An Introduction to Empirical Modeling

Douglas Robert Hundley
Mathematics Department
Whitman College

September 8, 2014

Contents

Chapter 1

Basic Models, Discrete Systems

1.1 Introduction

Mathematical modeling is the process by which we try to express physical processes mathematically. In so doing, we need to keep some things in mind:

- What are our assumptions?

  That is, what assumptions are absolutely necessary for us to emulate the desired behavior? For example, we might be interested in the fluctuation of a population. In that situation, we may or may not want to changes based on seasonality.

- The model should be **simple** enough so that we can understand it. If a model is so complicated as to defy understanding, then the model is not useful.

- The model should be **complex** enough so that we capture the desired behavior. The model should not be so simple that it does not explain anything- again, such a model is not useful. This does not mean that we need a lot of equations, however. One of the lessons of *chaos theory*[1] is the following:

  Very simple models can create very complex behavior

- To put the previous two items into context, when we build a model, we probably have some questions in mind that we'd like to answer. You can evaluate your model by seeing if it gives you the answer- For example,

[1] For a general introduction to Chaos Theory, consider reading "Chaos", by James Gleick. For a mathematical introduction, see "An Introduction to Chaotic Dynamics", by Robert Devaney

  - Should a stock be bought or sold?
  - Is the earth becoming warmer?
  - Does creating a law have a positive or negative social effect?
  - What is the most valuable property in monopoly?

  In this way, a model provides *added value*, and it is by this property that we might evaluate the goodness of a model.

- Once a model has been built, it needs to be checked against reality- Modeling is not a thought experiment! Of course, you would then go back to your assumptions, and revise, create new experiments, and check again.

You might notice that we have used very subjective terms in our definition of "modeling"- and these are an intrinsic part of the process. Some of the most beautiful and insightful models are those that are elegant in their simplicity. Most everyone knows the following model, which relates energy to mass and the speed of light:

$$E = mc^2$$

While it is simple, the model is also far-reaching in its implications (we will not go into those here). Other models of observed behavior from physics are so fundamental, we even call them physical "laws"- such as Newton's Laws of Motion.

In building mathematical models, you are allowed and encouraged to be creative. Explore and question your assumptions, explore your options for expressing those options mathematically, and most importantly, use your mathematical background.

1.2 What Kinds of Models Are There?

There are many ways of classifying mathematical models, which will make sense once you begin to build your own models. In general, we might consider the following classes of models:

1.2.1 Deterministic vs. Stochastic

A **stochastic** model is one that uses *random variation*. To describe this random variation properly, we will typically need ideas from statistics. An example: Model the outcomes of a roll of dice. Stochastic models are characterized by the introduction of statistics and probability. We won't be doing a lot of this in our course.

On the other hand, in a **deterministic** model, there is no randomness. As an example, we might model the temperature of a cup of coffee as it varies over time (a cooling model). Classically, the model would only involve the temperature of the coffee and the temperature of the environment.

There may not be a clean division of categories here; it is common for some models to incorporate both deterministic and stochastic parts. For example, a model for a microphone may include a model for the voice (deterministic), and a model for noise (stochastic).

It is interesting to consider the following: Does a deterministic model necessarily produce results that are completely predictable through all time? Interestingly, the answer is: Maybe yes, Maybe no. Yes, in the theoretical sense- we might be able to show that there exists a single unique solution to our problem. No, in the practical sense that we might not actually be able to compute that solution. However, this does not mean that all is lost- we might have excellent approximations over a short time span (think of the weather models on the news).

1.2.2 Discrete vs. Continuous Time

In modeling an occurrence that depends on time, it might happen at discrete steps in time (like interest on my credit card, or population), or in continuous time (like the temperature at my desk).

Modeling in Discrete Time

Discrete time models usually index time as a subscript- For example, a_n or x_n will be the value of a or x at time step n.

Discrete time models can be defined recursively, like the following:

$$a_{n+1} = a_n + a_{n-1}$$

In order to "solve" the system to produce a sequence, we would need to initialize the problem. For example, if $a_0 = 1$ and $a_1 = 1$, then we can find all the other elements of the sequence:

$$\{1, 1, 2, 3, 5, 8, 13, \cdots\}$$

You might recognize this as the famous Fibonacci sequence.

General discrete models

There are a couple of fundamental assumptions being made in these discrete models: (1) Time is indexed by the integers, and (2) The value of the process at some time $n + 1$ is a function of at most a finite number of previous states.

Mathematically, this means, given a_n and the L previous values of the state a, then the next state at time $n + 1$ is given by:

$$a_{n+1} = f(a_n, a_{n-1}, \ldots, a_{n-L})$$

Or, rather than modeling the states directly, we might model how the state *changes in time*:

$$a_{n+1} - a_n = \Delta a_n = f(a_{n-1}, \ldots, a_{n-L})$$

In either event, we will be left with determining the form for the function f and the length of the past, L. This form is called a **difference equation**.

We will work with both types of discrete models shortly. Before we do, let us contrast these models with continuous time models.

Modeling in Continuous Time

We may model using **Ordinary Differential Equations**. In these models, we are assuming that time passes continually, and that the rate of change of the quantity of interest depends only on the quantity and current time. We capture these assumptions by the following general model, where $y(t)$ is the quantity of interest.

$$\frac{dy}{dt} = f(t, y)$$

Note that this says simply that the rate of change of the quantity y depends on the time t and the current value of y.

Let us consider an example we've seen in Calculus: Suppose that we assume that acceleration of a falling body is due only to the force of gravity (we'll measure it as feet/sec^2). Then we may write:

$$y'' = -16$$

We can solve this for $y(t)$ by antidifferentiation:

$$y' = -16t + C_1, \quad y(t) = -8t^2 + C_1 t + C_2$$

where C_1, C_2 are unknowns that are problem-specific. These simples models are usually considered in a first course in ODEs.

