X)
    X
end
```

There's nothing special about this function at all. The only difference from the scalar functions we've seen is that I used a capital letter to remind me that `X` is a vector.

This is another example of a function that doesn't actually have a return value; it just displays the value of the input variable:

```
>> display_vector(1:3)

X = 1     2     3
```

Here's a more interesting example that encapsulates the code from Section 5.12 that adds up the elements of a vector: