Chapter 9

Fitting to Data

In this chapter, we will look at techniques for fitting curves and models to data. We will look at three techniques, classical least squares of the sort that appears in Appendix B, least squares with an evolutionary algorithm, and least squares with genetic programming. These three methods, in the order given, go from fast, reliable, and restricted to slow, semi-reliable, and completely unrestricted. The sense of the word "restricted" will become apparent as we go through the chapter. The third method of fitting a model to data, genetic programming, has the advantage that you need not select the type of curve you are fitting before you start. The material in this chapter on genetic programming will build nontrivially on the material in Chapter 8, so a quick review is suggested before beginning this chapter.

9.1 Classical Least Squares Fit

In Appendix B, the equations for least squares fit to a line are given in Equations B.1 and B.2. The idea behind least squares fit is simple. Take a model of the data, e.g., "I think that these data come from a linear curve of the form $y = ax + b$" and then use a minimization technique to pick a and b to make the error between the model and data as small as possible. The unknown values a and b are the parameters of the model.

From our experience in Chapters 2 and 3 we know that quadratic curves give unimodal (e.g., easy) optimization problems. Looking at the derivation of the least squares fit to a line in Section B.4, we see that the problem of minimizing the squared error between a data set and a line $y = ax + b$ is exactly a quadratic minimization problem. Because it gives a method of estimating the parameters of our model that is a simple, quadratic minimization, we will use minimizing the square of the differences between our model and the data as our method of finding model parameters (and of finding the whole model when we use genetic programming). Here is an example of a least squares fit of a model to three dimensional

data.

Example 9.1 *Suppose we have 6 data points that we believe, except possibly for modest measurement error, lie in a plane in* $\mathbb{R}^3$*. The points are given in the table below. Our model will be the general plane*

$$z = ax + by + c$$

with parameters a, b, and c. To compute the sum of squared errors (SSE) we take the sum of the z-values minus the model's predicted z-values, squared.

i	x_i	y_i	z_i
1	0	0	1
2	0	1	4
3	1	0	3
4	1	1	6
5	1	2	9
6	2	1	8

So, to compute a, b, and c we want to find the values that minimize

$$SSE(a, b, c) = \sum_{i=1}^{6} (z_i - a \cdot x_i - b \cdot y_i - c)^2. \tag{9.1}$$

Evaluating the sum we get:

$$SSE(a, b, c) = 7a^2 + 7b^2 + 6c^2 + 10ab + 10ac + 10bc - 68a - 72b - 62c + 207.$$

Setting the partial derivative with respect to a, b, and c equal to zero we obtain:

$$\begin{aligned} 14a + 10b + 10c &= 68 \\ 10a + 14b + 10c &= 72 \\ 10a + 10b + 12c &= 62 \end{aligned}$$

Which has a unique solution:

$$\begin{aligned} a &= 2 \\ b &= 3 \\ c &= 1 \end{aligned}$$

And we see all the data points exactly fit:

$$z = 2x + 3y + 1.$$

In classical least squares fit, the first step is to choose the type of curve to fit to the data. The general term for this curve type is the *model*. The unknown coefficients in the model are called the *parameters* of the model. In Example 9.1, the model is a plane in 3-space, $f(x, y) = ax + by + c$, with parameters a, b, and c. (The data given in Example 9.1 were clearly taken from this plane. You can tell by plugging the data into Equation 9.1 and computing the squared error for $a = 2$, $b = 3$, $c = 1$; it is exactly zero.)

The problem we will be ultimately concerned with is finding the model to fit to the data.

Definition 9.1 *The* **sum-of-squared-error** *or* **SSE** *of a model with a data set is the sum over the points in the data set of the square of the value of the dependent variable subtracted from the value obtained by plugging the independent variables into the model.*

Examine the data given in Table 9.1. What function best models this data?

Table 9.1: A data set from an unknown model

x	**0.25**	**0.5**	**0.75**	**1**	**1.25**	**1.5**	**1.75**	**2**	**2.25**	**2.5**	**2.75**	**3**	**3.25**
y	**0.516**	**0.125**	**-0.0781**	**0**	**0.453**	**1.38**	**2.86**	**5**	**7.89**	**11.6**	**16.3**	**22**	**28.8**

One thing you could do is plot the data. The result of doing this is shown in Figure 9.1. The plot shows that the data are quite unlikely lie on a line, but gives little other information. The data might lie on a parabola with a minimum near x=0.75, but it is hard to tell by direct observation of a few data points. The Stone-Weierstrass Theorem says that any continuous function can be approximated arbitrarily well with polynomials. The data plotted do look like they come from a continuous function. This means we need only select the degree of the polynomial to fit. One way to do this is to fit higher and higher degree polynomials until the squared error stops decreasing. If we try this for the data given in Table 9.1, then we obtain the results given in Table 9.2.

Notice that the SSE drops radically from a degree 1 model to a degree 2 model, and again from degree 2 to degree 3. The sum of squared error actually goes up a bit (probably not significantly) when we move to a degree 4 model of the data. The least-squares-curve-fitting software used also found whole number values for the coefficients of the cubic fit as well; this is suggestive. Examining the quartic fit, we see the coefficients that were zero in the cubic fit remain quite small. (In fact, the data set was generated by plugging into $f(x) = x^3 - 2x + 1$, as can be seen in Figure 9.2.) Notice that the data are plotted as glyphs, while the model is plotted as a continuous line. This is standard practice for displaying data together with

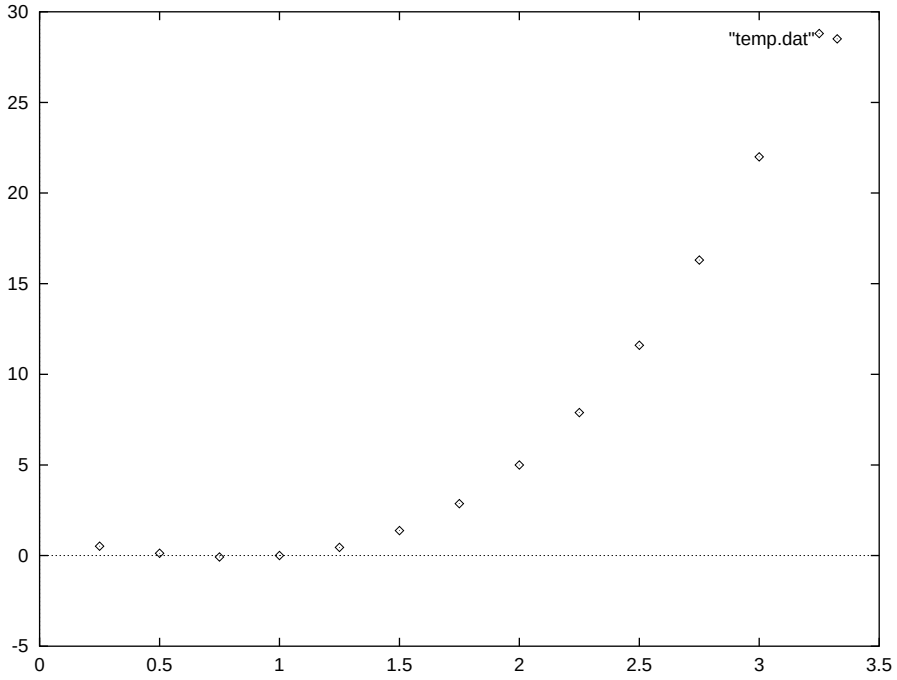


Figure 9.1: Plot of the data from Table 9.1

a model of that data. In situations other than toy examples, the model will not fit the data exactly and the minimum SSE will not be so close to zero.

It turns out that fitting polynomial models to bivariate data amounts to inverting matrices (see Problem 9.4). Least squares fit to nonpolynomial models can be quite daunting, involving very annoying algebraic manipulation. One special class of models, trigonometric series, can be fit using *Fourier series* techniques. Data resulting from time series are an excellent candidate for Fourier analysis and any course in time series and many courses in mathematical analysis treat Fourier series in depth. A time series is simply a set of data taken across a period of time with the measurement times saved as the independent variable on which the data depend. For example the time of dawn each day forms a time series.

Another attack on fitting of data relies on first transforming the data so that a simple model fits, fitting the model, and then inverting the transformation. Example 9.2 shows a standard example of this kind of transformation.