To produce a more complex model, we might say that the rate of change depends not only on the quantity now, but also the value of the quantity in the past (for example, when regulating bodily functions the brain is reading values that are from the past). Such a model may take the following form, where $x(t)$ denotes the quantity of interest (such as the amount of oxygen in the blood):

$$\frac{dx}{dt} = f(x(t)) + g(x(t - \tau))$$

This is called a *delay differential equation*. One of the most famous of these models is the "Mackey-Glass" equation- You might look it up on the internet to see what the solution looks like!

If our phenomena requires more than time, we have to model via **Partial Differential Equations**. For example, it is common for a function $u(t, x)$ to depend both on time t and position x. Then a PDE may be:

$$\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2}$$

which might be interpreted to read: "The velocity of u at a particular time and position is proportional to its second spatial derivative. Modeling with PDEs is generally done in our Engineering Mathematics course.

1.2.3 Empirical vs. Analytical Modeling

"Empirical modeling" is modeling directly from data, rather than by some analytic process. In this case, our assumptions may take the form of a "model function" to which we will find unknown parameters. You've probably seen this before in articles where the researcher is fitting a line to data- in this example, we assume the model equation is given as $y = mx + b$, where m and b are the unknown parameters.

We'll finish this chapter with some analytical modeling using discrete dynamical systems (or, equivalently, discrete difference equations).

1.3 Discrete Dynamical Systems

The simplest type of dynamical system might be written in the following recurrence form (recurrence because we're writing the next value in terms of the current value):

$$x_{n+1} = ax_n$$

where we would call x_0 the *initial condition*. So, given x_0, we could compute the future values of the dynamical system:

$$x_1 = ax_0 \qquad x_2 = ax_1 = a^2x_0 \qquad x_3 = ax_2 = a^3x_0 \quad \cdots$$

The set of values $x_1, x_2, x_3, \ldots$ are called **the orbit** of x_0. We also notice that in this particular example, we were able to express x_n in terms of x_0. This is called the **closed form** of the solution to the difference equation given. That is,

$$x_n = a^n x_0$$

solves the system: $x_{n+1} = ax_n$. In fact, we can predict the long term behavior of this system:

$$|x_n| \to \begin{cases} 0 & \text{if } |a| < 1 \\ \infty & \text{if } |a| < 1 \\ x_0 & \text{if } a = 1 \end{cases}$$

And, if $a = -1$, the orbit oscillates between $\pm x_0$. In this case, we say that $\pm x_0$ are periodic with period 2.

Generally, if we have the L^{th} order difference equation:

$$x_{n+1} = f(x_n, x_{n-1}, \ldots, x_{n-(L-1)})$$

we would need to know L values of the past. For example, here's a second order difference equation:

$$x_{n+1} = x_n + x_{n-1}$$

So, if we have $x_0 = 1$ and $x_1 = 1$, then

$$x_2 = 2, \quad x_3 = 3, \quad x_4 = 5, \quad x_5 = 8, \quad x_6 = 13, \ldots$$

This is the Fibonacci sequence. In this case, the orbit grows without bound.

1.3.1 Periodicity

Before continuing, let's get some more vocab.

Given $x_{n+1} = f(x_n)$, a point w is a **fixed point** of f (or a fixed point for the orbit of x_0) if

$$w = f(w)$$

(Because dynamically, that point will never change). Continuing, a point w is a **periodic point of order** k if

$$f^k(w) = w$$

The least such k is the prime period. Here's an example- Find the fixed points and period 2 points for the following:

$$x_{n+1} = f(x_n) = x_n^2 - 1$$

SOLUTION: The fixed point is found by solving $x = f(x)$:

$$x^2 - 1 = x \qquad \Rightarrow \quad x^2 - x - 1 = 0 \quad \Rightarrow \quad x = \frac{1 \pm \sqrt{5}}{2}$$

The period two points are found by solving $x = F(F(x))$. Notice that we already have part of the solution (fixed points are also period 2 points).

$$F(F(x)) = (x^2 - 1)^2 - 1 = x^4 - 2x^2$$

so we solve:

$$x^4 - 2x^2 = x$$

which generally can be difficult to solve. However, we can factor out $x^2 - x - 1$ and x to factor completely:

$$x^4 - 2x^2 - x = 0 \quad \Rightarrow \quad x(x+1)(x^2 - x - 1) = 0$$

Therefore, $x = 0$ and $x = -1$ are the prime period 2 points.

Points may also be *eventually fixed*, like $x = \sqrt{2}$ for $F(x) = x^2 - 1$. If we compute the actual orbit, we get

$$\sqrt{2}, 1, 0, -1, 0, -1, \ldots$$

Solving First Order Equations

Consider making the first order sequence slightly more complicated:

$$x_{n+1} = ax_n + b$$

where a, b are constants (so $f(x) = ax + b$). Then, given an arbitrary x_0, we wonder if we can write the solution in closed form:

$$x_1 = ax_0 + b \qquad x_2 = ax_1 + b = a(ax_0 + b) + b = a^2x_0 + ab + b$$

For x_3, we have:

$$x_3 = a(\quad a^2x_0 + ab + b\quad) + b = a^3x_0 + a^2 + b + ab + b$$

and so on. Therefore, we have the closed form:

$$x_n = a^n x_0 + b(1 + a + a^2 + \cdots + a^{n-1})$$

Do we recall how to get the partial (or finite) geometric sum? Back then, we might have written it this way: Let S be the partial sum. That is,

$$\begin{array}{rl} S & = 1 + a + a^2 + \ldots + a^{n-1} \\ aS & = a + a^2 + \ldots + a^n \\ \hline (1-a)S & = 1 - a^n \end{array} \qquad S = \frac{1-a^n}{1-a} = \frac{a^n - 1}{a-1}$$

Given $x_{n+1} = ax_n + b$, the closed form solution is

$$x_n = a^n x_0 + b\,\frac{a^n - 1}{a-1}$$

and the fixed point is:

$$ax + b = x \quad \Rightarrow \quad (a-1)x = -b \quad \Rightarrow \quad x = \frac{b}{1-a}$$

Notice that we could re-write the closed form in terms of the fixed point:

$$x_n = a^n \left(x_0 - \frac{b}{1-a} \right) + \frac{b}{1-a}$$

EXAMPLE: Discrete Compound of Interest

Generally, if we begin with P_0 dollars accruing at an annual interest of r percent (as a number between 0 and 1), then

$$P_{n+1} = \left(1 + \frac{r}{12}\right) P_n$$

If you deposit an additional k dollars each month, you would add k to the previous amount, and we would have the form $F(x) = ax + b$ which we studied in the previous section.

