Advanced Modelling in Biology Primer

Imperial College London

Spring 2009

Introduction and Topics Covered

Introduction

This primer is meant to be a guide as well as a starting point for the student taking Advanced Biological Modelling and is not meant to supplant lectures. The reason for developing such a primer was intended to make the course more exciting and relevant to todays algorithms and methods giving both a foundation to what is being done in different fields and allowing for further reading into topics that you might be more interested in.

Given that this is only a one semester course taught two hours every week, it is impossible to expect to cover every topic and application, yet resources are available both here and on the internet if one wanted to go deeper into any subject.

The topics covered in this primer and corresponding lectures are listed below, but are subject to revision. A website is also dedicated to this course which has most of the information covered in this primer as well as many other links to various sources on the internet if any subject is still unclear to you or if you desire further reading.

The website can be found at:

`http://openwetware.org/wiki/User:Johnsy/Advanced_Modelling_in_Biology`

Optimization

- Introduction to optimization: definitions and concepts, standard formulation. Convexity. Combinatorial explosion and computationally hard problems.
- Least squares solution: pseudo-inverse; multivariable case. Applications: data fitting.
- Constrained optimization:
  - Linear equality constraints: Lagrange multipliers
  - Linear inequality constraints: Linear programming. Simplex algorithm. Applications.

- Gradient methods: steepest descent; dissipative gradient dynamics; improved gradient methods.
- Heuristic methods:
  - Simulated annealing: Continuous version; relation to stochastic differential equations.
  - Neural networks: General architectures; nonlinear units; back-propagation; applications and relation to least squares.
- Combinatorial optimization: hard problems, enumeration, combinatorial explosion. Examples and formulation.
- Heuristic algorithms: simulated annealing (discrete version); evolutionary (genetic) algorithms. Applications.

Discrete Systems

- Linear difference equations: general solution; auto-regressive models; relation to z-transform and Fourier analysis.
- Nonlinear maps: fixed points; stability; bifurcations. Poincar section. Cobweb analysis. Examples: logistic map in population dynamics (period-doubling bifurcation and chaos); genetic populations.
- Control and optimization in maps. Applications: management of fisheries.

Advanced Topics (Networks & Chaos)

- Networks in biology: graph theoretical concepts and properties; random graphs; deterministic, constructive graphs; small-worlds; scale-free graphs. Applications in biology, economics, sociology, engineering.
- Nonlinear control in biology: recurrence plots and embeddings; projection onto the stable manifolds; stabilization of unstable periodic orbits and anti-control. Applications to physiological monitoring.

Contents

Chapter 1

Optimization

1.1 What do we mean when we say optimization?

The general formulation for any optimization problem is this: We want to minimize a function $f(x)$, known as our cost or objective function, subject to a series of constraints (e.g. $g_i(x) = 0$ and $h_i(x) < 0$).

$f(x)$ is a function in the general sense, meaning that it can be either continuous or discrete. A continuous function $f(x)$ is one which is defined for all values of x. A discrete function is one which is only defined for a discrete input set $x_i = [x_1, x_2, \dots]$.

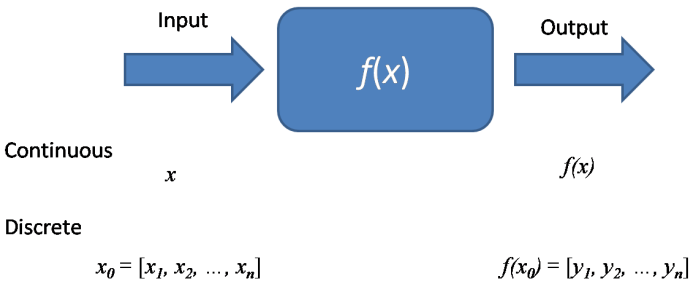

Figure 1.1: Continuous vs. Discrete Functions

It is easy to think of $f(x)$ as a black box having an output for any given input.

1.2 Standard Optimization vs. Combinatorial Optimization

Standard optimization, simply put, is optimization which deals with functions where the output is continuous (both variables and functions are continuous). On the other hand, combinatorial optimization deals with functions where the output is a discrete set. Remember that the fact that it is discrete does not

mean that it is not infinite! It is also important to note that calculus does not apply to combinatorial problems, making them more difficult to analyze.

1.3 The Traveling Salesman Problem

The traveling salesman problem is probably the most famous example of a combinatorial optimization problem. Given N number of cities that a salesman has to visit, in what sequence do you travel through each city to minimize the distance that you travel?

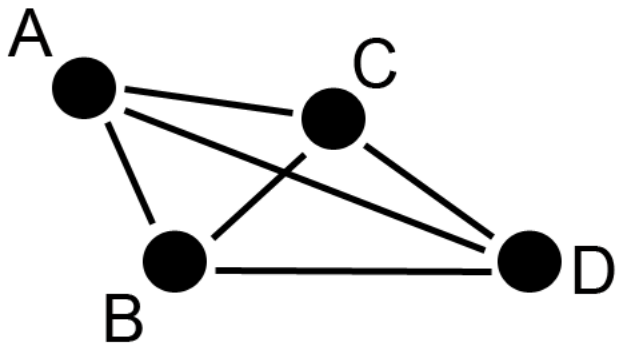

Figure 1.2: The Traveling Salesman Problem

From the diagram above, you can see that there are N! number of possible sequences for the N number of cities. Although it is easy to go through the simple example above to find the best route, this becomes computationally difficult as the number of nodes (cities) increases.

There are two possible ways of solving this problem.

- **Heuristics** - defined as a "trial and error" method of problem solving. Heuristics involves a lot of informal guesses that tend to lead to the correct answer, but are not guaranteed to give the optimal solution. An example of this is starting at one node, you select the shortest distance from that node to any other node. At the next node, you again select the shortest distance to another node which is not already covered. Although you can see that this minimizes the pathway locally, it does not guarantee that the shortest pathway is chosen. Can you come up with a graph for which this algorithm does not provide you with the shortest path?
- **Complete Enumeration** - going through all possible N! solutions until the optimal path is found. The computational time required for complete enumeration is on the order of N!, or in big-oh notation: O(N!). O(N!) is known as combinatorial explosion and is a computationally NP-hard problem (Non-deterministic Polynomial time). Some problems have been proved to only be solvable by complete enumeration. Other methods might sometimes yield the optimal solution, but do not guarantee it for the problem.

1.4 Dimensionality and Finding Minima

Let's now review some basic calculus involved in finding extrema in a function. Recall that for a one dimensional system, we calculate the derivative of a function $f(x)$ to find the minimum, given a continuous and differentiable function:

$$\frac{df}{dx} = 0 \tag{1.1}$$

To ensure that we have a minimum, we look to the second derivative, such that

$$\frac{d^2f}{dx^2} > 0 \tag{1.2}$$

Remember that the second derivative must be positive since for a minimum to occur, the gradient of the function must go from being negative to positive.