Example 9.2 *Assume we have data drawn from a system undergoing exponential growth:*

$$P(t) = A_0 e^{rt} + \epsilon(t), \tag{9.2}$$

with $P(t)$ being the population at time t, A_0 being the population at time $t = 0$, r being the growth-rate parameter, and $\epsilon(t)$ being the random error away from exponential growth. If we

Model	Least Square Fit	SSE
$y = ax + b$	$y = 8.75x - 7.86$	200.6
$y = ax^2 + bx + c$	$y = 5.25x^2 - 9.62x + 3.62$	5.03
$y = ac^3 + bx^2 + cx + d$	$y = x^3 - 2x + 1$	3×10^{-6}
$y = ax^4 + bx^3 + cx^2 + dx + e$	$y = 0.001x^4 + 0.993x^3 + 0.014x^2 - 2.01x + 1$	6×10^{-6}

Table 9.2: Results of fitting successively higher degree polynomials to the data given in Table 9.1

assume $\epsilon(t)$ is small and take the natural log, the model of the data becomes:

$$Ln(P(t)) = rt + Ln(A_0) \tag{9.3}$$

which is a linear function of 2 variables. Simply taking the natural log of the independent variable, permits us to fit a line to the data. The slope r of that line is the growth-rate parameter, while its intercept is the natural log of the initial population.

Problems

Problem 9.1 *In Example 9.1, we saw that to fit a plane to points in 3-space we needed 3 parameters: the unknown coefficients of x and y, and an unknown constant. A* term *in a multivariate polynomial has degree n, if the sum of the exponents of its member variables are n. Thus the third degree terms in a general 2-variable polynomial, minus their coefficients, would be: x^3, x^2y, xy^2, and y^3. When fitting a multivariate polynomial model, each term of each degree requires its own coefficient. Compute the number of unknown coefficients needed to do a least squares fit to a data set of an nth degree polynomial model with k variables. (This is the number of parameters needed to do the fit.)*

Problem 9.2 *Suppose that you wished to fit the model* **y=a**, *a constant function, to a data set $\{(x_i, y_i) : i = 1 \ldots n\}$. What type of value would you obtain, relative to the data, for the parameter a? Hint: since we didn't tell you the values of the data, we're not looking for a numerical answer; give a descriptive name.*

Problem 9.3 *Suppose you wished to fit a parabola, $y = ax^2 + bx + c$, to the data given below. Compute the SSE in terms of a, b, c and, using multivariate calculus, compute the values of a, b and c that minimize the SSE.*

x	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
y	**2**	**3**	**10**	**15**	**26**	**35**	**49**	**63**

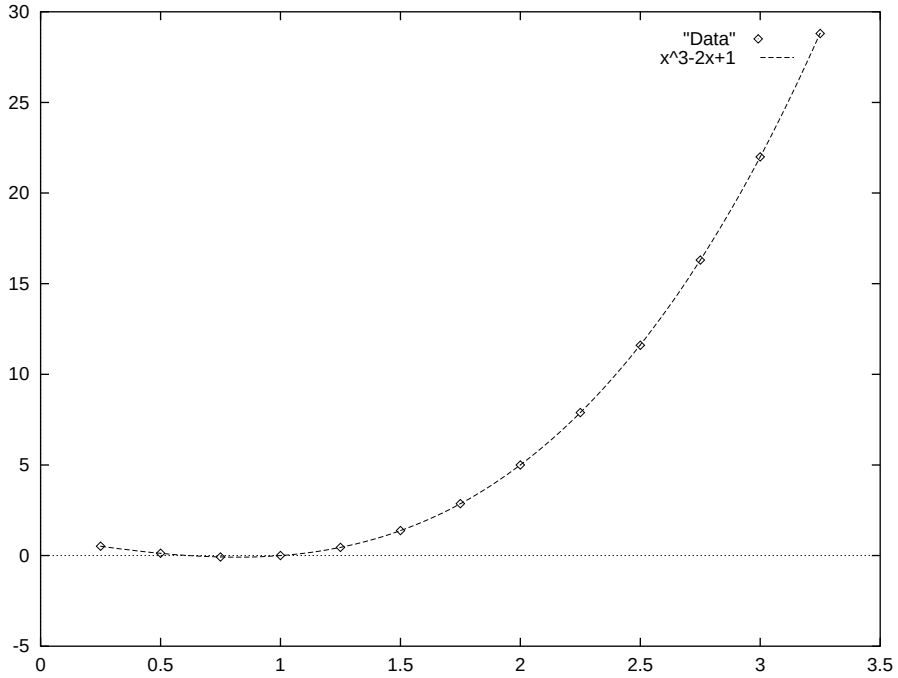


Figure 9.2: Plot of the data from Table 9.1 together with the best fitting polynomial curve

Problem 9.4 *This problem requires familiarity with simple linear algebra. Show, by algebraic derivation, that fitting a polynomial of the form* $f(x) = a_0 + a_1x + \cdots + a_nx^n$ *to a data set can be phrased as the solution of a matrix equation:*

$$A\vec{x} = \vec{b}.$$

Give the entries of the matrix A *and the vector* $\vec{b}$ *as summations over (functions of) the coordinates of the data points.*

Problem 9.5 *True or False. The total amount of deviation of the values of the model from the data points is all that matters when minimizing SSE. The distribution among the data points of the deviations from the model is unimportant.*

Problem 9.6 *For the population data given below, use the techniques outlined in Example 9.2 to find the growth rate for the population.*

Time:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Population:	**19**	**22**	**26**	**33**	**39**	**47**	**56**	**65**	**80**	**94**	**112**	**140**	**164**

You also obtain an estimate for the initial population, for which you have an observed value. Is there any reason to prefer the observed or estimated value? Explain.

Problem 9.7 *Problem 9.4 implies that fitting a univariate polynomial requires that we invert a matrix; in essence, the solving of linear equations suffices. Can we do a least squares fit of the model*

$$y = Sin(ax + b)$$

to bivariate data without solving nonlinear equations? Justify your answer, assuming an arbitrary number of data points.

Problem 9.8 *The data below were taken from a polynomial of degree no more than 5 with integer coefficients. Either find or write a program that can do a least squares fit of polynomial models to data. Construct a table like Table 9.2 and give the polynomial. Note that the values were rounded off to make the table and so will require some estimation on your part.*

x	**0.2**	**0.4**	**0.6**	**0.8**	**1**	**1.2**	**1.4**	**1.6**	**1.8**	**2**
y	**-2.84**	**-2.39**	**-1.69**	**-0.85**	**0**	**0.686**	**0.998**	**0.686**	**-0.538**	**-3**

Problem 9.9 Essay. *Explain why fitting a single, continuous curve to an entire data set may not be a good idea. Give at least one example where splitting the data set into two parts yields a better result than treating the data set as a unified whole. What sorts of processes in the natural world might produce such a data set?*

9.2 Simple Evolutionary Fit

In this section, we will adapt the real function optimizer from Chapter 3 to do a least squares fit to various models. We will introduce some evolutionary algorithms (which may be of some interest in their own right) to serve as sources of data for a least squares fit. We will start with a simple example of the art: fitting a line to data.

Experiment 9.1 *Write or obtain software for an evolutionary algorithm operating on genes of the form (a, b), where a and b are real numbers. Initialize a and b uniformly at random in the range $-10 \leq a, b \leq 10$. Let the fitness function be the SSE between a data set of points $\{(x_i, y_i) : i = 1 \ldots n\}$ and the model $y = ax + b$. This function is to be minimized, of course.*

The evolutionary algorithm should operate on a population of 100 genes using single tournament selection with tournament size 4. Use single point crossover and single point real mutation with mutation size $\epsilon = 0.2$. Call a population successful when its SSE drops to within 0.1% of the true minimal SSE. (You must compute the true minimal SSE using calculus - it may be handy to build the ability to do this into your software. See Equations B.1 and B.2 for a formula for the correct values of a and b.)

For each of the data sets below run the algorithm 50 times, saving the number of generations required for success. Plot the fraction of populations that have succeeded as a function

of the number of generations. Report which data sets are the most difficult to fit - a separate question from the quality of fit.

Data set 1:

x	0	1	2	3	4	5	6	7	8	9
y	2	2	6	6	10	10	14	14	18	18

Data set 2:

x	0	1	2	3	4	5	6	7	8	9
y	1	2	5	10	17	26	37	50	65	82

Data set 3:

x	0	1	2	3	4	5	6	7	8	9
y	4	0	4	0	4	0	4	0	4	0

Experiment 9.1 is intended mostly to get started with curve fitting. The data sets are, respectively: a line with periodic errors, a quadratic curve, and a simple periodic function. None of them really fit well with a linear model although the first data set comes close. An interesting question is whether being less like a line makes a data set easier or harder to fit with a linear model.

The next experiment is intended to make a point about mutation. When doing a least squares fit to data, the sensitivity to changes in one or another dimension in the space of parameters may not be uniform. Likewise, as we approach minimum error, smaller mutations are better at helping the algorithm converge.

Experiment 9.2 *Modify the software from Experiment 9.1 to use different mutation operators as specified below. For each of the mutation operators given, save and plot the time to success on data set 1 from Experiment 9.1 and compare them with one another and with the data taken for the first data set in Experiment 9.1.*

- *One point real mutation with mutation size* $\epsilon = 0.2$*, 50% of the time, and* $\epsilon = 0.02$*, 50% of the time.*
- *One point real mutation with mutation size* $\epsilon = 0.02$*.*
- *One point real mutation with mutation size* $\epsilon = \frac{0.2}{\sqrt{n}}$*, where* n *is the current generation number, starting at 1.*

- *Two point real mutation with mutation size* $\epsilon = 0.02$.