CAUTION: Careful what you use for the interest rate. For example, with a 5% annual interest rate, the number r you see in the formula is $\frac{5}{100}$, so the overall quantity

$$1 + \frac{r}{12} = 1 + \frac{5}{1200}$$

Example

Suppose that Peter works for 4 years, and during this time he deposits $1000 each month on a savings account at an annual interest rate of 5% (with no initial deposit). During the next 4 years, he withdraws equal amounts p so that

at the end of 4 years, he has a zero balance again. Find p and the total interest earned.

SOLUTION: We'll treat the two time intervals separately, since the dynamics change after 4 years. For the first 4 years, we have $P_0 = 0$ and at the end of 4 years ($n = 48$), we have

$$P_{48} = b\frac{a^{48} - 1}{a - 1}$$

Substituting $n = 48$, $a = 1 + \frac{5}{1200} = \frac{241}{200}$, and $b = 1000$, we have:

$$P_{48} = \$53,014.89$$

For the next four years, the dynamical system has the form:

$$P_{n+1} = aP_n - k$$

where the new initial amount is $P_0 = 53014.89$, and k is the amount we're withdrawing each month. After 4 years, we have zero dollars exactly:

$$P_{48} = 53014.89a^{48} - k\frac{a^{48} - 1}{a - 1} = 0 \quad \Rightarrow \quad k = \frac{a - 1}{a^{48} - 1}(53014.89a^{48}) = k$$

That gives $k \approx \$1220.89$. For the total interest, Peter has withdrawn $48 \times 1220.89 = 58602.72$, and he has deposited $48 \times 1000 = 48000$. Therefore, putting it all together, Peter has made about \$10,602.72 in interest.

Exercises

1. You decide to purchase a home with a mortgage at 6% annual interest and with a term of 30 years (assume no down payment). If your house costs \$200,000, what will the monthly payment be? On the other hand, if you can only make \$1000 monthly payments, how much of a house can you afford?

Visualizing First Order Equations

See Ch 4 of the Devaney's text...

Given the form $x_{n+1} = F(x_n)$, and an initial point x_0, there is a nice way to visualize the orbit. Consider the graph of $y = F(x)$. Some observations:

- The points of intersection between $y = x$ and $y = F(x)$ are the fixed points of the recurrence.
- If we start with x_0 along the "$x-$axis, then go vertically to the graph, we will be at the point (x_0, x_1).
- To find x_2, first go horizontally from (x_0, x_1) to (x_1, x_1). Then treat x_1 as a domain value, and go vertically to the graph of F. The coordinate will now be (x_1, x_2).

- Continue this process to visualize the orbit of x_0.

From last time, we finished by considering

$$x_{n+1} = f(x_n)$$

In this particular instance, we can perform "graphical analysis" by looking at the graph of $y = f(x)$:

- Include the line $y = x$; the points of intersection are the fixed points.
- To find the orbit, given a number x_0:
  - Go vertically to $x_1 = f(x_0)$. Thus, you are located at the point (x_0, x_1). We want to use x_1 as the next domain point:
  - Go horizontally to the line $y = x$. Thus, you are now located at the point (x_1, x_1), so you can use x_1 as a domain value.
  - Go vertically to (x_1, x_2), where $x_2 = f(x_1)$.
  - Go horizontally to (x_2, x_2).
  - Go vertically to (x_2, x_3)
  - And so on...

IN CLASS EXAMPLES: $y = \sqrt{x}$ and $y = ax + b$.
We can define **attracting**, **repelling** fixed points.
Now, before going further, let's focus again on first and second order difference equations.

Definition: A difference equation is an equation typically of the form

$$x_{n+1} - x_n = f(x_{n-1}, \ldots, x_{n-L})$$

However, we will also see it as a discrete system:

$$x_{n+1} = f(x_n, \ldots, x_{n-L})$$

So we'll refer to either type when discussing difference equations.

Non-homogeneous Difference Equations

Consider now a slightly different form for difference equations:

$$x_{n+1} = ax_n + b_n$$

If b_n was actually constant, then we already derived the closed form of the solution. In this case, we'll focus on what happens if b_n is a function of n.

First, some vocab: If we only consider $x_{n+1} = ax_n$, then that is called the **homogeneous part of the equation**. The solution to that we've already

determined to be $x_n = Ca^n$ (where C depends on the initial condition), and we'll refer to this as the *homogeneous part of the solution.*

If we have a solution p_n to the full equation, where $b_n \neq 0$, we'll refer to that as the *particular part of the solution.*

We will show that, if p_n is any **particular solution**, then

$$x_n = ca^n + p_n$$

is a solution to the difference equation, and actually solves the DE with arbitrary starting conditions.

To show that x_n is indeed a solution, compute x_{n+1}, then compare with $ax_n + b_n$:

$$\begin{aligned} x_{n+1} &= ca^{n+1} + p_{n+1} \\ ax_n + b &= a(ca^n + p_n) + p_n = ca^{n+1} + ap_n + p_n \end{aligned}$$

This is a solution as long as $p_{n+1} = ap_n + p_n$, which it is. Finding p_n can be challenging, but there are some cases where we can "guess and check":

Example: Find the general solution to

$$x_{n+1} = 3x_n + 2n + 1$$

SOLUTION: We'll guess that b_n has the same general form as $2n + 1$, so we guess

$$b_n = A + Bn$$

Substituting this back into the difference equation, we have

$$A + B(n+1) = 3(A + Bn) + 2n + 1 \quad \Rightarrow \quad A + Bn + B = 3A + 3Bn + 2n + 1$$

$$0 = (2A - B + 1) + (2B + 2)n = 0$$

This equation is true for all $n = 1, 2, \ldots$, so therefore $2B+2 = 0$ and $2A-B+1 = 0$. That lets us solve, $B = -1$ and $A = -1$

$$p_n = -(n + 1)$$

The general solution:

$$x_n = c3^n - (n + 1)$$

where c is a constant that depends on the initial condition.