The first derivative tells us if there is a change in the monotonicity of the function while the second derivative guarantees us the minimum. From a two dimensional system, $f(x, y)$, we also can look to the gradient and the second derivative (this time, the Hessian) such that:

$$\nabla f = 0 \tag{1.3}$$

And that:

$$H = \begin{pmatrix} \frac{d^2f}{dx^2} & \frac{d^2f}{dxdy} \\ \frac{d^2f}{dxdy} & \frac{d^2f}{dy^2} \end{pmatrix} > 0 \tag{1.4}$$

The condition above is known as **positive definite** (all elements in the matrix are positive), and this guarantees that we have a minimum. H is also positive definite if the following condition exists:

$$x^T Hx > 0, \forall x \tag{1.5}$$

The function is also positive definite if the eigenvalues of H are both positive for symmetric matrices (usually this matrix will be symmetric). The eigenvalues being positive is related to the function being convex in both directions. Recall that if one eigenvalue is positive and the other negative, then the point is a saddle node.

1.5 Steepest Descent and Newton's Method for Optimization

Let us return to our problem of continuous optimization where we want to minimize the function $f(x)$, now given that it is positive definite. Although it is easy to look at a graph and find the minimum, what tools are available in Matlab which will allow us to perform optimization without using graphical methods? There are two main methods for approaching optimization which you will explore more in the problem sets.

- **fsolve()** function in Matlab is an implementation of the steepest descent method and finds the zeros of a function (so one must supply the derivative of a function to obtain an extremum)
- **ode45()** function performs the analysis over time of a function if we can define the function to be minimized in terms of a potential function (and similar to fsolve(), the gradient of the function must also be supplied)

On using ode45() for optimization:

- Let $f(x)$ be our energy function such that if we integrate $\frac{dx}{dt} = -\nabla f$ with respect to time, then we are assured that our energy function will decrease along all trajectories.
- We must supply ode45() with an initial condition to run and with this method, we are guaranteed a minimum, but not necessarily the global minimum. Only if the function is convex that we are guaranteed a global minimum and hence convex functions are easier to optimize than non-convex functions.

How do you know which function to use when you are performing your optimization? Let's consider the similarities and differences between fsolve() and ode45() to get a better idea of how they work and how they can be useful.

Similarities between fsolve() and ode45():

- Both must be supplied the gradient function in order to work
- Both must sample many initial conditions, in fact an infinite number of them, to guarantee that the global minimum is achieved. This is particularly important for energy functions with very narrow wells.
- Both take large time steps if possible to maximize the change in gradient and speed up calculations

Differences between fsolve() and ode45():

- The output of fsolve() is the x value at which an extremum occurs
- The output of ode45() is a time trace of the x value as time progresses and it is the final value of x over time which determines the value at which the extremum occurs. See below for an explanation as to why this is the case.
- The time steps of ode45() can be changed to suit the needs of the user, which is not available for fsolve().

Convex functions - functions in which for all points in x that our Hessian is positive definite (will only have one extremum for the entire function). Convexity is the result of the variables and if we can redefine the variables such that our function becomes convex, then this makes optimization much easier.

In the computer, steepest descent is performed by the Euler method of integration given our function:

$$\frac{dx}{dt} = -\nabla f \tag{1.6}$$

The Euler method expands out the differential to give:

$$x_{t+1} = x_t - (\nabla f)\delta t \tag{1.7}$$

The computer adjusts δt to make it more efficient to reach the minimum and will minimize ∇f as much as possible until it reaches a given tolerance value (ε) such that:

$$\| x_{t+1} - x_t \| < \varepsilon \tag{1.8}$$

When using ode45(), the computer integrates using the above Euler's method and evaluates the derivative after every time step. When the derivative equals to zero, then $x_{t+1} = x_t$ and the value of x will stop changing and be constant over time. This is why when we use ode45() to obtain our extrema, then we must take the final value of x to be the point where the extremum occurs.

Possible problems with steepest descent:

- If the function has a very narrow well with the minimum, then the initial conditions must be very close to the minimum to achieve optimization
- If the function has several minima of various depths, then many initial conditions must be used to guarantee that we obtain the global minimum of the function.
- Convergence of steepest descent is related to how fast you get to the solution. For very flat, but convex functions, then this will take relatively long to converge since we are not sure of the step size that we should be taking because the variation in the function is too small. To correct for flatness in a function we can use **conjugate gradient methods** which don't always take the path against the gradient but will take steps out of the direction of the gradient (similar to going up the ridges of a valley) before again taking steps in the direction of th gradient.
- Although we might have a convex function, steepest descent will only get to the minimum of a function given an infinite time (remember that this only works for continuous functions).

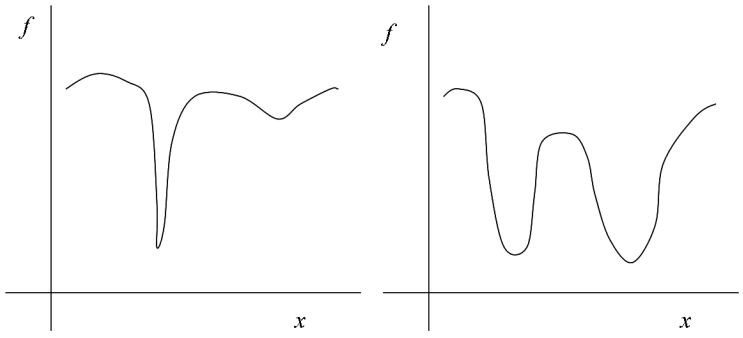

Figure 1.3: Graphs that pose problems for steepest descent algorithm

Can we use steepest descent for combinatorial optimization? Yes! For example, you take take your discrete solution $x_0 = [\mu_1, \mu_2, \ldots, \mu_N]$ and change

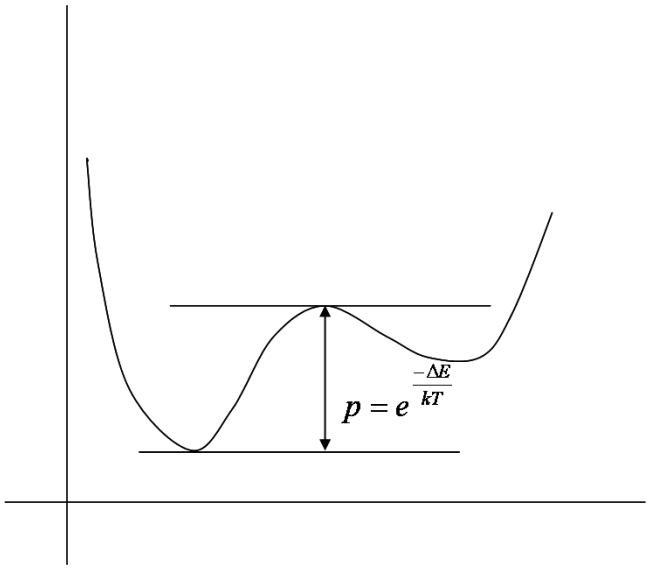

Figure 1.4: The probability of a jump occuring is related to the Boltzmann distribution

the state of each μ that minimizes the cost function. In effect, steepest descent is similar to a greedy algorithm in that it can minimize each μ locally in an attempt to minimize the cost function globally. The only difference between using steepest descent instead of simulated annealing with is that we allow upward changes in energy for simulated annealing and not for steepest descent.