Which mutation operator gave the best performance? In your write up speculate as to the reasons for the differential performance.

One of the things that would be nice for curve fitting software to do is to automatically figure out the model for us. With the simple evolutionary fit, this requires something like a lexical product of fitness (defined in Section 5.1) in which the degree of the polynomial is used as a tie breaker when the SSE of two polynomials is essentially the same. Using the results shown in Figure 9.2, we can get a reasonable notion of "similar" SSE. The next experiment implements an evolutionary curve fitter that tries to find which model to use.

Experiment 9.3 *Modify the software from Experiment 9.1 to operate on a different sort of gene. The genes should have 7 real numbers and an integer. Use one point crossover with the integer being viewed as being in the gene before the first real number. The real numbers are the coefficients of a polynomial whose degree is specified by the integer. The first real is the constant term, the last is the coefficient of* x^6.

Mutation should be real single point mutation with mutation size $\epsilon = 0.2$ *(7 times in 8), and (1 time in 8) it should modify the integer degree by* ± 1*, with a lower bound of 1 and an upper bound of 6. Correct impossible mutations by not performing them. The coefficients above the degree given by the integer are unused, but are available for crossover.*

As before, we wish to minimize the SSE of the polynomial specified by the gene. When two genes have SSE fitness that differs by less than 0.1 take as more fit the polynomial that is lower degree.

Test the algorithm on the data sets given in Table 9.1 and Problem 9.8, running the algorithm 20 times for each polynomial. Report if the algorithm was able to find the degree correctly and if there was a consensus degree in the best members of final populations. Repeat the experiment on the first data set with the degree forced to always be 5. The program can reset the degree by zeroing out the higher degree coefficients: does it do so efficiently?

Now, we will analyze a simple artificial life system with evolutionary curve fitting software. This system is based on a game called the *Public Investment Game*. Suppose that you want to model public spending on something most people need, say roads. Clearly, it would be "fair" for people with similar amounts of money to pay the same amount for roads. However, a person who spends less than the others still benefits from the roads.

A simple model of this type of situation can be constructed as follows. A referee gives each person in the game \$100. Each person may secretly put some number of dollars in an envelope. The contents of the envelope are doubled and the result divided among all players evenly. The initial money represents each person's available money. The money placed into the envelope represents the money spent to build the roads: public spending. The doubling of the money in the envelope represents the value of having roads.

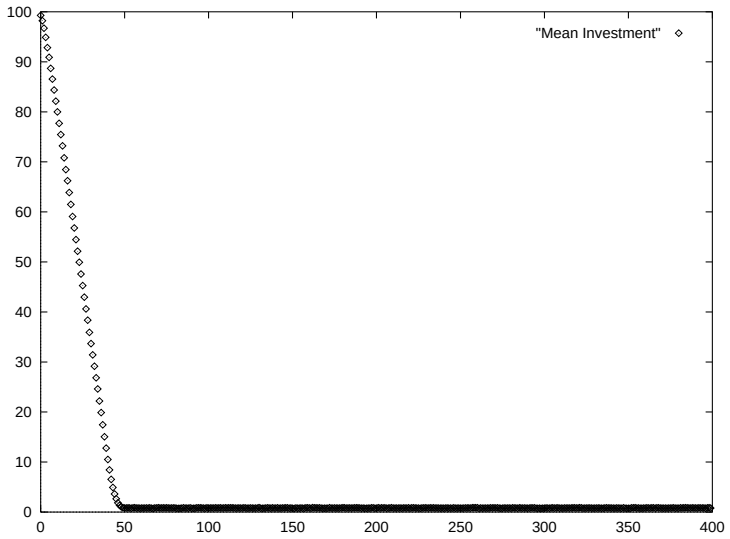


Figure 9.3: Average investment level of individuals over 50 populations from play 1 to play 400 of the Public Investment Game

The question we will examine is how simple strategies of play evolve under different conditions. As we will see, playing this game as a single shot game and then selecting for players that have the most money quickly leads to no public spending.

The basic software we will use works as follows. A population of 60 investors is represented by an array of integers initialized to a value of 100. The integer is the amount to put in the envelope. The population is shuffled into random groups of 12 investors that play the Public Investment Game, once. The fitness of each investor is 100 minus the money put in the envelope plus a share of the doubled investment money. The population is then shuffled into random groups of two investors. The investor in each pair with the lower fitness adopts the strategy (integer) of the investor with the higher fitness, plus a uniformly distributed integer in the range -5 to 5. Numbers above 100 or below 0 are mapped to 100 or 0, respectively. Ties are broken uniformly at random.

This is a very simple evolutionary algorithm mimicking a population of inexperienced investors learning from one another by observation. If this algorithm is run 100 times for 400 plays of the Public Investment Game, then the average over all populations and all investors within a population behaves as shown in Figure 9.3. Problem 9.11 gives insight into why this happens.

An interesting phenomenon occurs, if we complicate the model by adding two global features: a minimum investment level called the *law*, and a penalty for failure to invest at least the amount specified by the law called the *fine*. The law could be thought of as the mandated level of taxation and the fine as the penalty for tax evasion. This situation differs from reality in that (i) investors are allowed to overpay tax and (ii) tax cheats are 100%

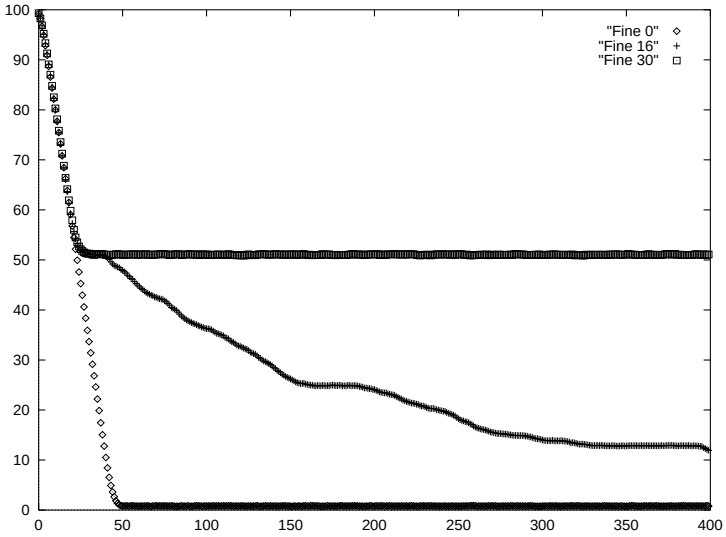


Figure 9.4: Average investment over 50 populations of 60 investors with a law of 50 and fines of 0, 16, and 30

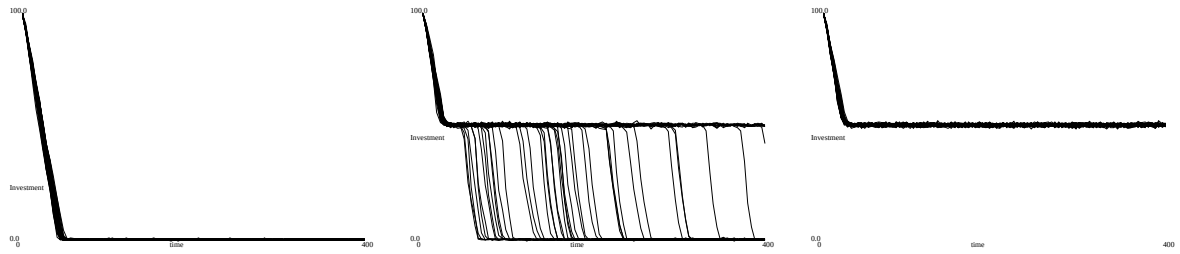
likely to be caught. Modify your software as follows: decrease the fitness of an investor who has an investment amount less than the law by the amount of the fine.

We will investigate what happens with a law of 50 and a number of different fine levels. Figure 9.4 shows the average investment with a law (minimum investment) of 50 and fines of 0, 16, and 30 over 100 populations of 60 investors. Figure 9.5 shows the individual population tracks that led to the average results displayed in Figure 9.4.

In the cases of the extreme fines, the results are simple. As fast as rate of adaption (the +/-5 variation in copied strategies) allows, the population either drops to zero or to the law. With the fine of 16, however, the rapid adaption down to the level of the fine is followed by an almost evaporative series of defections to zero. The average track simply descends slowly. For some of the possible combinations of investments near the law, it is possible to be slightly ahead if you don't meet the law. If too many investors arrive at that state at the same time, then the whole population is paying the fine and it no longer matters.

Logically, the law and fine have two sorts of impact. A small fine is simply a cost of doing business - it is less than the gain from low personal investment. Given that our investors are simply integers, with no sort of moral sense, fines that are less than the profit of bidding low are ignored. On the other hand, high fines will force compliance. A fine of 200, for example, is more than you can possibly make no matter what happens and amounts to instant failure for any investor not obeying the law. This means that at some point above zero but below 200 the fines start causing strong compliance.

