NCAR/TN-450+IA
NCAR TECHNICAL NOTE

March 2002

AN INTRODUCTION TO GENETIC ALGORITHMS FOR NUMERICAL OPTIMIZATION

Paul Charbonneau

HIGH ALTITUDE OBSERVATORY

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
BOULDER, COLORADO

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

In 1998 I was invited to present a lecture on genetic algorithms at a Mini-Workshop on Numerical Methods in Astrophysics, held June 3–5 at the Institute for Theoretical Astrophysics, in Oslo, Norway. I subsequently prepared a written version of the lecture in the form of a tutorial introduction to genetic algorithms for numerical optimization. However, for reasons beyond the organizers' control, the planned Proceedings of the Workshop were never published. Because the written version, available through the `PIKAIA` Web Page since september 1998, continues to prove popular with users of the `PIKAIA` software, I decided to "publish" the paper in the form of the present NCAR Technical Note.

The paper is organized as follows. Section 1 establishes the distinction between local and global optimization and the meaning of performance measures in the context of global optimization. Section 2 introduces the general idea of a genetic algorithm, as inspired from the biological process of evolution by means of natural selection. Section 3 provides a detailed comparison of the performance of three genetic algorithm-based optimization schemes against iterated hill climbing using the simplex method. Section 4 describes in full detail the use of a genetic algorithm to solve a real data modeling problem, namely the determination of orbital elements of a binary star system from observed radial velocities. The paper closes in section 5 with reflections on matters of a somewhat more philosophical nature, and includes a list of suggested further readings.

I ended up making very few modifications to the text originally prepared in 1998, even though if I were to rewrite it now some things undoubtedly would turn out different. The suite of test functions I now use to test modifications to `PIKAIA` has evolved significantly from that presented in §2 herein. Version 1.2 of `PIKAIA`, publicly released in April 2002, would compare even more favorably to the iterated simplex method against which `PIKAIA 1.0` is pitted in §3 herein. I updated and expanded the list of further reading (§5.5) to better reflect current topic and trends in the genetic algorithm literature. In addition to some minor rewording here and there throughout the text, I also restored a Figure to §1, and a final subsection to §2, both originally eliminated to fit within the 50-page limit of the above-mentioned ill-fated Workshop Proceedings.

Back in 1998, I chose to give this paper the flavor of a tutorial. Each section ends with a summary of important points to remember from that section. You are of course encouraged to remember more than whatever is listed there. You will also find at the end of each section a series of Exercises. Some are easy, others less so, and some require programming on your part. These are designed to be done using `PIKAIA`, a public domain self-contained genetic algorithm-based optimization subroutine. The source code for `PIKAIA` —as well as answers to most exercises— are available on the tutorial Web Page, from which you can also access the `PIKAIA` Web Page:

`http://www.hao.ucar.edu/public/research/si/pikaia/tutorial.html`

The Tutorial Page also includes various animations for some of the solutions discussed in the text. The `PIKAIA` Web Page contains links to the HAO ftp archive, from which you can obtain, in addition to the source code for `PIKAIA`, a User's Guide, as well as source codes for the various examples discussed therein. The idea behind all this is that by the time you are done reading through this paper *and* doing the Exercises, you should be in good shape to solve global numerical optimization problems you might encounter in your own research.

The writing of this preface offers a fine opportunity to thank my friends and colleagues Viggo Hansteen and Mats Carlsson for their invitation and financial support to attend their 1998 Mini-Workshop on Numerical Methods in Astrophysics, as well as for their kind hospitality during my extended stay in Norway. The ρ CrB data and some source codes for the orbital element fitting problem of §4 were provided by Tim Brown, who was also generous with his time in explaining to me some of the subtleties of orbital element determinations. Thorough readings of the 1998 draft of this paper by Sandy and Gene Arnn, Tim Brown, Sarah Gibson, Barry Knapp and Hardi Peter are also gratefully acknowledged.

Throughout my twelve years working at NCAR's High Altitude Observatory, it has been my privilege to interact with a large number of bright and enthusiastic students and postdocs. My forays into genetic algorithms have particularly benefited from such collaborators. Since 1995, I have had to keep up in turn with Ted Kennelly, Sarah Gibson, Hardi Peter, Scott McIntosh, and Travis Metcalfe. I thank them all for keeping me on my toes all this time.