1.6 Combinatorial Optimization with Greedy Algorithms

For discrete sets in combinatorial problems, we utilize greedy algorithms which takes a starting condition and looks to it's nearest neighbors and minimizes that distance before continuing. Remember that without complete enumeration, this will not necessarily give the best solution and may get caught in large distances later on in the algorithm. This algorithm minimizes locally in an attempt to minimize globally and this is no longer an O(N!) problem anymore, making it computationally easier than complete enumeration. A good example of a greedy algorithm is Dijkstra's algorithm for finding the minimum pathway through a network. Other minimum tree spanning algorithms include Kruskal's or Prim's Algorithm.

1.7 Formulating a Combinatorial Optimization Problem

Although it is easy to see how we can optimize combinatorial problems, it may be more difficult to formulate the problem in terms of equations which we want

to minimize. For an example, consider a circuit design where we are given N circuits to be divided into two different connected chips. The traffic matrix A is the interchip cost matrix or our "energy" look up table in the form:

$$A_{ij} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \tag{1.9}$$

We first define the state for each circuit being either $\mu = \pm 1$ where if the value of $\mu = +1$, then it is on the first chip, and if the value of $\mu = -1$, then it is on the second chip.

The state of the system will be given by the list of all the circuits: $[\mu_1, \ldots, \mu_n]$. It can be seen that there are 2^n possible solutions for the circuit design.

But what exactly do we want to optimize? First, we want to reduce the interchip traffic, or the energy required to transfer information between chips. All related chips (as defined by a very low value in the matrix A) would ideally be together on one chip by having only a very limited number of connections between the two chips. Second, we would like to make the size of each chip roughly equivalent. We don't want it such that all the circuits are contained on one chip while the other chip only has a few circuits.

How do we first deal with the interchip traffic? Because we defined each circuit to having a state of ± 1, this makes it relatively easy. If they are on the same chip, then $\mu_i - \mu_j = 0$ and we can formulate the total cost of the function as:

$$u_{traffic} = \frac{1}{2} \sum_{i,j} (\frac{\mu_i - \mu_j}{2})^2 A_{ij} \tag{1.10}$$

Now to deal with the sizes of both chips, we can just sum up the values of the -1 and +1. If they are the same size, then the sum will be equal to 0.

$$u_{size} = (\sum \mu_i)^2 \tag{1.11}$$

The total cost function that we now want to minimize is:

$$u_{total} = \frac{1}{2} \sum_{i,j} (\frac{\mu_i - \mu_j}{2})^2 A_{ij} + (\sum \mu_i)^2 \tag{1.12}$$

We can also give different weightings (λ) to each cost value to change the importance of the size of the chips or the interchip traffic by multiplying with a factor:

$$u_{total} = \frac{1}{2} \sum_{i,j} (\frac{\mu_i - \mu_j}{2})^2 A_{ij} + \lambda(\sum \mu_i)^2 \tag{1.13}$$

1.8 Simulated Annealing

This method attempts to correct for the presence of several minima and is a heuristic algorithm for non-convex problems derived from materials physics. We slightly adapt the Newton method by including a random term which will cause our function to sometimes go against the gradient.

$$x_{t+1} = x_t - (\nabla u)\delta t + \alpha w(0, \sigma) \tag{1.14}$$

- $w(0, \sigma)$ is a randomly generated number from a normal distribution with mean centered at zero and a standard deviation σ.
- α is our control parameter (e.g. temperature) to determine the "jumpiness" of the step. The function usually begins with a high temperature and slowly decreases α allowing a complete exploration of the energy function. If we can adjust the temperature infinitely slowly, then we are guaranteed to be at the global minimum. The adjustment of α over time is known as the **cooling schedule** and is arbitrarily set. For some energy functions, a initially slow cooling period is required, then rapidly cools down after a certain number of time steps. For generally more convex functions, it is possible to decrease the temperature quickly at the beginning and slowly towards the end. Several cooling schedules are usually used to see which one obtains the best results.

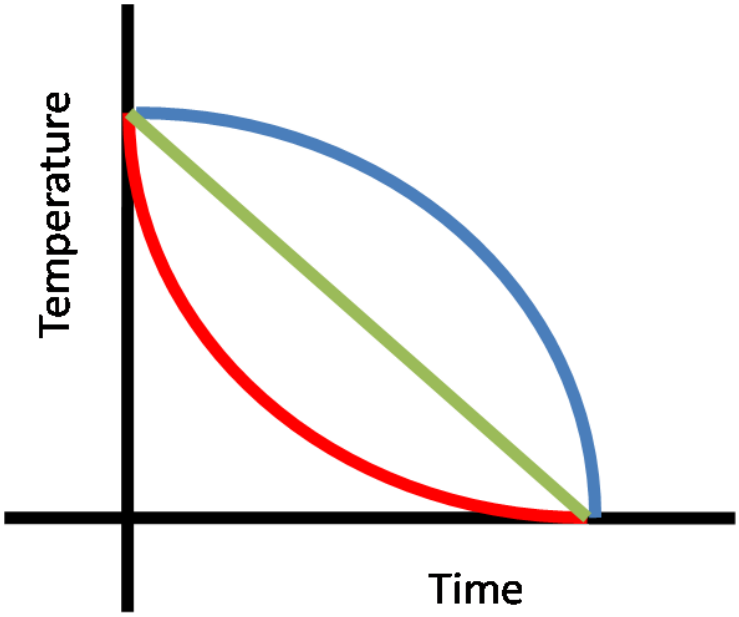

Figure 1.5: Three Different Cooling Schedules

- For any change in energy that decreases, then we will always accept the change, but for an increase in energy, we will go upwards with a certain probability defined by the Boltzmann distribution:

$$p = e^{\frac{-\Delta E}{kT}} \tag{1.15}$$

Pseudocode for the Simulated Annealing Algorithm (can be used for both standard or combinatorial optimization problems):

1. Select an initial state x_0 at random and evaluate the energy $u(x_0)$.

2. Select x_1, the next state at random and evaluate the energy $u(x_1)$.
3. Accept the change if the cost function is decreased
4. Otherwise, accept the change with a probability $p = e^{\frac{-\Delta E}{kT}}$, where δE is the difference in energy between the new state and the previous state
5. Decrease T (temperature) according to the cooling schedule
6. Repeat steps 2 - 5 until we reach the desired number of time steps

An example of simulated annealing from Lawrence David at MIT:

Imagine that there is a large hilly region with several peaks and valleys and one small fish. You, God, suddenly made it rain very hard and the entire region was filled with water (remember Noah and the ark?). Now because the water level is quite high, the fish can explore almost every part of the region. However, now that you've finished cleansing the land of all the sinners, you want to reduce the water level so that man can once again survive. You slowly reduce the water level. The fish however, wants to live as long as possible so will tend to explore all the places until he has found the lowest point in the region. By reducing the water level, the fish sometimes can get stuck in local valleys (minima), but if we can reduce the water very slowly, then the fish will eventually be able to seek out the lowest valley and survive.