Sines and Cosines

We can do something similar for sines and cosines, although we need to use sum/difference formulas that you may not recall:

$$\begin{aligned} \sin(A + B) &= \sin(A)\cos(B) + \sin(B)\cos(A) \\ \cos(A + B) &= \cos(A)\cos(B) - \sin(A)\sin(B) \end{aligned}$$

NOTE: I'll provide these formulas for quizzes/exams.

Here's an example:

$$x_{n+1} = -x_n + \cos(2n)$$

The homogeneous part of the solution is $c(-1)^n$. For the particular part, we'll guess that

$$p_n = A\cos(2n) + B\sin(2n)$$

and we'll see if we can solve for A, B. Substituting, we have:

$$A\cos(2(n+1)) + B\sin(2(n+1)) = -A\cos(2n) - B\sin(2n) + \cos(2n)$$

Using the formulas,

$$A(\cos(2n)\cos(2) - \sin(2n)\sin(2)) + B(\sin(2n)\cos(2) + \sin(2)\cos(2n)) = -A\cos(2n) - B\sin(2n) + \cos(2n)$$

Collecting terms, we look at the coefficients of $\cos(2n)$ and $\sin(2n)$ separately:

$$\cos(2n)\left[A\cos(2) + B\sin(2) + A\right] = \cos(2n)$$

$$\sin(2n)\left[-A\sin(2) + B\cos(2) + B\right] = 0$$

These equations must be true for each integer n, therefore

$$\begin{array}{rl} A(1+\cos(2)) + B\sin(2) & = 1 \\ -A\sin(2) + B(1+\cos(2)) & = 0 \end{array} \quad \Rightarrow \quad A = \frac{1}{2}, \quad B = \frac{\sin(2)}{2(1+\cos(2))}$$

The overall solution is therefore:

$$x_n = C(-1)^n + \frac{1}{2}\cos(2n) + \frac{\sin(2)}{2(1+\cos(2))}\sin(2n)$$

EXERCISE: Find the general solution to

$$x_{n+1} = \frac{1}{2}x_n + \frac{n}{2^n}.$$

Do this by assuming that the the particular solution is of the form

$$p_n = \frac{n(An+B)}{2^n}$$

Closing Notes

We might notice that solving these difference equations is very similar to solving

$$A\mathbf{x} = \mathbf{b}$$

in linear algebra, or solving

$$ay'' + by' + cy = g(t)$$

in differential equations (the Method of Undetermined Coefficients). This is not a coincidence- They all rely on the underlying equation being from a *linear operator*.

For now, we will close the introduction in order to get an introduction to Matlab.

Chapter 2

A Case Study in Learning

2.1 A Case Study in Learning

In a broad sense, *learning is the process of building a "desirable" association between stimulus and response (domain and range), and is measured through resulting behavior on stimulus that has not been previously seen.*

In machine learning, problems are typically cast in one of two models: Either *supervised* or *unsupervised* learning.

In *supervised learning*, we are given examples of proper behavior, and we want the computer to emulate (and extrapolate from) that behavior.

In the other type of learning, *unsupervised learning*, no specific outputs are given per input, but rather an overall goal is given. Here are some examples to help with the definition:

- Suppose you have an old clunker of a car that doesn't have much of an engine. You're stuck in a valley, and so the only way out will be to go as fast as you can for a while, then let gravity take you back up the other side of the hill, then accelerate again, and so on. You hope that you can build up enough momentum to get out of the valley (that's the goal).

- Suppose you're driving a tractor-trailer, and you need to back the trailer into a loading dock (that's your goal).

- In a game of chess, the input would be the position of each of the chess pieces. The overall goal is to win the game.

In general, supervised learning is easier than unsupervised learning. One reason is that in unsupervised learning, a lot of time wasted in trial-and-error exploration of the possible input space. Contrast that with supervised learning, where the "correct" behavior is explicitly given.

2.1.1 Questions for Discussion:

1. Consider the concept of *superstition*: This is a belief that one must engage in certain behaviors in order to receive a certain reward, where in reality, the reward did not depend on those behaviors. Is it possible for a computer to engage in superstitious activity? Discuss in terms of the supervised versus unsupervised learning paradigms.

2. A signal light comes on and is followed by one of two other lights. The goal is to predict which of the lights comes on given that the signal light comes on. The experimenter is free to arrange the pattern of the two response lights in any way- for example, one might come on 75% of the time.

   Let E_1, E_2 denote the event that the first (second) light comes on, and let A_1, A_2 denote the prediction that the first (second) light comes on (respectively). Let π be the probability that E_1 occurs.

   (a) If the goal is to maximize your reward through accurate predictions, what should you do in this experiment? Just give a heuristic answer- you do not have to formally justify it.

   (b) How would you program a machine to maximize it's prediction accuracy? Can you state this in mathematical terms?

   (c) What do you think happens with actual subject (human) trials?

2.2 The $n-$Armed Bandit

The one armed bandit is slang for a slot machine, so the $n-$armed bandit can be thought of as a slot machine with n arms. Equivalently, you may think of a room with n slot machines.

The problem we're trying to solve is the classic Las Vegas quandry: How should we play the slot machines in order to maximize our returns?

Discussion Question: Is the $n-$armed bandit a case or supervised or unsupervised learning?

First, let us set up some notation: Let a be an integer between 1 and n that defines which machine we're playing. Then define the expected return:

$$Q(a) = \text{ The expected return for playing slot machine } a$$

You can also think of $Q(a)$ as the *mean* of the payoffs for slot machine a.

If we knew $Q(a)$ for each machine a, our strategy to maximize our returns would be very simple: "Play only machine a".