Paul Charbonneau

March 2002, Boulder

1. INTRODUCTION: OPTIMIZATION

1.1 Optimization and hill climbing

Optimization is something that most readers of this tutorial will have first faced a long time ago in their first calculus class; one is given an analytic function $f(x)$, and presented with the task of finding the value of x at which the function reaches its maximum value. The procedure taught toward this end is (1) differentiate the function with respect to x; (2) set the resulting expression to zero; (3) solve for x, call the result $x_{\max}$, and there you have it[1]. Even though most of us would no longer think twice about it, this is actually a pretty neat trick!

For the reader trained in physics the limitation of this analytical method was encountered perhaps first in optics, when studying the diffraction pattern of a single vertical slit (e.g., Jenkins & White 1976, chap. 15). You might recall that the intensity of the diffraction pattern varies as $(\sin x/x)^2$, where x is directly proportional to the distance along the direction perpendicular to the slit on the screen on which the diffraction pattern is projected. The location of the intensity minima are readily found to be $x_{\min} = \pm n\pi$, with $n = 1, 2, ...$ ($n = 0$ is trickier). However, calculating the locations of the intensity maxima by the analytical procedure described above leads to a nasty nonlinear transcendental equation which cannot be solved algebraically for x. One has to turn to iterative or graphical means (in the course of which the trickier $n = 0$ case of the minima is also resolved). This difficulty with the diffraction problem is symptomatic of the fact that it is usually harder (very often *much* harder) to find the zeros of functions than their extrema, the more so the higher the dimensionality of the said functions (see Press et al. 1992, §9.6, for a concise yet lucid discussion of this matter). The inescapable conclusion is that once one moves beyond high school calculus min/max problems, optimization is best carried out numerically.

Upon opening a typical introductory textbook on numerical analysis, one is almost guaranteed to find therein a few optimization methods described in some

[1] In fact, you also have to differentiate the result of step (2) once again, and verify that the resulting expression is negative when evaluated at $x_{\max}$; but this subtlety might have been elaborated upon only in the next lecture...

detail. In nearly all cases, those methods will fall under the broad category of *hill climbing schemes*. The operation of a generic hill climbing scheme is illustrated on Figure 1, in the context of maximizing a function of two variables, i.e., finding the maximum "altitude" in a 2-D "landscape". Hill climbing begins by choosing a starting location in parameter space (panels [A]–[B]). One then determines the local steepest uphill direction, moves a certain distance in that direction (panel [C]), re-evaluates the local uphill direction, and so on until a location in parameter space is arrived at where all surrounding directions are downhill. This marks the successful completion of the maximization task (panel [D]). Most textbook optimization methods basically operate in this way, and simply differ in how they go about determining the steepest uphill direction, choosing how a big a step is to be taken in that direction, and whether or not in doing so use is made of gradient information accumulated in the course of previous steps.

Hill climbing methods work great if faced with unimodal landscapes such as the one towards which the rabid paratrooper of Fig. 1(A) is about to deposit his lower backside. Unfortunately, life is not always that simple. Consider instead the 2-D landscape shown on Figure 2; the maximum is the narrow central spike indicated by the arrow, and is surrounded by concentric rings of secondary maxima. The only way that hill climbing can find the true maximum in this case is if our paratrooper happens to land somewhere on the slopes of the central maximum; hill climbing from any other landing site will lead to one of the rings. The central peak covers a fractional surface area of about 1% of the full parameter space ($0 \leq x, y \leq 1$). Unlike on the landscape of Fig. 1(A), here the starting point is critical if hill climbing is to work. Hill climbing is a *local* optimization strategy. Figure 2 offers a *global* optimization problem.

Of course, if the specific optimization problem you are working on happens to be such that you can always come up with a good enough starting guess, then all you need is local hill-climbing, and you can proceed merrily ever after. But what if you are in the situation most people find themselves in when dealing with a hard global optimization problem, namely *not* being in a position to pull a good starting guess out of your hat?

I know what you're thinking. If the central peak covers about 1% of parameter space, it means that you have about one chance in a hundred for a random drop to land close enough for hill climbing to work. So the question you have to ask yourself is: do I feel lucky?[2] Your answer to this question is embodied in the First

[2] Well, do you, punk?