Now you might be asking what this is all about? But this is similar to the simulated annealing problem in that the water level is equivalent to the temperature and the highest point is what our Boltzmann probability maximum that we can jump. As God, you control the cooling schedule until your fish reaches the global minimum.

1.9 Genetic/Evolutionary Algorithms

This method is only for combinatorial problems and was inspired by genetics. Evolutionary algorithms are generally faster and get around some of the problems found in simulated annealing. Instead of minimizing an energy function, we maximize a "fitness" when we use evolutionary algorithms. The advantages of using this algorithm over simulated annealing are the **parallelism** (faster because you are considering several solutions at each step) and the **increased exploration of state space** with "reproduction/crossovers" and "mutations" (ie random factors) of each solution. Remember that this is a heuristic algorithm and that there are no guarantees that the optimal solution will be found.

We first define our function u in state space with a discrete set of solutions.

1. Start out with a population of randomly generated solutions (initial guesses) of size P
2. **Reproduction** - take random pairs of parents and create their offspring (generating a total of $\mathrm{P} + \mathrm{P}/2$ solutions), reproduction is achieved through mutating the strands and then performing crossovers.
    - **Mutation** - Mutate each solution with a given probability at each location

- **Crossover** - Move entire portions of the solution to different positions from the two randomly selected parent solutions

3. **Rank** the population according to their energy $u(x)$
4. **Eliminate (Selection)** the bottom P/2 to optimize $u(x)$ leaving again a population of size P.
5. Repeat from the reproduction step until you have achieved the number of time steps that are desired.

Pseudocode for an Evolutionary Algorithm:

```
generation = 0;
  initialize population
  while generation < max_generation
   for i from 1 to population_size
    select two parents
    crossover parents to produce a child
    mutate child with certain probability
    calculate the fitness of each individual in population
   end for
   eliminate the bottom P/2 members
   generation++
   update current population
  end while
```

1.10 Constrained Optimization

If we have constraints that are in the form of **equalities**, then we want to use **Lagrange multipliers** to solve for the solution. An example of this is to find out which distribution yields maximum entropy.

Below is adapted from the Wikipedia page on Lagrange Multipliers:

Suppose we wish to find the discrete probability distribution with maximal information entropy. Then

$$f(p_1, p_2, \ldots, p_n) = -\sum_{k=1}^{n} p_k \log_2 p_k \tag{1.16}$$

Of course, the sum of these probabilities equals 1, so our constraint is

$$\sum_{k=1}^{n} p_k = 1 \tag{1.17}$$

We can use Lagrange multipliers to find the point of maximum entropy (depending on the probabilities). For all k from 1 to n, we require that

$$\frac{\partial}{\partial p_k}(f + \lambda(\sum_{k=1}^{n} p_k - 1)) = 0 \tag{1.18}$$

(see that $\sum_{k=1}^{n} p_k - 1 = 0$ allowing us to do the above operation) which gives

$$\frac{\partial}{\partial p_k}\left(-\sum_{k=1}^{n} p_k \log_2 p_k + \lambda(\sum_{k=1}^{n} p_k - 1)\right) = 0 \tag{1.19}$$

Carrying out the differentiation of these n equations, we get

$$-\left(\frac{1}{\ln 2} + \log_2 p_k\right) + \lambda = 0 \tag{1.20}$$

If we now solve for p_k:

$$p_k = 2^{\frac{1}{\ln 2} - \lambda} \tag{1.21}$$

This shows that all p_i are constant and equal (because they depend on λ only). By using the constraint $\sum_k p_k = 1$, we find

$$p_k = \frac{1}{n} \tag{1.22}$$

Hence, the uniform distribution is the distribution with the greatest entropy.

If the constraints are in the form of inequalities, then we want to use linear programming (used in the field of operations research and pioneered by Dantzig with the Simplex Algorithm). Because all our constraints are linear, then the feasible region is a convex polytope, so that the optimal solution will be at one of the vertices. This can be expanded to N-dimensions, except now, we cannot use a graphical method for solving. However, since we only care about the vertices of the polytope, we can go through each vertex in turn to maximize the cost function. This is known as the **Simplex Algorithm** which efficiently checks each of the vertices in our polytope until an optimal solution is found. First, we find one vertex and check if it's optimal. Then, we change one of the variables to get the next vertex and move on to increase the cost function maximally. Check each vertex to see if it is optimal until you have maximized the cost function. Computationally, the Simplex Algorithm is also combinatorial (ie O(N C N-m)) where N is the number of variables and m is the number of inequalities.

1.11 Example: Linear Programming

The Problem:

You are managing a farm that has 45 Ha in surface and you want to split your cultivated surface between wheat (W) and corn (C). The amount of labor that you can use is limited at 100 workers. Each hectare of wheat requires 3 workers while each hectare of corn requires 2 workers. The amount of fertilizer needed is 20 kg per hectare of wheat and 40 kg per hectare of corn. You can only use up to 1200 kg of fertilizer. When you go to the market, for each hectare of wheat, you will get a profit of $200 while each hectare of corn returns $300.

The Solution:

The first thing to do in any linear programming problem is to identify the constraints and the cost function that we wish to maximize. In the above problem, there are three limitations to the total amount of wheat and corn that can be grown: land, labor and fertilizer limitations. Let us first consider the land limitations. The total number of hectares of wheat (W) and corn (C) must be less than or equal to 45. Writing this as an inequality (Constraint 1):

$$W + C \leq 45 \tag{1.23}$$

For the labor limitations, we cannot exceed 100 workers (Constraint 2):

$$3W + 2C \leq 100 \tag{1.24}$$

For the fertilizer limitations, we cannot exceed 1200 kg of fertilizer (Constraint 3):

$$20W + 40C \leq 1200 \tag{1.25}$$

And finally, our cost/profit function which we wish to maximize:

$$Profit = 200W + 300C \tag{1.26}$$

Since this is only a two dimensional problem, we can solve this using a graphical method. We plot all of the constraints to come up with a feasible region, or the region of solutions which the problem satisfies all of the conditions.

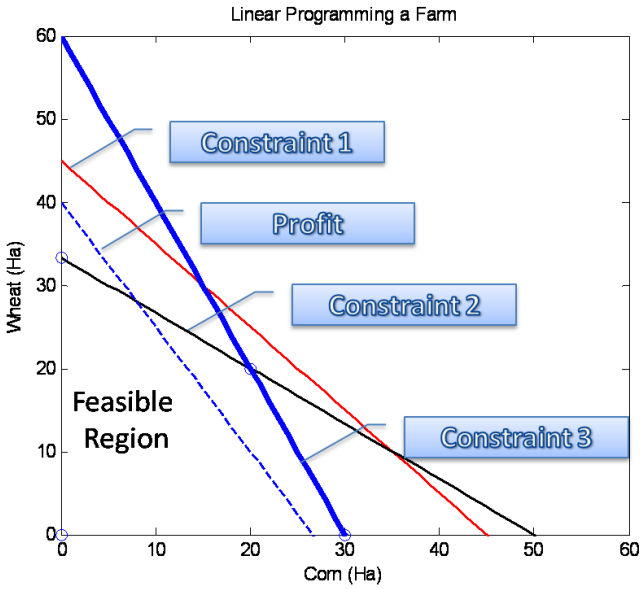

Figure 1.6: Linear Programming Example

With a graphical method, it is easy to see that the optimal solution will occur at (20,20), the last point which the profit function touches of the polytope if you continue moving it upwards keeping the same slope. This corresponds to farming 20 hectares each of wheat and corn. Although we have not optimally used all of the land, we have maximized the profit given the conditions.