Examining Figure 9.5, we see that this logic is not complete. With a fine of zero, the investment level moves smoothly to zero (plus a small amount due to the variation of -5



Fine 0 **Fine 16** **Fine 30**

Figure 9.5: The average investment level for 50 individual populations with the law set to 50 and fines of 0, 16, and 30

to +5 when another investor's strategy is copied). With a fine of 30, a smooth decline to the law of 50 is observed; the curve is similar in character to the decay to zero for no fine. However, when the fine is 16, something odd happens. As the average bid approaches the law, the investors manage to divide between those who pay the fine and those who obey the law. If we examine individual populations, we see that they are either all fine payers or all in compliance with the law after 400 rounds of playing the game and learning from fellow investors. Once a population is made mostly of the exactly lawful (investment levels of the law but not much more) or criminals (investment levels below the law), it tends to stay in that condition. Somehow the dynamics of the simulation permit either outcome, even though the investors start with generous lawful investment levels of 100.

Investigating more deeply, we perform the following experiment. For fines of 0 to 40, in increments of 2 dollars, we find the average investment after 400 plays of the Public Investment Game. The play and learning from fellow investors takes place as before, but now we save only the average investment level in generation 400. This quantity is plotted versus the fine in Figure 9.6. This is the data we wish to model with a least squares fit.

If the assertion that an individual population must either become uniformly law-abiding or uniformly criminal (within a mutation driven small distance of zero investment), then the expected investment level for a given fine is the law times the probability of a given population becoming lawful. In addition, Figure 9.6 indicates this probability is zero for small fines, one for large fines, and follows a sigmoid curve in between. This is enough information to choose a model for the average investment as a function of the fine data. We need a sigmoid curve with a minimum of zero, a maximum of the law (50 in this case), and which can have its maximum slope and horizontal position modified. The shifted hyperbolic tangent curve fits our requirements admirably, see Figure 5.7 and Equation 5.4.

Experiment 9.4 *Modify the software from Experiment 9.2, using the best mutation scheme you found there, to fit to the model*

$$y = 50 \times (tanh(a \cdot (x - b)) + 1)/2.$$

fine	avg	fine	avg
0	0.734667	22	41.7112
2	0.834667	24	46.1248
4	0.977167	26	50.0627
6	1.9965	28	50.5992
8	3.37967	30	50.8558
10	5.683	32	50.772
12	6.51933	34	50.8998
14	10.0245	36	50.8272
16	16.8492	38	50.8257
18	24.7958	40	50.7795
20	32.6318		

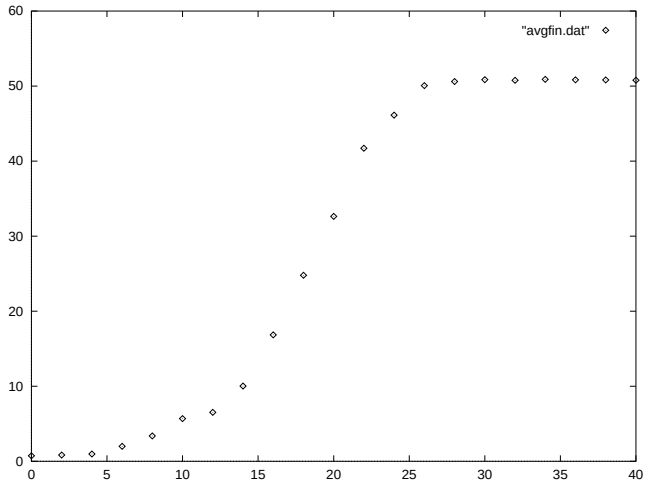


Figure 9.6: Average investment level after 400 plays over 100 populations of 60 investors as a function of the fine amount with a law of 50

Fit this model to the data given in Figure 9.6. Run for 5000 generations with 10 populations and compare the results. Also, plot the data together with a graph of the fitted curve with the lowest error. Is there a consensus about the values of a and b? Find the average over your 10 runs of a and b, and compute the SSE for those averages. How does the resulting SSE compare with the best result from each individual population? You may want to work Problems 9.13-9.16 concurrently with this experiment.

Problems

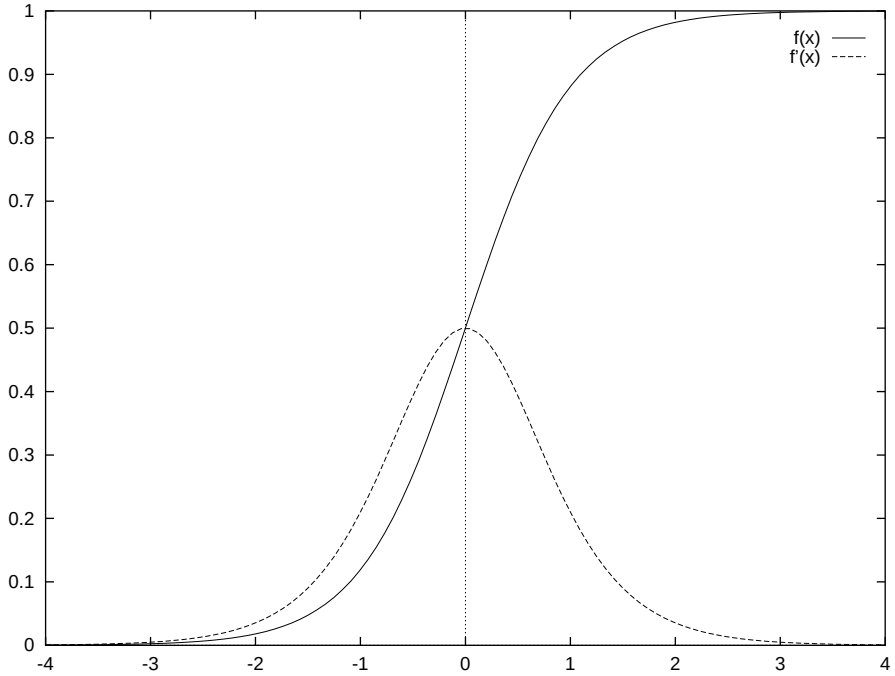
Problem 9.10 *In the discussion before Experiment 9.2, the claim was made that "When doing a least squares fit to data, the sensitivity to changes in one or another dimension in the space of parameters may not be uniform." Suppose we fit the data below to a quadratic curve* $y = ax^2 + bx + c$. *Document the variation in the SSE when varying each of the coefficients a, b, and c by* ± 0.1. *This will require first performing the quadratic fit, hopefully a* very *simple task.*

x	**-4**	**-3**	**-2**	**-1**	**0**	**1**	**2**	**3**	**4**
y	**25**	**16**	**9**	**4**	**1**	**0**	**1**	**4**	**9**

Problem 9.11 *Prove: in the Public Investment Game with $k > 3$ players, if one player's bid is lower than all the others then that player will make the most money in that round.*

Problem 9.12 *Suppose we are having a population of integers learn to play the Public Investment Game as described in this section with 60 players that play in groups of 12. The law is 50, and the fine is 16. If all the players begin with an investment level of 5 or less, show that the population will not become lawful. (A lawful population is one in which a majority have investment levels above the law.)*

For the next several problems you need to read Experiment 9.4 and the material leading up to it. Also examine the graph of $(tanh(x) + 1)/2$ and its derivative, given below.



The function $f(x) = \frac{tanh(x)+1}{2}$ and its derivative.

Problem 9.13 *If $f(x) = 50 \cdot (tanh(a \cdot (x - b)) + 1)/2$ is fit by the method of least squares to a data set like the one given in Figure 9.6, then the numbers a and b contain information about the data being modeled. Given the graph above of the function $\tanh(x)$ and its derivative, what do a and b tell you about the location of the sloped part of the data set and about the slope $\frac{dy}{dx}$?*

Problem 9.14 *Examine the data in Figure 9.6. The data point for a fine of 10 seems to be high, if the curve really is sigmoid. Given the experiment that provided the model, would you expect the value given by the fitted curve or the experimental value of 5.683 to be closer*

to the value we would obtain by averaging over 1,000,000 populations (10,000 times as many as were used to produce the data)?