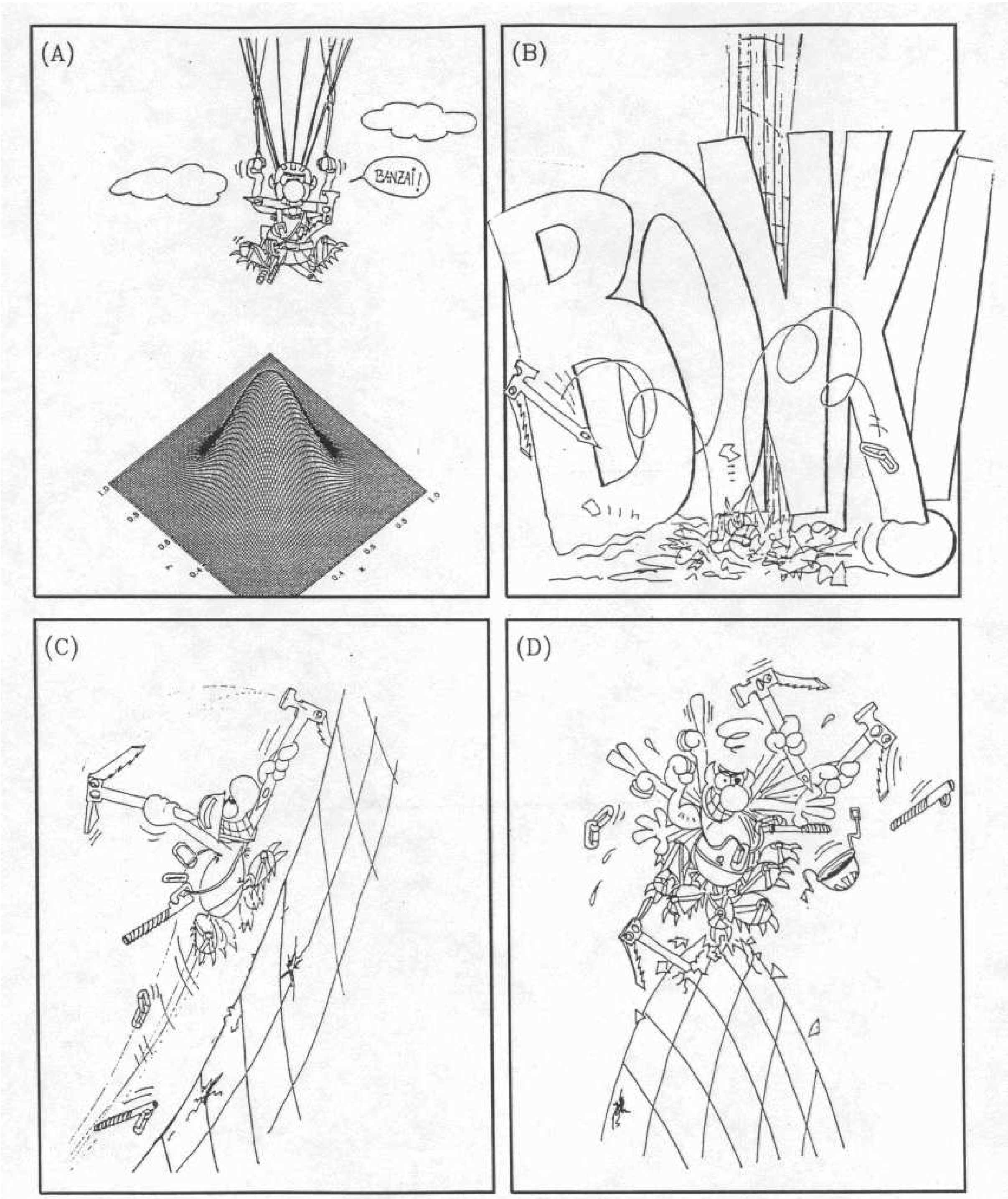

Figure 1: Operation of a generic hill climbing method (allegory). From a randomly chosen starting point (panel [A]), the direction of maximum slope is followed (panel [C]) until one reaches a point where all surrounding directions are downhill (panel [D]). Landing (panel [B]) is not problematic from the computational point of view.