Is the upper right corner of the polytope always the optimal point of operation? By no means is this the optimal point at which to operate. The optimal point is determined by the profit function. For example, if the profit for each hectare of corn is only \$1 while the profit for wheat is \$200, then it is easy to see that we should produce all wheat (corresponding to the upper left point of the polytope.

1.12 Standard Least Squares Optimization

Least Squares Optimization is used for data fitting given one or more independent variables and one dependent variable. We establish the relationship between the variables either by a theoretical relationship or by observation and we wish to know which "line" best describes the data obtained from experimentation. For standard least squares, this problem has an analytical solution.

We have a collection of N points (x_i, y_i) and we have strong belief that this dependency is linear. We must assume that x is an independent variable and that there is no error involved with the measurement of this parameter. We would like to find out the coefficients of the following equation that best describes the data.

$$y = a_0 + a_1 x \tag{1.27}$$

For the least squares method, we optimize the distance between the predicted line and the actual points that we have. We have a **overdetermined** system (a system where we have too few variables for the number of equations) which we can write in matrix form below:

$$\hat{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} = a_0 \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} + a_1 \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_N \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = Xa \tag{1.28}$$

The error function is now the difference between the predicted values and the observed values:

$$e = \hat{y_i} - y_i \tag{1.29}$$

For the least squares optimization we would like to minimize $e^T e$

$$\begin{aligned} e^T e &= (\hat{y_i} - y_i)^T (\hat{y_i} - y_i) \\ &= \hat{y}^T \hat{y} - \hat{y}^T y - y^T \hat{y} + y^T y \end{aligned} \tag{1.30}$$

But we know that $\hat{y} = Xa$, so substituting, we get:

$$\begin{aligned} E = e^T e &= (Xa)^T (Xa) - (Xa)^T y - y^T (Xa) + y^T y \\ &= a^T X^T X a - a^T X^T y - y^T X a + y^T y \end{aligned} \tag{1.31}$$

To minimize the error, we want $\nabla E_a = 0$:

$$\begin{aligned} \nabla E_a &= 2X^T X a - 2X^T y = 0 \\ a &= (X^T X)^{-1} X^T y \end{aligned} \tag{1.32}$$

This can be expanded into N variables, each with a linear dependence. The result of this is that it will find the best hyperplane in $m - 1$ dimensions if we have m number of independent variables.

Let us compare the above with a determined system where there are enough equations to find the coefficients exactly.

$$y_1 = a_0 + a_1 x_1 \tag{1.33}$$
$$y_2 = a_0 + a_1 x_2 \tag{1.34}$$

We can easily rewrite this in matrix form:

$$\vec{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \vec{X}\vec{a} \tag{1.35}$$

Now we can see to solve for the coefficients, we can just take the inverse matrix of X:

$$\vec{a} = (\vec{X})^{-1}\vec{y} \tag{1.36}$$

Compared to the overdetermined system above calculated for the standard least squares, we can see a similarity between the matrices inverted. Because we are "inverting" a rectangular matrix for the least squares method and not a square matrix, we call this the **pseudoinverse**. (Note that the inverse of a matrix can only be taken on a square matrix)

Pseudoinverse = $(X^T X)^{-1} X^T$

In Matlab, this is implemented by the function **pinv()**.

1.12.1 Example: Standard Least Squares for a Straight Line Fit

If we reinvert the matrix, we obtain that

$$X^T X a = X^T y \tag{1.37}$$

Let us take the four points (0,0), (1,8), (3,8), and (4,20) and perform standard least squares to find the line which best fits the points (minimizes the vertical error).

$$X = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} \tag{1.38}$$

$$y = \begin{bmatrix} 0 \\ 8 \\ 8 \\ 20 \end{bmatrix} \tag{1.39}$$

Performing the calculations:

$$X^T X = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 8 & 26 \end{bmatrix} \tag{1.40}$$

and

$$X^T y = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 4 \end{bmatrix} \begin{bmatrix} 0 \\ 8 \\ 8 \\ 20 \end{bmatrix} = \begin{bmatrix} 36 \\ 112 \end{bmatrix} \tag{1.41}$$

The system is then solved with

$$\begin{bmatrix} 4 & 8 \\ 8 & 26 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 36 \\ 112 \end{bmatrix} \tag{1.42}$$

By solving the system of equations, we obtain that

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \end{bmatrix} \tag{1.43}$$

Hence the best fit line through the points is $y = 1 + 4x$. Below shows the best fit line with the four points.

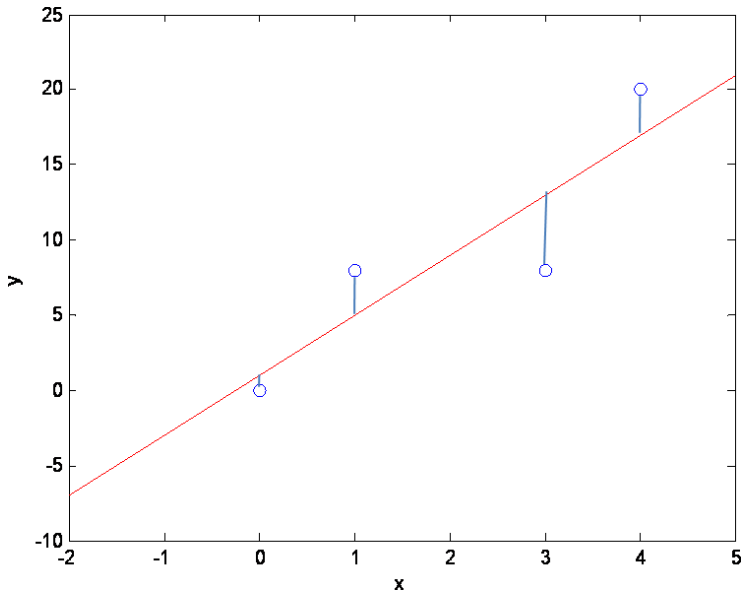

Figure 1.7: Linear Least Squares Example

1.13 Total Least Squares, Singular Value Decomposition, and Principal Component Analysis

Total least squares is a method to finding the best fit curve given that we don't know the independent and dependent variables of the system. For example, given several different parameters that one is looking at, we are unsure of which variables depend on the other. In effect, total least squares finds the best fit line such that the error minimized is the perpendicular distance between the

measured point and the optimal line (as opposed to the vertical distance in standard least squares).