Of course, what makes the problem interesting is that we don't know what the any of the returns are, let alone which machine gives the maximum. That leaves us to estimate the returns, and because there will always be uncertainty

associated with these estimates, we will never know if the estimates are correct. We hope to construct estimates that get better over time (and experience).

Let's first set up some notation. Let

$$Q_t(a) = \text{ Our estimation of } Q(a) \text{ at time } t.$$

so we hope that our estimates get better in time:

$$\lim_{t\to\infty} Q_t(a) = Q(a) \tag{2.1}$$

Suppose we play slot machine a a total of n_a times, with payoffs $r_1, \ldots, r_{n_a}$ (note that these values could be negative!). Then we might estimate $Q(a)$ as the mean of these values:

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{n_a}}{n_a}$$

In statistical terms, we are using the sample mean to estimate the actual mean which is a reasonable thing to do as a starting point. We'll also initialize the estimates to be zero: $Q_0(a) = 0$.

We now come to the big question: What approach should we take to accomplish our goal (of maximizing our reward). The first one up is a good place to start.

2.2.1 The Greedy Algorithm

This strategy is straightforward: Always play the slot machine with the largest (estimated) payoff. If a_{t+1} is the machine we'll play at time $t+1$, then:

$$a_{t+1} = \arg\max \{Q_t(1), Q_t(2), \ldots, Q_t(n)\}$$

where "arg" refers to the argument of the maximum (which is an integer from 1 to n corresponding to the max) . If there is a tie, then choose one of them at random.

We'll need to translate this into a learning algorithm, so set's take a moment to see how we might implement the greedy algorithm in Matlab.

Translating to Matlab

The `find` and `max` commands will be used to find the argument of the maximum value. For example, if x is a (row) vector of numbers, then the following command:

```
idx=find(x==max(x))
```

will return all indices of the vector x that are equal to the max.

Here's an example. Suppose we have vector x as given. What does Matlab do?

```
x=[1 2 3 0 3];
idx=find(x==max(x));
```

The result will be a vector, `idx`, that contains the values 3 and 5 (that is, the third and fifth elements of x are where the maximum occurs).

Going back to the greedy algorithm, I think you'll see a problem- What if the estimations are wrong? Then its very possible that you'll get stuck on a suboptimal machine. This problem can be dealt with in the following way: Every once in a while, try out the other machines to see what you get. This is what we'll do in the next section.

2.2.2 The $\epsilon-$Greedy Algorithm

In this algorithm, rather than always choosing the machine with the greatest current estimate of the payout, we will choose, with probability ϵ, a machine at random.

With this strategy, as the number of trials gets larger and larger, $n_a \to \infty$ for *all* machines a, and so we will be guaranteed convergence to the proper estimates of $Q(a)$ for all a machines.

On the flip side, because we're always investigating other machines every once in a while, we'll never maximize our returns (we will always have suboptimal returns).

Implementing *epsilon*−greedy in Matlab

Using some "pseudo-code", here is what we want our algorithm to do:

`For each time we choose a machine:`

- `Select an action:`
  - `Sometimes choose a machine at random`
  - `Otherwise, select the action with greatest return. Check for ties, and if there is a tie, pick on of them at random.`
- `Get your payoff`
- `Update the estimates` Q

`Repeat.`

Our first programming problem will be to implement the statement "Sometimes choose a machine at random". If we define $\epsilon =$`E` to be the probability of this event, and `N` is the number of trials, then one way of selection is to set up a vector with N elements which we'll call `greedy`, that will "flag" the events- that is, on trial j, if `greedy(j)`$= 1$, choose a machine at random. Otherwise, choose using the greedy method. The following code will do just that (`N` is the number of trials)

```
greedy=zeros(1,N);
if E>0
   m=round(E*N);  %Total number of times we should choose at random
   greedy(1:m)=ones(1,m);
   m=randperm(N); %Randomly permute the vector indices
   greedy=greedy(m);
   clear m
end
```

And here's the full function. We assume that the actual rewards for each of the bandits is given in the vector `Aq`, and that when machine a is played, the sample reward will be chosen from a normal distribution with unit variance and mean `Aq(a)`.

```
function [As,Q,R]=banditE(N,Aq,E)
%FUNCTION [As,Q,R]=banditE(N,Aq,E)
%  Performs the N-armed bandit example using epsilon-greedy
%    strategy.
%  Inputs:
%        N=number of trials total
%        Aq=Actual rewards for each bandit (these are the mean rewards)
%        E=epsilon for epsilon-greedy algorithm
%  Outputs:
%        As=Action selected on trial j, j=1:N
%        Q are the reward estimates
%        R is N x 1, reward at step j, j=1:N

numbandits=length(Aq);       %Number of Bandits
ActNum=zeros(numbandits,1);  %Keep a running sum of the number of times
                             %  each action is selected.
ActVal=zeros(numbandits,1);  %Keep a running sum of the total reward
                             %  obtained for each action.
Q=zeros(1,numbandits);       %Current reward estimates
As=zeros(N,1);               %Storage for action
R=zeros(N,1);                %Storage for averaging reward

%**********************************************************************
%  Set up a flag so we know when to choose at random (using epsilon)
%**********************************************************************
greedy=zeros(1,N);
if E>0
   m=round(E*N);  %Total number of times we should choose at random
   greedy(1:m)=ones(1,m);
   m=randperm(N);
   greedy=greedy(m);
   clear m
end
if E>=1
    error('The epsilon should be between 0 and 1\n');
end
```

```
%*********************************************************************
%
%  Now we're ready for the main loop
%*********************************************************************
for j=1:N
    %STEP ONE:  SELECT AN ACTION  (cQ) , GET THE REWARD  (cR) !
        if greedy(j)>0
            cQ=ceil(rand*numbandits);
            cR=randn+Aq(cQ);
        else
            [val,idx]=find(Q==max(Q));
            m=ceil(rand*length(idx));  %Choose a max at random
            cQ=idx(m);
            cR=randn+Aq(cQ);
        end
        R(j)=cR;
    %UPDATE FOR NEXT GO AROUND!
        As(j)=cQ;
        ActNum(cQ)=ActNum(cQ)+1;
        ActVal(cQ)=ActVal(cQ)+cR;
        Q(cQ)=ActVal(cQ)/ActNum(cQ);
end
```

Next we'll create a test bed for the routine. We will call the program 2,000 times, and each call will consist of 1,000 plays. We will set the number of bandits to 10, and change the value of ϵ from 0 to 0.01 to 0.1, and see what the average reward per play is over the 1000 plays.

