Motivation: Parameter-Fitting

When we develop models, it is generally the case that the values of the parameters are unknown. We also generally have some idea of what the simulation output should look like. So, we develop a function that simulates the model with a given set of parameters and quantifies how close the simulation is to the desired output. The output is small if there is very little error and large if there is a lot of error. We call this function a cost function and our goal is to find the set of parameters that minimizes the cost function.

Numerical Optimization

Mathematically, this means our goal is to solve this problem

$$\min_{\mathbf{p}} c(\mathbf{p})$$

where $\mathbf{p}$ is the vector of $N_p$ parameters and $c$ is a function that returns a scalar.

As phrased above, the are no explicit constraints on the values of $\mathbf{p}$, but we will often impose the constraints that all parameters must be positive and that they must be within some “reasonable” bounds in terms of the biological system they are modeling.

We will begin by looking at local optimization methods. These methods assume that you know what area of parameter space to search and that it is perfectly fine to arrive in a local minimum. They also assume the cost function is smooth and that it has a value at every point in the local parameter space.

Numerical Optimization in 1 Dimension

Let us first consider the case where $p$ is a scalar.
**Example Model.** We will begin with the van der Pol oscillator, which has only one parameter, \( \mu \):

\[
\frac{dy_1}{dt} = y_2, \\
\frac{dy_2}{dt} = \mu (1 - y_1^2) y_2 - y_1.
\]

When \( \mu = 50 \), the period of oscillation is long (nearly 80). When \( \mu = 1 \), the period is short (approximately 6) and the system is not stiff.

**Example Cost Function.** Suppose we want to use the van der Pol oscillator to model the circadian clock. Our goal is to find the value of \( \mu \) that brings the period closest to 24. One approach would be to simulate the model, compute its period, then compute the cost as the absolute value of the difference between that period and 24. However, that would make the cost function undesirable, because the neighborhood around the minimum would not be smooth enough (it would not be differentiable). We may want to use a minimization scheme that depends on differentiable cost functions, so let’s avoid the absolute value. Instead, I am going to use the square of the relative difference between the computed period and 24. This allows me to know ahead of time some nice properties of the cost function – as long as the period is continuously dependent on \( \mu \), we will have a very pretty convex function.

```matlab
function errval = vdpCircadianError(mu)

% Initial condition and time bounds
 t0 = 0; 
 y0 = [2 0]; 
 tend = 480;

% Get the "true solution"
 options = odeset('RelTol',1e-3,'AbsTol',1e-6);
 [tm,ym] = ode15s(@vdp,t0:0.1:tend,y0,options,mu);
 mid = round(length(tm)/2);

 per = getPeriod(tm(mid:end),ym(mid:end,1));
 errval = ((per-24)/24)^2;
```
**Hill Descending Method.** We know that the cost function is probably smooth, but we don’t have any expression for the cost function itself (it is not an explicit function of the value of $\mu$). So we need a method that allows us to evaluate the cost function at many different values of $\mu$ and if we are intelligent about how we choose values of $\mu$, then we will find a value that approximately minimizes the function. The basic strategy of a hill-descending method is to begin with one value of $\mu$, then evaluate the cost function for it. Our goal is to find a value of $\mu$ nearby that allows us to take a step down the cost function hill. So we evaluate the cost function at values a little smaller than and a little larger than $\mu$. If one of those values leads to a lower cost, then we move our search to points near it. We repeat this process as long as we can make significant improvements to the cost.

**Input.**

- Handle to cost function. The cost function must take a scale parameter as input and return a scalar cost as output. Our goal is to minimize the cost.
- An initial guess as to a good value for the parameter
- The maximum number of iterations the algorithm should take
- Stopping criterion near zero. Once the difference between the cost at iteration $k$ and iteration $k - 1$ is less than the stopping criterion, the algorithm stops.

**Output.**

- best parameter value
- parameter value at every iteration
- cost at every iteration

**Algorithm.**

- Allocate space for return values and initialize any other necessary variables.
- Put the initial parameter value and its cost into the first entry of the return values.
- for $k = 2$ to MAX_ITER
  - Evaluate cost at a step taken in the positive direction
  - If it is better, then update the current parameter value and cost.
  - If there is cost improvement, but it is minimal, then we must have hit the bottom of the well, so break out of the loop.
  - Evaluate cost at a step taken in the negative direction
  - If it is better, then update the current parameter value and cost.
  - If there is cost improvement, but it is minimal, then we must have hit the bottom of the well, so break out of the loop.
  - If neither step improved the cost, then break out of the loop.
- Return the results.