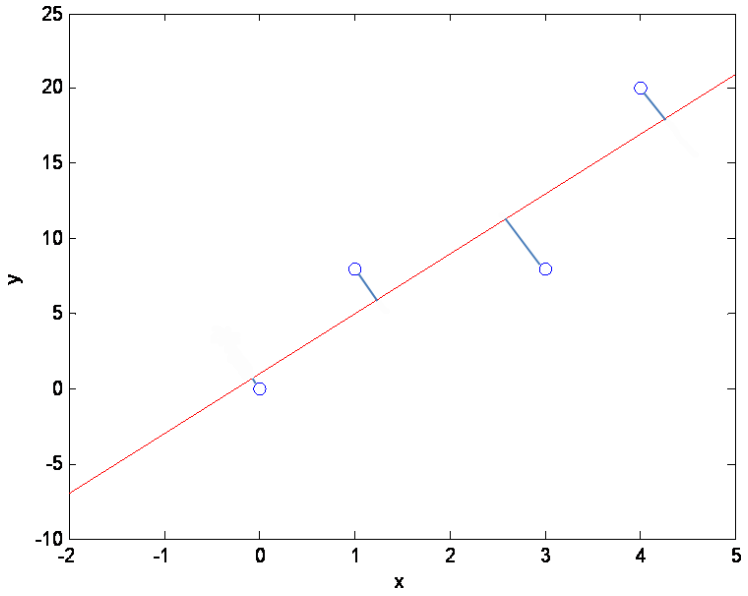

Figure 1.8: Total Least Squares Method utilizing the perpendicular distance between the point and the best-fit line

Singular Value Decomposition (SVD) yields the covariance matrix between the variables that we have and the matrix of principle components from which we can perform Principal Component Analysis. Singular value decomposition is equivalent to the diagonalization of rectangular matrices and typically yield a series of ranked numbers. The largest of these values are known as the principal components of the system and allow us to determine which of the variables are most strongly correlated to one another. For example, given 5 variables and our SVD yields that 2 of the variables are much higher than the other, we have 2 principal components which should be sufficient to describe the trends in the data and the other variables have little or no correlation to what is being observed. Using PCA allows us to discard redundant variables in our system leaving what is known as the *lower rank approximation* to the system.

The decomposition of a matrix X yields the following:

$$X = U\Sigma V^T \tag{1.44}$$

Where U and V are unitary matrices and σ is the matrix with the singular values. Usually, the singular value decomposition is preformed on the matrix of the variables with the mean subtracted from all the data points and not usually on the raw data. This allows an easier interpretation of the decomposition. The covariance matrix can then be calculated from the altered data and the singular values correspond to the eigenvalues of this $n \times n$ covariance matrix (with n number of variables).

To obtain the lower rank approximation, we reduce the number of rows of σ such that it becomes an $r \times r$ matrix, where r is the number of variables.

The reconstructed values of X now take into account only the singular values with the highest value and discard those with lower values that are may not be pertinent to explaining the trendline. For example, you might have singular values of 200, 100, 1 and 0.5. In this case, if you wanted to reduce the number of variables in the problem, the lower rank approximation would be a 2×2 matrix with only singular values of 200 and 100. The variables that correspond to these highest singular values are your principal components of the system.

1.14 Artificial Neural Networks

Optimization within artificial neural networks can also be a challenging problem. We first define a neural network through the **perceptron**, or a series of inputs, hidden nodes, and outputs in a funnel like structure which allow fast computation of certain types of problems. Connections exist between layers but do not exist within layers. Each edge contains a weight by which the input from the previous layer is multiplied to obtain the next layer.

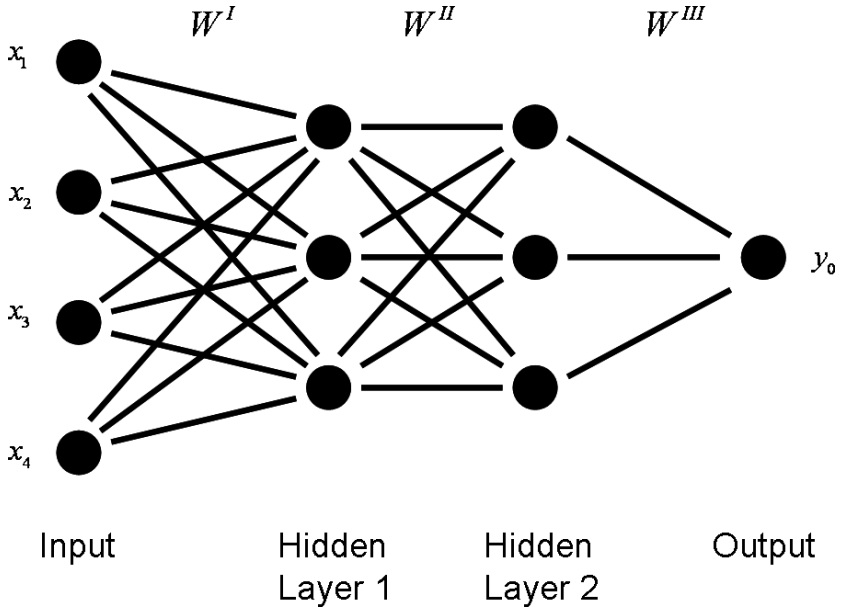

Figure 1.9: An example of a perceptron

Given that the input into the perceptron is x_0, what will our output be with the two hidden layers above? Consider the first hidden layer. The input into the first hidden layer is:

$$\chi^I_{l\times 1} = W^I_{l\times n}\vec{x}_{n\times 1} \tag{1.45}$$

We apply a non-linear function $f(x)$ at the hidden layer such that the output from the first layer and the input into the second hidden layer is:

$$\chi^{II}_{p\times 1} = W^{II}_{p\times l} f(\chi^I) \tag{1.46}$$

Similarly, our output of the perceptron is:

$$y = W^{III}_{1\times p} f(\chi^{II}) \tag{1.47}$$

It can be seen that these are nested expressions, so if we want to express the output in terms of the input and the weightings, we obtain the equation:

$$y = W^{III}_{1\times p} f(W^{II}_{p\times l} f(W^{I}_{l\times n} x_{n\times 1})) \tag{1.48}$$

The output is a nested series of functions that are weighted by each node in the system. This is usually a non-trivial solution and very non-linear. The weights of the edges are usually obtained through "learning", or taking several known inputs and outputs and allowing the weights of each to change such that the known output is obtained. The three types of learning are used and are derived from biological heuristics: the Hebbian Rule, the Darwinian Rule, and Steepest Descent.

The Hebbian Rule (potentiation) is based upon the theory that synapses that are used most are strengthened, ie the weights of those edges are higher based on how many times they are used in the learning sequence.

The Darwinian rule initially assigns random weights to each edge and evolves them over time to minimize the error in the output.