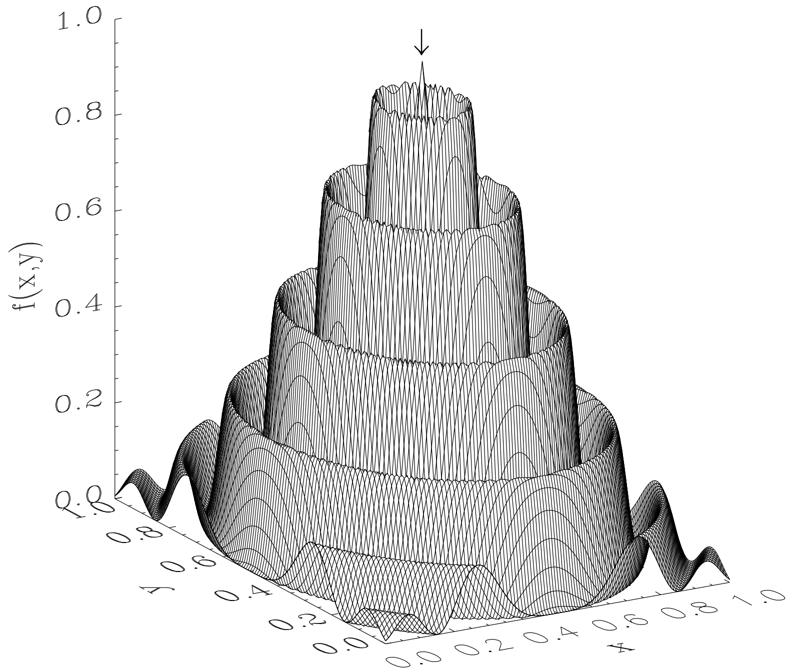

Figure 2: Two dimensional surface $f(x, y)$, with $x, y \in [0, 1]$, defining a hard maximization problem. The global maximum is $f(x, y) = 1$ at $(x, y) = (0.5, 0.5)$, and is indicated by the arrow.

Rule of Global Optimization, also known as

THE DIRTY HARRY RULE:

"You should never feel lucky"

Faced with the landscape of Figure 2 the most straightforward solution lies with a technique called *iterated hill climbing*. This is a fancy name for something very simple, as illustrated on Figure 3. You just run your favorite local hill climbing method repeatedly, each time from a different randomly chosen starting point. While doing so you keep track of the various maxima so located, and once you are satisfied that all maxima have been found you pick the tallest one and you are done with your global optimization problem. As you might imagine, deciding when to stop is the crux of this otherwise straightforward procedure.

Figure 3: An iterated hill-climbing scheme. After landing, each trial proceeds as on Fig. 1.

With a fractional coverage of 1% for the central peak of Figure 2, you might expect to have to run, on average, something of the order of 10^2 iterated hill climbing trials before finding the central peak. As one is faced with optimization problems of increasing parameter space dimensionality, and/or situations where the global maximum spans only a tiny fraction of parameter space, iterated hill climbing can add up to a lot of work. This leads us naturally to the Second Rule of Global Optimization, also known as

> THE NO FREE LUNCH RULE:
>
> "If you really want the global optimum, you will have to work for it"

These considerations also lead us to distinguish between three distinct aspects of performance, when dealing with a global optimization problem[3]:

(1) **Absolute performance:** How numerically accurate is the solution returned by my adopted method?

(2) **Global performance:** How certain can I be that the solution returned by my method is the true global maximum in parameter space?

(3) **Relative performance:** How much computational work is required by my method to return a solution?

Most fancy optimization methods you might read about in textbooks are designed to do as well as possible on (1) and (3) simultaneously. Such methods will do well on (2) only if provided with a suitable starting guess. If such a guess is consistently available for the problems you are working on, you need not read any further. But rest assured that Dirty Harry will catch up with you one of these days.

1.2 The simplex method

The distinction between local and global optimization, as well as the related performance issues, are perhaps best appreciated by looking in some detail at the behavior of a local hill climbing method on a global optimization problem. Toward this end we retain the 2-D landscape of Figure 2 as a test bed, and attempt to maximize it using the *Simplex Method*.