Problem 9.15 *Least squares fit will work faster, if the gene is initialized close to the correct answer. Give a procedure for initializing the genes of Experiment 9.4 that requires only eyeball estimates of the data, but that will perform better than the random initialization used in the experiment in its current form.*

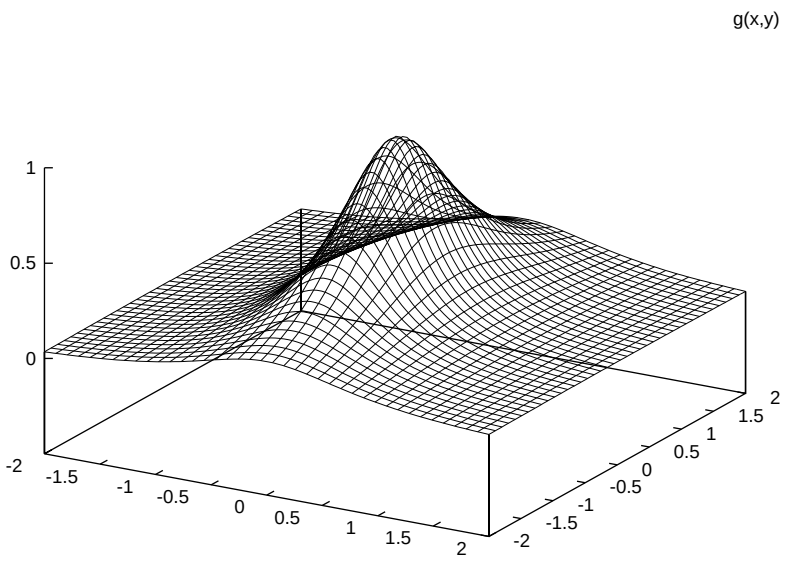


Figure 9.7: A curve $g(x, y) = \frac{1}{1+2x^2+6y^2}$ where the variation away from the mode is not symmetric

Problem 9.16 *The current model of Experiment 9.4 presumes that for small values of the fine, the average investment level is 0. Since half of all investors are just mutated, and for 5 out of 11 of those that means a number between 1 and 5 has been added to their investment, it follows that the average investment must be greater than 0. In addition, some of these mutants survive. Give a generalization of the model used in Experiment 9.4 that models the nonzero nature of the minimum average investment. Also, estimate the value of the minimum investment for a fine of zero.*

Problem 9.17 *Suppose we are using least squares to do a high degree polynomial fit to bivariate data with an evolutionary algorithm. This problem is unimodal and, as Problem*

9.10 shows, subject to different slopes in different directions about the optimum. Would you expect range or domain niche specialization to help more in this situation? An example of a unimodal curve with this sort of directional dependence of variation is shown in Figure 9.7.

Problem 9.18 Essay. *We could estimate the current rate of variation of the SSE with each parameter by simply varying each parameter the same amount and directly computing the change in SSE. We could then set mutation size for that parameter to be some global mutation size divided by that variation. Discuss the pros and cons of implementing such as system with (i) the estimate of variation made only for the best gene in each generation and used on all genes and (ii) the estimate made individually for each gene. Include the implications of Experiment 9.2 in your discussion (if you performed it) and also remember to consider both evolutionary search advantages and computational cost. Use the curve given in Figure 9.7 as an example of the sort of nonuniform variation in different directions for which we are trying to compensate.*

9.3 Symbolic Regression

In this section, we will use genetic programming (GP), a new version of the technique introduced in Chapter 8, to do fits to data. We will be evolving formulas to fit data - i.e. performing *symbolic regression*. Review the basics of genetic programming given in Section 8.1. The fitness function we will be using, as in the preceding section, is minimization of squared error. The difference is that we will not be fitting parameters of a given model, but rather selecting a model (complete with parameters) in the form of a parse tree or parse trees.

The use of parse trees lets us generate random formulas. This means that we are searching an enormously larger space than in the preceding sections. The search is no longer a simple unimodal optimization over the real numbers - it is a mixed search of a discrete space of formulas with incorporated real parameters.

In Chapter 8, the set of operations and terminals was tiny and fixed. In this chapter, we will need to worry about which terminals and operations we need for each experiment. We will also want to be able to enable and disable given terminals and operations. Because of this, you will need an entire class of parse tree routines that let you specify which terminals and operations are available. We will simply assume this capability and leave it to you (or your instructor) to write or download the necessary routines. If only a limited number of GP-experiments from this chapter are performed, then much simpler parse tree code can be used.

For our first exploration, we will use the set of terminals and operations given in Table 9.3. The set of terminals and operations used for the plus-one-recall-store problem in Chapter 8 had the advantage that it required no error trapping; each function could accept without

Node	Symbol	Semantics
		Terminals
real	varies	A real constant in the range -1 to 1
x	x	Holds the value of the independent variable
		Unary Operations
minus	_	Unary minus, negates its argument
square	sqr	Computes the square of its argument
sine	sin	Computes the sine of its argument
cosine	cos	Computes the cosine of its argument
arctangent	atan	Computes the arctangent of its argument
		Binary Operations
plus	+	Adds its arguments
minus	-	Computes the difference of its arguments
times	*	Multiplies its arguments
divide	/	Computes the quotient of its arguments, returning 0 for a divide by zero

Table 9.3: Terminals and Operations for the first set of experiments

error every value that it encountered. The operations in Figure 9.3 share this property, except for division. Division by 0 is not allowed. When doing genetic programming, we modify singularities so as to prevent impossible values. This means the formulas obtained may not be quite standard, and so a bit of care is required.

Since error trapping is very system and language specific, the problem of detecting impossible divisions is left to the writer of a given piece of GP-code. Notice that, for the present, real constants are to be in the range $-1 \leq c \leq 1$. This is less limiting than you might think, because such constants can be used to simulate other constants with a fairly small number of operations. We are now ready to do our experiment.

Experiment 9.5 *Write or obtain software for an evolutionary algorithm to perform genetic programming that operates on a population of 400 parse trees. Initialize the population to have trees with 6 nodes and use the chop operation to contain tree size at no more than 12 nodes. Use single tournament selection with a tournament size of 4 as your model of evolution. When bred, pairs of trees should be crossed over 50% of the time and simply copied 50% of the time. Mutate new trees (crossed or copied) 50% of the time. Let the fitness function be minimization of the SSE with the set of points:*

$$\mathcal{P} = \left\{ \left(\frac{i}{40} - 1, \frac{1}{(i/40 - 1)^2 + 1} \right) \ : i = 0 \ldots 80 \right\}.$$

Run 30 populations for either 500 generations or until the SSE drops below 10^{-6}*. Report the following:*

(i) One best-fitness formula from each population,

(ii) The number of populations that did not need to evolve 500 generations. Create a fraction-of-successes graph where success is defined as finishing early.

At this point, we pay off on a promise made in Chapter 8 and introduce some new mutation operators. These augment, rather than replacing, the subtree mutation introduced in Chapter 8. There are at least three ways to mutate a parse tree without deleting a new subtree. Two of these mutations would have been rather pointless in the PORS environment.

Definition 9.2 *A* **constant mutation** *locates a constant, e.g., a real number, in a parse tree and applies an appropriate mutation to it. A real-valued constant would have a number from some distribution added to it as with real point mutation. An integer or character constant could be replaced with an appropriate new value.*

Definition 9.3 *A* **terminal mutation** *locates a terminal and generates a new terminal. Constant mutations, defined above, are a special case of this. The difference is that this mutation can replace constants with variables and vice versa.*

Definition 9.4 *A* **operation mutation** *locates an operation and changes its identity while preserving its arity (a un*ary *operation is replaced with another un*ary *operation; a bin*ary *operation is replaced with another bin*ary *operation, etc.). To use this mutation requires that multiple operations of each arity be available.*

Experiment 9.6 *Perform Experiment 9.6 again, but this time instead of using subtree mutation alone, use it 25% of the time and also use constant mutation, teminal mutation, and operation mutation 25% of the time each. Report the same data and compare - does the new mix of mutation operations help?*

Experiments 9.5 and 9.6 try to approximate a known function on a fixed grid of sample points. It is interesting how many *different* solutions an evolutionary algorithm can find (see Problem 9.19). Take the problem, for example, of approximating the fake bell curve, $f(x) = \frac{1}{x^2+1}$. What effect does using a fixed set of sample points have on the evolutionary process? It may be the case that using a different random set of sample points for each generation is a good thing. Problem 9.23 addresses this issue. At present, we will treat the question experimentally.

Experiment 9.7 *Modify the software from Experiment 9.6 so that the SSE used for fitness is computed, in any one generation, on the set of points*

$$\mathcal{P}_r = \left\{ \left(\frac{i}{40} - 1 + r, \frac{1}{(i/40 - 1 + r)^2 + 1} \right) \ : \ i = 0 \ldots 80 \right\},$$

where $-0.4 \leq r \leq 0.4$. *Remember to have a single set of points (value of* r*) for each generation, but change* r *for each new generation's fitness evaluations. Run 30 populations. Take the best creature in the final generation of each population and compute the SSE on the set of points* $\mathcal{P}$ *used in Experiment 9.6. Compare the quality of solutions obtained in each experiment and the number of times the algorithm halted early.*

A traditional operation in genetic programming is the if-then-else operation. It requires that we put a boolean interpretation on the data type for the genetic programming. For real numbers one reasonable interpretation is "positive or zero is true, negative is false." We will use this interpretation for the remainder of the chapter. If-then-else, symbolized ITE, is a trinary function with the following definition:

$$ITE(x, y, z) = \begin{cases} y & (x \geq 0) \\ z & (x < 0) \end{cases} \tag{9.4}$$

The ITE operation permits evolution to embed decisions into formulas. If a function has split rules, then ITE permits us to code them in a transparent fashion. The next experiment is intended as a simple demonstration of this.