Steepest descent attempts to minimize the error function similar to the steepest descent methods described earlier. We can define our desired output as:

$$\hat{y} = W^{III} f(W^{II} f(W^{I} \vec{x})) \tag{1.49}$$

The error involved between the desired output and the observed output y can be minimized with the function:

$$E = \left\| y - \hat{y} \right\|^2 \tag{1.50}$$

All that is required now is to optimize $\frac{dE}{dW}$ by following the decrease in the gradient to obtain the weights that give the minimum error.

An implementation of this method is known as back propagation, where we first start from the output and calculate and minimize the local errors involved to find the best weights. Although back propagation is not a biological example, it is a heuristic algorithm that tends to work when training neural networks.

Chapter 2

Discrete Systems

2.1 What is a discrete system?

Discrete systems are systems with non-continuous outputs with the result for each time step being determined by the previous time step. The equivalent to differential equations (continuous systems) in discrete system are known as difference equations such as the one shown below. Since the difference equation tells us about the next time step, it is known as a 1st order difference equation.

$$x_{t+1} = rx_t \tag{2.1}$$

This is the discrete equivalent of the differential equation:

$$\frac{dx}{dt} = rx \tag{2.2}$$

Discrete systems are found in nature, for example in biology where reproduction occurs at discrete time steps. Also, sampling at discrete time intervals of a continuous system gives us discreteness.

2.2 How do we solve difference equations?

1) By guessing (*ansatz* - German for "guess")

Let us assume that the solution is in the form

$$x_t = A\beta^t \tag{2.3}$$

Then substituting into the difference equation, we get

$$A\beta^{t+1} = rA\beta^t \tag{2.4}$$

$$r = \beta \tag{2.5}$$

And hence the solution to the difference equation is

$$x_t = x_0 r^t \tag{2.6}$$

2) Z-transforms - the discrete equivalent to the Laplace transform, an organized way to find solutions

2.3 What are the behaviors of the solution $x_t = x_0 r^t$?

There are 4 different regimes, depending on the value of r.

- If $r > 1$, then x_t approaches infinity.
- If $0 < r < 1$, then x_t approaches zero.
- If $-1 < r < 0$, then x_t approaches zero in an oscillatory manner.
- If $r < -1$, then x_t approaches infinity in an oscillatory manner.

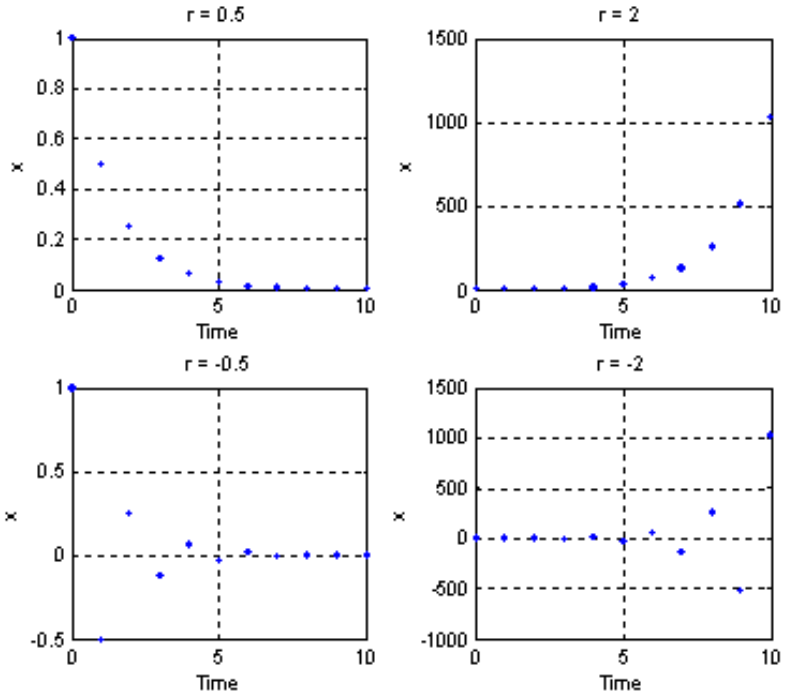

Figure 2.1: Plots of a discrete equations over time

Remember that if we have non-linear systems, it is difficult to obtain the global stability analysis of the problem, however, we can see from the above different behaviors that we can generalize it into stable and unstable behaviors as shown in the table below.

Stability Analysis

r States	**Stability**
$\|r\| < 1$	Stable
$\|r\| > 1$	Unstable
$r = 1$	Liapunov Stable

What does Liapunov stability imply? The fixed point is neither stable (goes towards the fixed point) nor unstable (goes away from the fixed point), but the trajectory is bounded. This is similar to a limit cycle or a center in continuous differential equations.

2.4 Higher Order Difference Equations

A second order difference equations would tell us about the state of the system two time steps away, for example:

$$x_{t+2} = ax_{t+1} + bx_t \tag{2.7}$$

How do we go about solving this system? We again solve by guessing for the correct solution. We have our initial guess again as $x_t = Ar^t$. Substituting into our original equation, we get:

$$Ar^{t+2} = aAr^{t+1} + bAr^t \tag{2.8}$$

The characteristic equation for this system is

$$r^2 - ar - b = 0 \tag{2.9}$$

Solving for r

$$r_\pm = \frac{a \pm \sqrt{a^2 + 4b}}{2} \tag{2.10}$$

Hence, our general solution will be the sum of all possible solutions, just like in second order differential equations

$$x_t = Ar_+^t + Br_-^t \tag{2.11}$$

We can then solve for A and B with our initial conditions.

For the stability of the system, we know that for the system not to blow up to infinity, $|r_+|, |r_-| < 1$. For this condition to be met, the following condition must be satisfied (can be solve for by setting $r = 1$).

$$b < 1 - a \tag{2.12}$$

Furthermore, we can generalize this solution for an N-dimensional system:

$$a_0 x_{t+N} + a_1 x_{t+N-1} + \cdots + a_N = 0 \tag{2.13}$$

We again make the guess that our solution is in the form $x_t = Ar^t$, substitute, and obtain our characteristic polynomial to be:

$$\alpha_0 r^N + \alpha_1 r^{N-1} + \cdots + \alpha_N = 0 \tag{2.14}$$

The roots of our polynomial give the values of r and they must all satisfy $|r| < 1$ for the entire system to be stable. With the root of the equation being: $\{r_1^*, \ldots, r_N^*\}$, our general solution becomes:

$$x_t = \sum_{i=1}^{N} A_i (r_i^*)^t \tag{2.15}$$

Solving these difference equations is non-trivial and computationally difficult. There are criteria (see Schur or Jury) to check to see if the parameters will lead to a stable conditions (which check to see if the roots are within the unit circle). We can only solve by hand to a maximum of 4 dimensions. With higher order dimensions, a computer becomes necessary to establish the roots of the equation.