Here's a script file that we'll use to call the `banditE` routine:

```
Ravg=zeros(1000,1);
E=0.1;
for j=1:2000
    m=randn(10,1);
    [As,Q,R]=banditE(1000,m,E);
    Ravg=Ravg+R;
    if mod(j,10)==0
        fprintf('On iterate %d\n',j);
    end
end
Ravg=Ravg./2000;
plot(Ravg);
```

The output of the algorithms are shown in Figure 2.1.

The Softmax Action Selection

In the Softmax action selection algorithm, the idea is to construct a set of probabilities. This set will have the properties that:

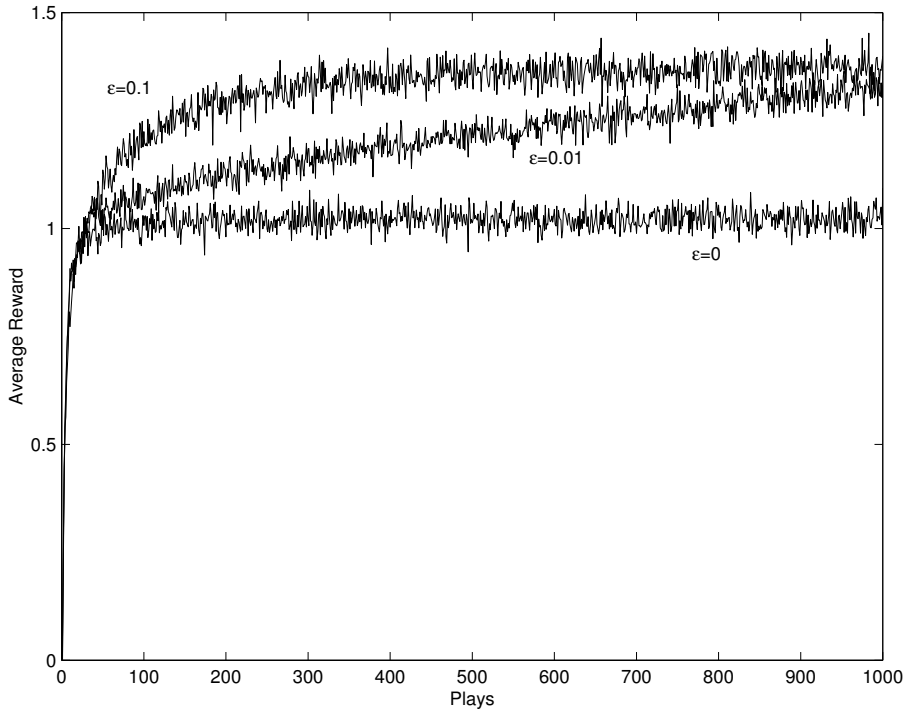

Figure 2.1: Results of the testbed on the 10-armed bandit. Shown are the rewards given per play, averaged over 2000 trials.

- The machine (or arm) giving the highest estimated payoff will have the highest probability.
- We will choose a machine using the probabilities. For example, if the probabilities are $0.5, 0.3, 0.2$ for machines $1, 2, 3$ respectively, then machine 1 would be chosen 50% of the time, machine 2 would be chosen 30% of the time, and the last machine 20% of the time.

Therefore, this algorithm will maintain an exploration of all machines so that we will not get locked onto a suboptimal machine.

Now if we have n machines with estimated payoffs recorded as:

$$Q = [Q_t(1), Q_t(2), \ldots, Q_t(n)]$$

we want to construct n probabilities,

$$P = [P_t(1), P_t(2), \ldots, P_t(n)]$$

The requirements for this transformation are:

1. $P_t(k) \geq 0$ for $k = 1, 2, \ldots$ (because all probabilities are positive). Another way to say this is to say that the range of the transformation is nonnegative.

2. If $Q_t(a) < Q_t(b)$, then $P_t(a) < P_t(b)$. That is, the transformation must be strictly increasing for all domain values.

3. Finally, the sum of the probabilities must be 1.

A function that satisfies requirements 1 and 2 is the exponential function. It's range is nonnegative. It maps large negative values (large negative payoffs) to near zero probability, and it is strictly increasing. Up to this point, the transformation is:

$$\hat{P}_t(k) = \mathrm{e}^{Q_t(k)}$$

We need the probabilities to sum to 1, so we normalize the $\hat{P}_t(k)$:

$$P_t(k) = \frac{\hat{P}_t(k)}{\hat{P}_t(1) + \hat{P}_t(2) + \ldots + \hat{P}_t(n)} = \frac{\exp(Q_t(k))}{\sum_{j=1}^{n} \exp(Q_t(j))}$$

This is a popular technique worth remembering- We have what is called a Gibbs (or Boltzmann) distribution. We could stop at this point, but it is convenient to introduce a control parameter τ (sometimes this is referred to as the temperature of the distribution). Our final version of the transformation is given as:

$$P_t(k) = \frac{\exp\left(\frac{Q_t(k)}{\tau}\right)}{\sum_{j=1}^{n} \exp\left(\frac{Q_t(j)}{\tau}\right)}$$

EXERCISE: Suppose we have two probabilities, $P(1)$ and $P(2)$ (we left off the time index since it won't matter in this problem). Furthermore, suppose $P(1) > P(2)$. Compute the limits of $P(1)$ and $P(2)$ as τ goes to zero. Compute the limits as τ goes to infinity (Hint on this part: Use the definition, and divide numerator and denominator by $\exp(Q(1)/\tau)$ before taking the limit).