Experiment 9.8 *First repeat Experiment 9.4, running 30 populations, on the set of points:*

$$\mathcal{S}_\prime = \left\{ \left(\frac{i}{10} - 1, f(i/10 - 1) \right) \ : \ i = 1 \ldots 20 \right\}.$$

where $f(x)$ *is 0 when* x *is negative and 1 otherwise. Now modify your GP-software from Experiment 9.5 to use the ITE operation. With the modified parse trees, rerun Experiment 9.5 on the set of points* $\mathcal{S}_0$. *Compare the SSE of the best creatures from the two experiments. How often did the second set of experiments come up with an essentially correct answer?*

Experiment 9.8, in addition to illustrating the use of the ITE operation in dealing with discontinuous functions, does something else entertaining. In the first half, we are using a continuous family of functions to approximate a collection of sample points drawn from a discontinuous function. This is nearly impossible (and gets harder as we increase the number of sample points). The point is this: trying to do an impossible approximation teaches you something about the approximating technique. Examine creatures with low SSE evolved in the first half of Experiment 9.8. How do they do their approximating?

Problems

Problem 9.19 *Review Experiment 9.5. For each of the formulas below, rate that formula as an accurate approximation to* $f(x) = \frac{1}{x^2+1}$ *in general and on the set of points* $\mathcal{P}$.

(i) $f_1(x) = Cos^2(Arctan(x))$,

(ii) $f_2(x) = Cos^2(x)$,

(iii) $f_3(x) = \frac{1}{x^2+0.99975}$, and

(iv) $f_4(x) = \frac{Cos(Arctan(x))Sin(ArcTan(x))}{x}$.

Problem 9.20 *For each of the functions in Problem 9.19 estimate how hard it is for the evolutionary algorithm described in Experiment 9.5 to find that function, given the fitness function and model of evolution used.*

Problem 9.21 *Several of the functions used in the experiments in this section are listed below. For each, state and support your opinion as to whether deleting the function from those available to the GP-system would decrease or increase time to success.*

(i) $Sin(x)$,

(ii) $Cos(x)$,

(iii) $Sqr(x)$, *and*

(iv) $Arctan(x)$.

Problem 9.22 *Take 3 of the best functions evolved in Experiment 9.5 that did* not *come from populations that finished early. Graph them on the interval* $-2 \leq x \leq 2$. *Recalling that the fitness function used to produce these functions operated in the interval* $-1 \leq x \leq 1$, *comment on their behavior outside the fitness relevant region.*

Problem 9.23 Essay. *Review Experiment 9.7. When we sample the SSE of the function on different sets of points each time, the fitness values of different parse trees jump about. First: prove that they do not do so in a manner that preserves relative fitness. By that we mean: if on one set of sample points parse tree 1 has higher fitness than parse tree 2, then the situation may be reversed on another set of sample points. Second: comment on the effect that this stochastic method of computing fitness has on the population's tendency to get stuck in local optima. Reason in terms of fitness landscapes, if possible.*

Problem 9.24 *Review Experiment 9.8. The set of points $\mathcal{S}_0$ is drawn from the Heaviside function,*

$$H_0(x) = \begin{cases} 1 & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$

Assume we measure the true SSE of a function $f(x)$ on an interval $I = [a, b]$ with an integral:

$$SSE_{[a,b]} = \int_a^b (f(x) - H_0(x))^2 \cdot dx.$$

Discuss the minimum SSE possible if $f(x) = a \cdot tanh(b \cdot x)$.

Problem 9.25 *In Problem 1.16, you were asked to find a minimal GP-representation for a given polynomial. Supposing you can use arbitrary constants, the symbol x, and the operations + and *, give the minimum number of operations needed for a single tree GP-representation of a polynomial*

$$f(x) = a_0 + a_1 x + \cdots a_n x^n$$

when you have no ability to factor and when each $a_i \neq 0$.

Problem 9.26 *Review the definition (Definition 3.4) of fitness landscapes. Describe the domain space for Experiment 9.5. If we are to graph a function, we must have some way of representing the domain as a space where distances between points are apparent. What definition of distance works for this space?*

Problem 9.27 *Describe the extensions to the GP-system described in this section needed to solve differential equations. In addition to describing how to automatically extract the derivative, describe the modifications to the SSE fitness function required to attempt to solve $y' = x + 2y$.*

Problem 9.28 *Examine the following function in LISP-like notation, an evolved formula from a run of Experiment 9.5.*

```
(Cos (Div (Sin (Div (Sin X1) -0.898361)) (Sub 0.138413 0.919129)))
```

The subtree (Sub 0.138413 0.919129) is a verbose way of coding the constant -0.780716. It would be possible to shrink parse trees by detecting subtrees that compute a constant value and replacing them with a real constant terminal. The question is: is this helpful? Consider the effects on evolution of such compaction.

9.4 Automatically Defined Functions

A standard technology used in genetic programming is the *automatically defined function* or *ADF*. In standard programming, an ADF would be called a subroutine. In order to use ADFs, we need to modify our parse tree routines to accommodate them. We will restrict ourself to one-variable ADFs for the present.

Our parse tree routines should be modified to allow or disallow the use of a unary operation called ADF1. Our basic creature will now be made of two parse trees. The first is a "main" parse tree that can use the ADF operation, and the second is an ADF that does not use that operation. In the main tree, when the ADF routine appears, its argument is sent to the ADF parse tree as the terminal x and the ADF parse tree is evaluated. (Both parse trees use the terminal x, but, in the ADF, the value of x is the value of the argument of the call to the ADF, while in the main parse tree, it is the input variable.)

Again: the ADF parse tree is not allowed to use the ADF operation. This would amount to recursion which would open a substantial can of worms. An example of a parse tree and accompanying ADF appear in Figure 9.8.

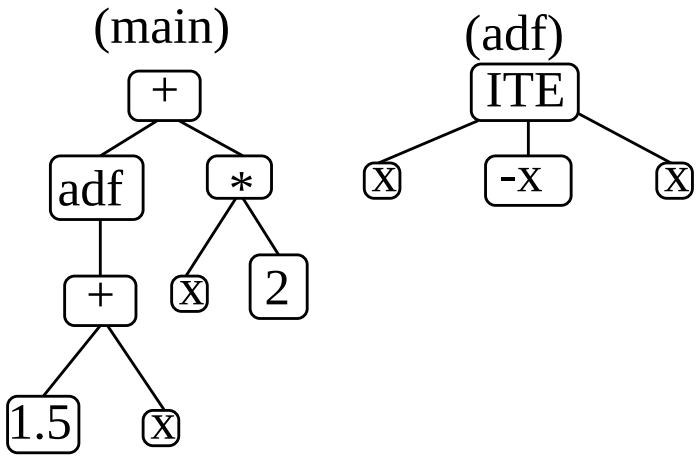


Figure 9.8: A parse tree and ADF

ADFs fulfill the same role in genetic programming that subroutines do in normal programming. They provide code fragments that can be used multiple times during the execution of the program. This implies that some problems will benefit more from ADFs than others. We will test this notion in the next experiment.

Experiment 9.9 *In this experiment, we will approximate two polynomials from a fixed set of points. Use 41 points with x-coordinates equally spaced in the range $-2 \leq x \leq 2$. Modify your parse tree routines to enable the use of ADFs. Also, disable the trigonometric functions used in previous experiments. Use the software of Experiment 9.5.*

Let the main parse trees have a maximum size of 36 and an initial size of 12. When ADFs are used, let the ADFs have an initial size of 4 and a maximum size of 8. Reproduction of parse trees with ADFs copy both the main tree and the ADF.

When crossover is required in a population with ADFs, do one of the following, selecting uniformly at random: cross the main trees in the usual fashion, cross the ADFs in the usual fashion, or take the main tree and ADF from distinct parents. Mutation should pick the main tree or the ADF to mutate with equal probability. A mutation should be used 50% of the time.

Approximate the functions:

(i) $f_1(x) = x^3 - 2x^2 - x + 2$, *and*

(ii) $f_2(x) = x^6 - 4x^5 + 2x^4 + 8x^3 - 7x^2 - 4x + 4$

Treat success as a SSE of 0.001 and run 50 populations for each of the two functions, both with and without ADFs. Report and graph the time to success for all 4 sets of runs. (The first function is much *easier to approximate, so concentrate on the difference the ADF made.)*

In addition, for the best parse trees, state the fraction which used an ADF for each polynomial. Finally, examine the best parse trees from each population and state if factorization of f_1 or f_2 was used. How did it represent the functions?

There is no reason, beyond simplicity, to limit ourselves to a single ADF. It is possible, for example, to design structures that contain a large set of parse trees that refer to one another. Problem 9.29 looks at possible generalizations of the notion of ADF given here. In Chapter 10, we will look at other ways of fragmenting parse trees to form something more like a program and less like a single statement.