---

[3] A different terminology may well be used in optimization textbooks, but you can be assured that they do discuss something equivalent.

The Simplex Method of Nelder & Mead (1965) is actually a very robust hill climbing scheme. A brief yet clear introduction to the method can be found in Press *et al.* (1992, §10.4). A simplex is a geometrical figure with $n+1$ vertices, that lives in a parameter space of dimension n. In 2-D space a simplex is a triangle, in 3-D space a tetrahedron, and so on. Given the function value (here the "altitude" $f(x, y)$) at each of the simplex's vertices (here an (x, y) point), the worst vertex is displaced by having the simplex undergo one of three possible types of "moves", namely contraction, expansion, or reflection (see Fig. 10.4.1 in Press *et al.*). The move is executed in a manner such that the function value of the displaced vertex is increased by the move (in the context of a maximization problem). The simplex undergoes successive such moves until no move can be found that leads to further improvement beyond some preset tolerance. Watching the simplex contract and expand and squirt around the landscape of Fig. 2 is good visual fun[4], and justifies well the name given by Press *et al.* to their simplex subroutine: `amoeba`. This is the implementation used here.

By the standards of local optimization methods, the simplex passes for a "slow" method. The absolute accuracy of the solution increases approximately linearly with the number of simplex moves. However, the simplex can pull itself out of situations that would defeat or seriously impede faster, "smarter" gradient-based local methods; it can efficiently crawl up long flat valleys, and squeeze through saddle points. In this sense, it can be said to exhibit pseudo-global capabilities.

Evidently the simplex method requires that one provide initial coordinates (x, y) for the simplex's three vertices. Despite the simplex method's pseudo-global abilities, on a multimodal, global problem the choice of initial location for the simplex often determines whether the global maximum is ultimately found. Figure 4 shows a series of convergence curves for the test problem of Figure 2. Each curve corresponds to a different, random initial simplex configuration. When the simplex finds the central peak, it does so rather quickly, requiring about 25 moves for 10^{-5} accuracy. The problem, of course, is that the simplex often does not converge to the central peak. Repeated trials reveal that the method achieves global convergence for only 2% or so of trials.

1.3 Iterated Simplex

The prospects of the simplex method for global performance are greatly enhanced if one of the starting vertices lies high enough on the slopes of central peak. This suggests that iterated hill climbing using the simplex method (hereafter iterated simplex) should achieve global performance within a few hundred

[4] An animation of the simplex at work on the 2-D landscape of Figure 2 can be viewed on the tutorial Web Page. Check it out!