What we find from the previous exercise is that the effect of large τ (hot temperatures) makes all the probabilities about the same (so we would choose a machine almost at random). The effect of small *tau* (cold temperatures) makes the probability of choosing the best machine almost 1 (like the greedy algorithm).

In Matlab, these probabilities are easy to program. Let `Q` be a vector holding the current estimates of the returns, as before, and let `t`= τ, the temperature. Then we construct a vector of probabilities using the softmax algorithm:

```
P=exp(Q./t);
P=P./sum(P);
```

Programming Comments

1. How to select action a with probability $p(a)$?

   We could do what we did before, and create a vector of choices with those probabilities fixed, but our probabilities change. We can also use

the uniform distribution, so that if `x=rand`, and $x \leq p(1)$, use action 1. If $p(1) < x \leq p(1) + p(2)$, choose action 2. If $p(1) + p(2) < x \leq p(1) + p(2) + p(3)$, choose action 3, and so on. There is an easy way to do this, but it is not optimal (in terms of speed). We introduce two new Matlab functions, `cumsum` and `histc`.

The function `cumsum`, which means *cumulative sum*, takes a vector x as input, and outputs a vector y so that `y=cumsum(x)` creates:

$$y_k = \sum_{n=1}^{k} x_n = x_1 + x_2 + \ldots + x_k$$

For example, if $x = [1, 2, 3, 4, 5]$, then `cumsum(x)` would output $[1, 3, 6, 10, 15]$

The function `histc` (for *histogram count*) has the form: `n=histc(x,y)`, where the vector y is monotonically increasing. The elements of y form "bins" so that $n(k)$ counts the number of values in `x` that fall between the elements `y(k)` (inclusive) and `y(k+1)` (exclusive) in the vector `y`. Try a particular example, like:

```
Bins=[0,1,2];
x=[-2, 0.25, 0.75, 1, 1.3, 2];
N=histc(x, Bins);
```

`Bins` sets up the desired intervals as $[0, 1)$ and $[1, 2)$ and the last value is set up as its own interval, 2. Since -2 is outside of all the intervals, it is not counted. The next two elements of x are inside the first interval, and the next two elements are inside the second interval. Thus, the output of this code fragment is $N = [2, 2, 1]$.

Now in our particular case, we set up the bin edges (intervals) so that they are the cumulative sums. We'll then choose a number between 0 and 1 using the (uniformly) random number $x =$ `rand`, and determine what interval it is in. This will be our action choice:

```
P=[0.3, 0.1, 0.2, 0.4];
BinEdges=[0, cumsum(P)];
x=rand;
Counts=histc(x,BinEdges);
ActionChoice=find(Counts==1);
```

2. We have to change our parameter τ from some initial value τ_{init} (big, so that machines are chosen almost at random) to some small final value, τ_{fin}. There are an infinite number of ways of doing this. For example, a linear change from a value a to a value b in N steps would be the equation of the line going from the point $(1, a)$ to the point (N, b).

   Exercise: Give a formula for the parameter update, τ in terms of the initial value, τ_{init} and the final value, τ_{fin} if we use a linear decrease as t ranges from 1 to N.

A more popular technique is to use the following formula, which we'll use to update many parameters. Let the initial value of the parameter be given as a, and the final value be given as b. Then the parameter p is computed as:

$$p = a \cdot \left(\frac{b}{a}\right)^{t/N} \tag{2.2}$$

Note that when $t = 0$, $p = a$ and when $t = N$, $p = b$[1]

"Win-Stay, Lose-Shift" Strategy

The "Win-Stay, Lose-Shift" strategy discussed in terms of Harlow's monkeys and perhaps the probability matching experiments of Estes might be re-formulated here for the $n-$armed bandit experiment.

In this experiment, we interpret the strategy as: If I'm winning, make the probability of choosing that action stronger. If I'm losing, make the probability of choosing that action weaker. This brings us to the class of *pursuit* methods.

Define a^* to be the winning machine at the moment, i.e.,

$$a^* = \max_a Q_t(a)$$

The idea now is straightforward- Slightly increase the probability of choosing this winning machine, and correspondingly decrease the probability of choosing the others.

Define the probability of choosing machine a as $P(a)$ (or, if you want to explicitly include the time index, $P_t(a)$. Then given the winning machine index as a^*, we update the current probabilities by using a parameter $\beta \in [0, 1]$:

$$P_{t+1}(a^*) = P_t(a^*) + \beta\,[1 - P_t(a^*)]$$

and the rest of the probabilities decrease towards zero:

$$P_{t+1}(a) = P_t(a) + \beta\,[0 - P_t(a)]$$

Exercises with the Pursuit Strategy

1. Suppose we have three probabilities, P_1, P_2, P_3, and P_1 is the unique maximum. Show that, for any $\beta > 0$, the updated values still sum to 1.

2. Using the same values as before, show that, for any $\beta > 0$, the updated values will stay between 0 and 1- that is, If $0 \leq P_i \leq 1$ for all i before the update, then after the update, $0 \leq P_i \leq 1$.

3. Here is one way to deal with a tie (show that the updated values still sum to 1): If there are k machines with a maximum, update each via:

$$P_{t+1} = (1 - \beta) * P_t + \beta/k$$

[1] In the C/C++ programming language, indices always start with zero, and this is leftover in this update rule. This is not a big issue, and the reader can make the appropriate change to starting with $t = 1$ if desired.

4. Suppose that for some fixed j, P_j is always a loser (never a max). Show that the update rule guarantees that $P_j \to 0$ as $t \to \infty$. HINT: Show that $P_j(t) = (1-\beta)^t P_j(0)$

5. Suppose that for some fixed j, P_j is always a winner (with no ties). Show that the update rule guarantees that $P_j \to 1$ as $t \to \infty$.