Many other such methods exist. You could brainstorm and come up with some good ones yourself. Remember, though, that being a neat idea is not enough. If you are going to modify the basic model of genetic programming, there are a series of questions you should ask yourself.

What does it cost in software? A different way to do genetic programming must be implemented, if it is to be of any use. If you did the modifications to permit ADFs, you have some idea that moving beyond basic parse trees can be an annoying and bug-ridden task. When vetting a new idea, think carefully through what will be needed to build the software that supports it. You can do ADFs with variable numbers of arguments, libraries of useful ADFs, libraries of parse tree fragments, and other similar things. The question is: can you get them running reliably?

What is the cost in instruction cycles? If you modify the representation used by GP-software, you will make it faster (small chance) or slower (large chance). If the change of representation reduces the evolution's time-to-solution (as ADFs should have done in at least

one part of Experiment 9.9), then a modest decrease in the speed of evaluating the fitness of individual creatures is acceptable. Be sure to do back-of-the-envelope estimates of the increased time your modifications will take.

What is the cost in readability? As well as coding your new representation for use on a computer, you will need some method of representing it to people. (There are some cases when a black box that nobody ever looks under the hood of is acceptable, but these are rare.) Ask yourself if you have a method of displaying the results of your new representation that is no worse than the Lisp-like display of parse trees. If you've got one that is substantially better, good; this even pays for some added computational or programming cost. Always keep in mind at design time that a little work now on your I/O and display routines can pay off substantially at data analysis time.

Problems

Problem 9.29 Essay. *It is stated in the discussion of ADFs on page 246 that: "The ADF parse tree is not allowed to use the ADF operation." Explain why not and discuss the difficulties and potentialities of relaxing this restriction.*

Problem 9.30 *Design a set of operations and terminals, including 3 terminals a, b, and c, that permit you to write a parse tree that computes the roots of the quadratic* $f(x) = ax^2 + bx + c$. *The first root should be returned the first time the parse tree is evaluated, the second the second time the parse tree is evaluated. You may either define a value NAN (not a number) or, for advanced students, return the complex roots. Since the parse tree must return two values it must have some sort of memory that lets it know if it has been called before. If the root is unique, return it twice. Question: would an ADF help?*

Problem 9.31 *Find a minimal GP-representation, as a single parse tree, of the polynomial* $f_2(x)$ *from Experiment 9.9 using the operations used in that experiment. Give the answer both with and without ADFs. Does using an ADF help?*

Problem 9.32 *In the spirit of Chapter 8, we give the following question. Suppose you are allowed to build a parse tree and ADF using only one real constant a, the variable x, the operations ADF, +, and *. If the sum of the nodes used in the main parse tree and ADF is at most n, what is the highest degree polynomial you can manage for* $n = 8$, *12, or 16 nodes? Beginning students can simply present their best construction - advanced students should prove they have a correct answer.*

Problem 9.33 *Reread the three bold-faced questions at the end of Section 9.3. Keeping them in mind, consider the following structure. We generate 6 parse trees,* T_1-T_6, *with at most 8 nodes over some otherwise reasonable set of operations and terminals, including only one variable terminal x. Each parse tree can use the parse trees with smaller index as ADFs.*

We use an evolutionary algorithm (leave the details vague) to fit (by minimizing SSE) to a system of two equations $f(x)$ and $g(x)$ where T_5 is used to approximate $f(x)$ and T_6 is used to approximate $g(x)$. Answer all three questions for this system.

9.5 Working in Several Dimensions

Thus far we have done only one-dimensional data fitting with genetic programming. We will now look at issues that arise in multiple dimensions. Our old friend, the fake bell curve, will again be the function of choice for our experiments. The first issue we choose to deal with is the curse of dimensionality.

In Experiment 9.5, we fitted a fake bell curve using a collection of 81 sample points with x-coordinates uniformly spaced on the interval $-1 \leq x \leq 1$. Imagine that we wish to fit the fake bell curve $\mathcal{B}_2(x, y) = \frac{1}{x^2+y^2+1}$ in two dimensions. If we used the same spacing of points, we would need to use $81^2 = 6561$ points. Clearly, as the number of dimensions increases, the number of points needed for a given spacing skyrockets. Before attempting anything too clever, we will check how much harder the problem gets when we hold the number of points constant.

Experiment 9.10 *Rerun Experiment 9.5 with a new termination condition: SSE at most 0.001 or 2000 generations. Also increase the initial and maximum tree size to 16 and 8 respectively. Save the time-to-solution data. Now modify the software from Experiment 9.5 to permit two variable terminals and use the evolutionary algorithm to fit parse trees representing functions $f(x, y)$ to the fake bell curve in two dimensions using points:*

$$\mathcal{P}_{B2} = \left\{ \left(\frac{i}{4} - 1, \frac{j}{4} - 1, \frac{1}{(i/4-1)^2 + (j/4-1)^2 + 1} \right) \ : i = 0 \ldots 8, j = 0 \ldots 8 \right\}.$$

Save time-to-solution and estimate the increase in difficulty of adding a dimension while holding the number of points constant. Does the two-dimensional version of the experiment find any essentially correct solutions?

This curse of dimensionality is not unique to us; engineers and physics have struggled with it for a long time (and found a way to cope). In engineering and physics, it is often necessary to integrate a function of many variables. If we suppose that error estimates require a rectilinear grid with a spacing of 0.05, then 9 dimensions will require, at 21 points per dimension, some 7.94×10^{11} sample points. Present day computers are not up to working with so many points, and future, more efficient computers would have trouble with just slightly higher dimensions.

The solution to integrating multi-dimensional functions turns out to be picking points at random for integration, termed *Monte Carlo* integration. An advanced text on numerical

analysis will go over the techniques of Monte Carlo integration, including the critical error estimates used to decide how many random points are enough. We will simply use the idea of sampling points to learn to approximate functions in many dimensions.

Experiment 9.11 *Modify the software from the second half of Experiment 9.10 to operate on n randomly selected points* $(x, y, \mathcal{B}_2(x, y))$ *with* $-1 \leq x, y, \leq 1$. *An array of points should be generated in each generation and used to evaluate the fitness of all the parse trees in that generation. Use this fitness for selection, but also compute the fitness measure used in Experiment 9.10 and use it to compute time to success. Compare the time to success of the previous experiment (measured in generations) with that of this experiment using data taken for n =20, 81, and 200 points. Do random points work any better than regularly spaced ones? How does the number of points used affect the time to success? Remember to compensate for the effect of computing more sample points.*

The fake bell curve, in any number of dimensions, could be defined as $\mathcal{B}_b(p) = \frac{1}{d(p)^2+1}$, where p is a point in $\mathbb{R}^n$ and $d(p)$ is the distance from the point to the origin. Given that this distance function enormously simplifies computation of the fake bell curve, it is possible that the fake bell curve would benefit from the use of ADFs.

Experiment 9.12 *Modify the parse tree routines from Experiment 9.9 and/or 9.10 to permit two-variable ADFs and then re-perform the 3 sets of runs from Experiment 9.11 using ADFs. Use the variation operators given in Experiment 9.9. Permit the ADF trees an initial 6 and a maximum of 10 nodes. Compare with the results of Experiment 9.11: does using ADFs help?*

One problem with genetic programming, as performed so far in this chapter, is the mutation of real constant terminals. Real constant terminals are called *ephemeral constants*. There is no mutation that makes small changes to the value of such real constants - we either leave them alone or completely change them. There are a number of ways to deal with this. The simplest is to add a mutation operator that locates a constant within a tree and performs a real mutation, as in Chapter 3.

The fake bell curve uses only the constant value 1, so we will switch to a different function with more constants. Recall that the area of an ellipse with major axis $2a$ and minor axis $2b$ is πab.

Experiment 9.13 *Implement a mutation operator that can locate a real constant (equal chance among all constants in a parse tree) and do a real mutation with mutation size* ϵ *to it. Add this new mutation operator to the software used in Experiment 9.11.*

In this experiment we will evolve parse trees to find the area of an ellipse of diameters $2a$ *and* $2b$. *Use a population of 400 parse trees with an initial size of 6 and a maximum size*

of 12 nodes. Let your model of evolution be single tournament selection with tournament size 4. Use crossover in 80% of all breeding events and (normal subtree) mutation in 50% of breeding events. Minimize the SSE of your parse trees with the formula $Area = \pi ab$ *for 80 randomly chosen values of* a *and* b *in the range* $0.1 \leq a, b \leq 4$*, choosing new random values in each generation. Test for success by looking for an SSE of 0.001 or less on the ellipses with* a, b *valued at any multiple of 0.5 in the range* $0.5 \leq a, b \leq 4$*.*

Do 30 runs, saving time to success data and giving up after 2000 generations. Now, run the 30 populations again with each new tree mutated 50% of the time with the new mutation operator (in addition to and with probability independent of the subtree mutation) with $\epsilon = 0.2$*. Did the new mutation operator help?*

Now, let's do an experiment to explore the problems caused by increasing the number of dimensions.