2.5 Fourier Analysis and Discrete Time Analysis

We can first motivate this by an example. Let us consider a single sinusoid in the time domain.

$$y(t) = Asin(\omega t) \tag{2.16}$$

Is it possible to write this function in terms of a difference equation? Consider the time steps t and $t+1$ for a sinusoidal function:

$$y_t = Asin(\omega t) \tag{2.17}$$

$$\begin{aligned} y_{t\pm1} &= Asin(\omega(t \pm 1)) \\ &= A[sin(\omega t)cos(\omega) \pm sin(\omega)cos(\omega t)] \end{aligned} \tag{2.18}$$

If we now add y_{t+1} and y_{t-1} we obtain:

$$\begin{aligned} y_{t+1} + y_{t-1} &= 2Acos(\omega)sin(\omega t) \\ &= 2cos(\omega)y_t \end{aligned} \tag{2.19}$$

Hence we can conclude that:

$$y_{t+1} = 2cos(\omega)y_t - y_{t-1} \tag{2.20}$$

We can see that a sinusoid function in the time domain can be expressed in terms of a second order difference equation. In general, if we have a sum of N sinusoidal waves making up our function, we can write this as a difference equations with 2N terms. In mathematical terms, this can be expressed as:

$$\begin{aligned} y_t &= \sum_{m=1}^{N} a_m sin(m\omega t + \phi_m) \\ &\equiv \\ y_{t+2N} &= \sum_{m=0}^{2N-1} b_m y_{t-m} \end{aligned} \tag{2.21}$$

Recall that the first equation above is just the Fourier transform of a function in time domain. Hence, for any function in the time domain, this can be written as either a series of sinusoids using Fourier transform or as a difference

equation with two terms to each Fourier term. We can also see that the coefficients obtained from performing the Fourier transform are directly related to the coefficients of the difference equation that is generated.

The process of going from the time domain to the difference equation is known as Linear Discrete Time Analysis, since we can see that we obtain a linear difference equation from a sinusoidal function. This is analogous to getting a linear second order differential equation when we are considering the same sinusoid when we do continuous differentiation.

2.6 Cobweb Analysis

Cobweb analysis allows us to visualize easily the long term behavior of the system and allows us to predict stability of a discrete system. Below shows four different cobweb analysis for different values of r for the equation $x_{t+1} = rx_t$. We can still see that if $|r| < 1$, the system is stable and if $|r| > 1$, then the system is unstable.

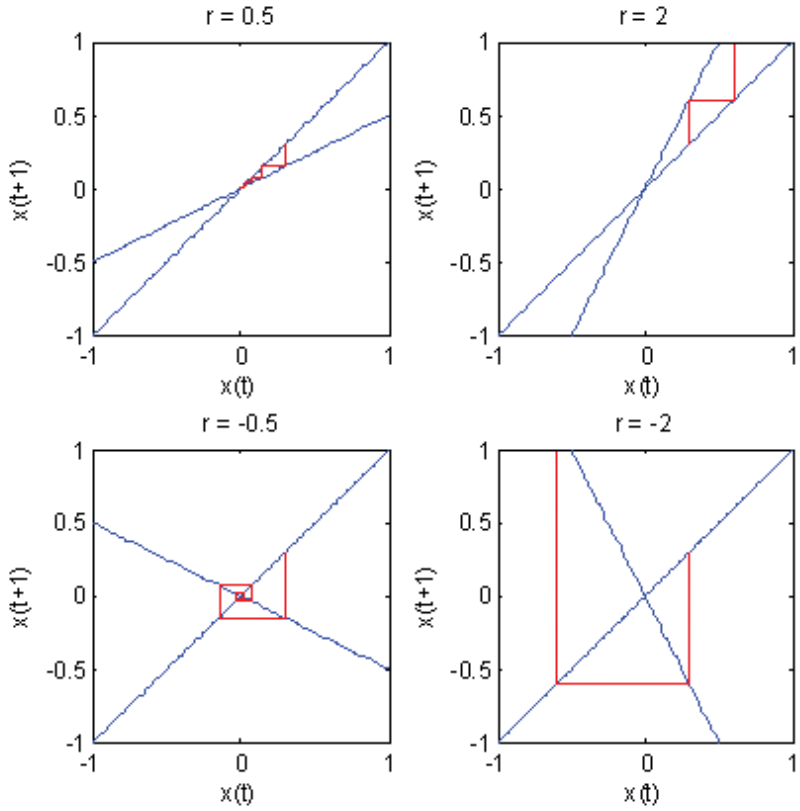

Figure 2.2: Cobweb diagrams can aid in stability analysis

Cobweb analysis is useful, especially in complex problems where an analytical solution may not be easy to obtain.

2.7 Logistic Maps

Let us now consider the logistic map equation as an example of how linear stability analysis is preformed on difference equations/systems.

$$x_{t+1} = rx_t(1 - x_t) \tag{2.22}$$

Where $0 < r < 4$ are the boundaries set.

First, to find the fixed points, we set $x_{t+1} = x_t$ to obtain the two fixed points:

$$x_t^* = 0, 1 - \frac{1}{r} \tag{2.23}$$

For linear stability analysis, we need that $\left|\frac{df}{dx_t}\right| < 1$

$$f(x_t) = x_{t+1} = rx_t(1 - x_t) \tag{2.24}$$

$$\frac{df}{dx_t} = r - 2rx_t \tag{2.25}$$

Now evaluating at the fixed points

$$f'(x_t^* = 0) = r \tag{2.26}$$

$$f'(x_t^* = 1 - \frac{1}{r}) = -r + 2 \tag{2.27}$$

Hence, the stability of the $x_t^* = 0$ fixed point is stable if $0 < r < 1$ (recall that we are bounded between $0 < r < 4$ for our system). The stability of the $x_t^* = 1 - \frac{1}{r}$ fixed point is stable for $1 < r < 3$.

After $r = 3$, we see that the system is unstable and that there are no fixed points. In fact, we will oscillate between fixed points and observe what is known as period doubling in our system. How do we calculate these fixed points analytically? If we have a period 2 oscillation, that means that

$$x_{t+2} = x_t \tag{2.28}$$

And from our function, we know that

$$x_{t+1} = f(x_t) \tag{2.29}$$

And by substitution

$$x_{t+2} = f(x_{t+1}) = f(f(x_t)) \tag{2.30}$$

In general, to solve for p period solutions, we want that

$$x_{t+p} = x_t \tag{2.31}$$

And we can solve

$$x_{t+p} = f(f...f(f(x_t)))_p \tag{2.32}$$

Returning back to our period-2 oscillations, we substitute our logistic equation into the x_{t+2} equation

$$x_{t+2} = f(f(x_t)) = r[rx_t(1 - x_t)][1 - rx_t(1 - x_t)] \tag{2.33}$$

We now set this equal to x_t and solve for the fixed points. As it can be seen that both our initial fixed points are fixed points of this system with two further fixed points being added. The two new fixed points are the points around which the system oscillates.

The bifurcations occur when the looking at the difference plot and seeing when you get more crossings on the graph. As the value of r is increased, the bifurcations occur more closely. After a certain value of r, you can get a point where there are no period solutions exist and this is the point where chaos emerges. Within the chaotic region, numerical analysis has yielded windows of periodicity such as period-3 oscillations, but quickly become chaotic again as r is changed.

With the analysis above, we can easily see that a fixed point is actually a period-1 solution to our system.