Matlab Functions `softmax` and `winstay`

Here are functions that will yield the softmax and win-stay, lose-shift strategies. Below each is a driver. Read through them carefully so that you understand what each does. We'll then ask you to put these into Matlab and comment on what you see.

```
function a=softmax(EstQ,tau)
% FUNCTION a=softmax(EstQ, tau)
%   Input:  Estimated payoff values in EstQ (size 1 x N,
%           where N is the number of machines
%           tau - "temperature":  High values- the probs are all
%           close to equal; Low values, becomes "greedy"
%  Output:  The machine that we should play (a number between 1 and N)

if tau==0
    fprintf('Error in the SoftMax program-\n');
    fprintf('Tau must be greater than zero\n');
    a=0;
    return
end

Temp=exp(EstQ./tau);
S1=sum(Temp);
Probs=Temp./S1;  %These are the probabilities we'll use

%Select a machine using the probabilities we just computed.
x=rand;
TotalBins=histc(x,[0,cumsum(Probs)']);
a=find(TotalBins==1);
```

Here is a driver for the softmax algorithm. Note the implementation details (e.g., how the "actual" payoffs are calculated, and what the initial and final parameter values are):

```
%Script file to run the N-armed bandit using the softmax strategy

%Initializations are Here:
NumMachines=10;
ActQ=randn(NumMachines,1);  %10 machines
```

```
NumPlay=1000;               %Play 100 times
Initialtau=10;              %Initial tau ("High in beginning")
Endingtau=0.5;
tau=10;
NumPlayed=zeros(NumMachines,1);  %Keep a running sum of the number
                                 %  of times each action is selected
ValPlayed=zeros(NumMachines,1);  %Keep a running sum of the total
                                 %  reward for each action
EstQ=zeros(NumMachines,1);
PayoffHistory=zeros(NumPlay,1);  %Keep a record of our payoffs

for i=1:NumPlay

    %Pick a machine to play:
    a=softmax(EstQ,tau);

    %Play the machine and update EstQ, tau
    Payoff=randn+ActQ(a);
    NumPlayed(a)=NumPlayed(a)+1;
    ValPlayed(a)=ValPlayed(a)+Payoff;
    EstQ(a)=ValPlayed(a)/NumPlayed(a);
    PayoffHistory(i)=Payoff;
    tau=Initialtau*(Endingtau/Initialtau)^(i/NumPlay);
end
[v,winningmachine]=max(ActQ);
winningmachine
NumPlayed
plot(1:10,ActQ,'k',1:10,EstQ,'r')
```

Here is the function implementing the pursuit strategy (or "Win-Stay, Lose-Shift").

```
function [a, P]=winstay(EstQ,P,beta)
% function [a,P]=winstay(EstQ,P,beta)
%  Input:  EstQ, Estimated values of the payoffs
%          P = Probabilities of playing each machine
%          beta= parameter to adjust the probabilities, between 0 and 1
%  Output:  a = Which machine to play
%           P = Probabilities for each machine

[vals,idx]=max(EstQ);
winner=idx(1);  %Index of our "winning" machine

%Update the probabilities.  We need to do P(winner) separately.
NumMachines=length(P);
P(winner)=P(winner)+beta*(1-P(winner));
```

```
Temp=1:NumMachines;
Temp(winner)=[];  %Temp now holds the indices of all "losers"
P(Temp)=(1-beta)*P(Temp);

%Probabilities are all updated- Choose machine a w/prob P(a)
x=rand;
TotalBins=histc(x,[0,cumsum(P)']);
a=find(TotalBins==1);
```

And its corresponding driver is below. Again, be sure to read and understand what each line of the code does:

```
%Script file to run the N-armed bandit using pursuit strategy

%Initializations
NumMachines=10;
ActQ=randn(NumMachines,1);
NumPlay=2000;
Initialbeta=0.01;
Endingbeta=0.001;
beta=Initialbeta;
NumPlayed=zeros(NumMachines,1);
ValPlayed=zeros(NumMachines,1);
EstQ=zeros(NumMachines,1);
Probs=(1/NumMachines)*ones(10,1);

for i=1:NumPlay

    %Pick a machine to play:
    [a,Probs]=winstay(EstQ,Probs,beta);

    %Play the machine and update EstQ, tau
    Payoff=randn+ActQ(a);
    NumPlayed(a)=NumPlayed(a)+1;
    ValPlayed(a)=ValPlayed(a)+Payoff;
    EstQ(a)=ValPlayed(a)/NumPlayed(a);
    beta=Initialbeta*(Endingbeta/Initialbeta)^(i/NumPlay);
end
[v,winningmachine]=max(ActQ);
winningmachine
NumPlayed
plot(1:10,ActQ,'k',1:10,EstQ,'r')
```

Homework: Implement these 4 pieces of code into Matlab, and comment on the performance of each. You might try changing the initial and final values

of the parameters to see if the algorithms are *stable* to these changes. As you form your comments, recall our two competing goals for these algorithms:

- Estimate the values of the actual payoffs (more accurately, the mean payout for each machine).
- Maximize our rewards!

2.2.3 A Summary of Reinforcement Learning

We looked in depth at a basic problem of unsupervised learning- That of trying to find the best winning slot machine in a bank of many. This problem was unsupervised because, although we got rewards or punishments by winning or losing money, we did not know at the beginning of the problem what those payoffs would be. That is, there was no expert available to tell us if we were doing something correctly or not, *we had to infer correct behavior from directly playing the machines.*

We also saw that to solve this problem, we had to do a lot of *trial and error* learning- that's typical in unsupervised learning. Because an expert is not there to tell us the operating parameters, we have to spend time exploring the possibilities.

We learned some techniques for translating learning theory into mathematics, and in the process, we learned some commands in Matlab. We don't expect you to be an expert programmer - this should be a fairly gentle introduction to programming. At this stage, you should be able to read some Matlab code and interpret the output of an algorithm. Later on, we'll give you more opportunities to produce your own pieces of code.

In summary, we looked at the greedy algorithm, the $\epsilon-$greedy algorithm, the softmax strategy, and the pursuit strategy. You might consider how closely (if at all) these algorithms would reproduce human or animal behavior if given the same task.

There are many more topics in Reinforcement Learning to consider, we presented only a short introduction to the topic.