Experiment 9.14 *Modify the software used in Experiment 9.10 to use 200 randomly selected sample points for fitness evaluation. Perform 30 runs each, attempting to approximate the fake bell curve in 1,2,3, and 4 dimensions. Compare the results obtained in each case. Estimate the relative difficulty of the problems.*

As is always the case, there are a large number of things we could do to further explore these issues. Comparing random sample points with a fixed grid that moves (as in Experiment 9.7), for example. This chapter has dealt only with artificial, manufactured data. You are encouraged to locate real data of interest and use the techniques you've learned to try to fit to the data. In later chapters, we will explore using genetic programming to model discrete data and heterogeneous data. In real life applications, mixed real values and discrete data are the rule rather than the exception.

Problems

Problem 9.34 *Experiment 9.10 uses the 81 points of Experiment 9.5 to form a regular two-dimensional grid with 81 points. Suppose that we wish to do a similar comparison across 1, 2, and 3 dimensions with the number of points the same for each dimension. What numbers of points less than 1000 permit this? Hint: this is almost trivial.*

Problem 9.35 *Examine the operations given in Table 9.3. When we move to multiple dimensions, there is more potential utility to having operations that take many arguments. With some operations, it is obvious what modifications are needed to allow more arguments. For example, it is obvious what an n-ary addition operator does - it adds up all of its arguments. For each of the operations in Table 9.3, give your best shot at defining how that operation would act on an arbitrary list of arguments. Also, rate your construction as completely natural, as reasonable but different from the original, or as contrived.*

Problem 9.36 Essay. *Read the three bold-faced questions at the end of Section 9.3. Keeping them in mind, consider the following idea. Modify all operations so that they operate on a list of arguments of arbitrary size. Answer all three of the questions with respect to this system.*

Problem 9.37 Essay. *Read the three bold-faced questions at the end of Section 9.3. Keeping them in mind, consider the following idea. Our experience in approximating the fake bell curve has shown that one weakness of genetic programming is the poor job it does discovering constants. Consider a system with exactly 10 available constant terminals (instead of the generic real number constant terminal). Each parse tree is augmented with an array of 10 real numbers. These array genes undergo two-point crossover and single point real mutation with an appropriate mutation size during breeding. Answer all three of the questions with respect to this system.*

Problem 9.38 *Review Experiment 9.13. We observed in Experiment 9.5 that it was easier for the evolutionary algorithm to find the approximation $f(x) = Cos^2(Tan^{-1}(x))$ for the fake bell curve in one dimension than for it to find the correct formula. For each of the following features, comment in a sentence or two on the degree to which it might help the software from Experiment 9.5 find the formula $\frac{1}{x^2+1}$.*

(i) A terminal **one** that returns the value 1.

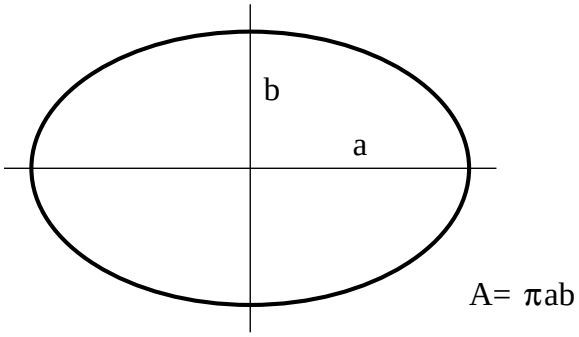
(ii) The mutation operator introduced in Experiment 9.13.

(iii) A reciprocal operator.

Problem 9.39 Review. *Using calculus, verify the formula for the area of an ellipse given on page 250. Recall that an ellipse is defined by the relation*

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1.$$

Refer to the graph below for inspiration.



Problem 9.40 *Suppose that in the course of more than 100 populations run working out Experiments 9.5 and 9.7, only one formula appeared as the best-of-run that approximated the functional form of the fake bell curve. This was*

$$f(x) = \frac{0.993126}{0.991796 + x^2}.$$

A large number of variations on the solution

$$g(x) = Cos^2(Arctan(x))$$

appeared as best-of-runs. Why? Consider the shape of the cosine function and the number of ephemeral constants required.

9.6 Introns and Bloat

Genetic programming is the first instance of a variable-sized representation we have studied. In Chapter 8, we did genetic programming, but in a fashion that minimized the impact of size variation. The PORS trees were used to explore the behavior of a particular size of parse tree. This limits their ability to use their variable size for much of anything.

The *depth* of a parse tree is the maximum number of links from the root to any terminal. The main function shown in Figure 9.8 has depth 3, while the ADF has depth 1. In standard genetic programming, as opposed to genetic programming described thus far in this text, the depth of trees is controlled, but not their size. The decision to control size in the genetic programming in this text was motivated by research on the subject of bloat. When working with a variable-sized genome, *bloat* is defined to be the use of space to gain advantages not directly related to fitness. In this section, we will do a few experiments to explore the phenomenon of bloat.

Bloat came as a surprise to the first researchers to encounter it. They observed that the average size of parse trees grows to near the maximum possible, whatever restrictions on size are present. In Problem 8.3, we saw that parse trees can grow without bound as a result of subtree crossover. The question remains however as to *why* they grow in this manner. Since there are more large trees than small trees, bloat could be the result of simple random drift. However, providing secondary fitness pressure for "parsimony" (small tree size) does not have much effect. There seems to be a positive selection pressure for large size.

One explanation of parsimony-resistant bloat makes an analogy to natural genetics. Genes, in living creatures, often contain DNA that does not code for anything and which is spliced out at the RNA stage, during translation of DNA into protein. These chunks of DNA are called *introns*. Introns permit DNA to undergo crossover that does not disrupt the protein-coding portions. Since subtree crossover of the sort used in genetic programming is highly disruptive, there is a selection pressure to develop substructures in the parse trees

that allow the parse tree to resist crossover-based disruption. The question then becomes "what do these substructures look like?" It is intuitive that they will depend on numerical identities, but their exact form is not clear. Let us experiment.

Experiment 9.15 *Rewrite the software used in Experiment 9.5 so that it saves the average size of parse trees in the final population. Rerun the experiment in its original form and with the upper bound on the size of parse trees changed from 12 to 60. Make the following analysis.*

(i) Plot the generation of solution versus the average size of tree in the population. Is there a correlation in either of the two sets of runs?

(ii) Which set of runs had the better average number of generations to solution?

(iii) Which set of runs had the better fitness among those runs that ran out of time?

(iv) Examine the best-of-run formulas. Are there any structures that might be introns? Are there any structures that look like they were created by repeated subtree crossover?

One way to see if bloat is useful is to make it easier. A *designated intron* is a function that returns its argument or, if it is not a unary function, one of its arguments. If bloat is useful for avoiding crossover-based disruption, making it easy by including a designated intron should enhance bloat. It is an open question as to whether it will enhance the performance of a GP-system.

Experiment 9.16 *Rewrite the software used in Experiment 9.5 so that it saves the average size of parse trees in the final population. Also, modify the parse tree software to include a unary operation,* say*, that simply returns its argument. Rerun the experiment twice with the maximum size of parse trees set to 12 and to 60. Compare with the results from Experiment 9.15. Is the say operation used oftener than it would be by random chance? Do the runs with the say operation exhibit superior performance?*

Problems

Problem 9.41 *The following formulas are results that occurred in our version of Experiment 9.5. Verify each equals* $f(x) = \frac{1}{x^2+1}$*, and then spot the potential bloat in each. Some of the functions have more than one instance of something that could be bloat.*

(i) $q(x) = Cos(-Arctan(x))Cos(Arctan(x))$,

(ii) $g(x) = Cos(Arctan(x))Cos(-(Arctan(x) - (x - x)))$,

(iii) $h(x) = Cos^2(Arctan(x)(Cos(x - x))^2)$,

(iv) $a(x) = Cos^2(x\frac{Arctan(x)}{x})$,

(v) $b(x) = \frac{\frac{x}{x}}{\left(\frac{\left(\frac{x}{Cos(Arctan(x))}\right)}{Cos(Arctan(x))}\right)}$, and

(vi) $c(x) = Cos^2((x - x) + Arctan(x))$.

Problem 9.42 *Give a method of using introns for string genes of the sort used in Sunburn or VIP, from Chapter 4. Explain how the introns might help. Give, also, the design of an experiment to tell if the introns help.*

Problem 9.43 *A* recessive gene *in biology is a gene that has no effect on the phenotype of the creature carrying it. If a subtree could be anything without changing the fitness of the tree above it, then it could be termed recessive. Construct a tree that has a recessive subtree and that is an exact answer for Experiment 9.5.*

Problem 9.44 *Reread problem 9.28. Are such constant subtrees a possible source of bloat? What other purpose could they serve?*