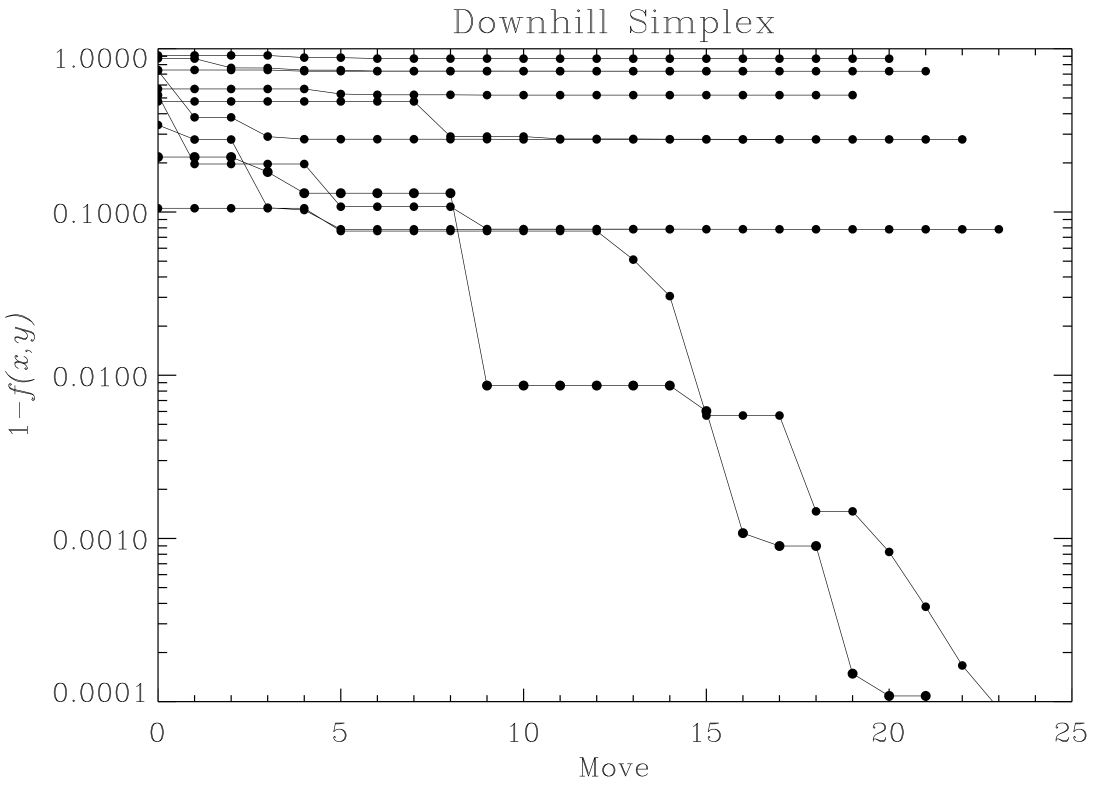

Figure 4: Absolute performance of the simplex method on the test problem of Figure 2. Each curve corresponds to a different starting simplex. Failure of the simplex to locate the central peak leads to the convergence curves leveling off at relatively high values of $1-f(x,y)$. With 2 converged runs out of 10 trials, this plot is *not* representative of the simplex method's global performance on this problem, which is in fact significantly poorer, namely about 2%.

iterations. And indeed it does; repeatedly running the simplex (500 times) on the test problem of Figure 2 leads to the central peak being located in 99.5% of trials[5]. The price to pay of course, is in the number of function evaluations required to achieve this level of global performance: nearly 10^4 function evaluations per iterated simplex run, on average[6]. Welcome back to the No Free Lunch Rule...

[5] It is recommended practice when using the simplex in single-run mode to carry out a *random restart* once the simplex has converged; this entails reinitializing randomly all but one of the converged simplex's vertices, and letting the simplex reconverge again. What is described here as iterated simplex consists in reinitializing *all* vertices randomly, so as to make each successive trial fully independent from all others.

[6] A single simplex move may entail more than one function evaluation. For

1.4 A set of test problems

One should rightfully suspect that the simplex method's performance on the test problem of Figure 2 might not be representative of its performance on other problems. This very legitimate concern will evidently carry over to the various genetic algorithm-based optimization schemes discussed further below. It will therefore prove useful to have available not just one, but a set of test problems. The four test problems described below are all *very* hard global optimization problems, on which most conventional local optimization algorithms would fail miserably. Also keep in mind that it is *always* possible to design a test problem that will defeat *any* global optimization method[7].

1.4.1 P1: maximizing a function of two variables [2 parameters]

Our first test problem (hereafter labeled "P1") is our now familiar 2-D landscape of Figure 2. Mathematically, it is defined as

$$f(x,y) = \cos^2(n\pi r)\exp(-r^2/\sigma^2) \ , \tag{1a}$$

$$r^2 = (x-0.5)^2 + (y-0.5)^2, \qquad x,y \in [0,1] \ , \tag{1b}$$

where $n = 9$ and $\sigma^2 = 0.15$ are constants. The global maximum is located at $(x,y) = (0.5, 0.5)$, with $f(x,y) = 1.0$. This global maximum is surrounded by concentric rings of secondary maxima, centered on the global maximum at radial distances

$$r_{\max} = \{0.110192, 0.220385, 0.330582, 0.440782, 0.550986\} \ . \tag{2}$$

Between these are located another series of concentric rings corresponding to minima:

$$r_{\min} = \frac{m-1/2}{n}, \qquad m = 1, ..., 6 \tag{3}$$

The error ε associated with a given solution (x,y) can be defined as

$$\varepsilon = 1 - f(x,y) \ . \tag{4}$$

---

example if the trial move does not lead to an increase in f, the move might be repeated with a halved or doubled displacement length (or a different type of move might be attempted, depending on implementation). On the maximization problem of Figure 2, one simplex move requires 1.8 function evaluations, on average.

[7] The high-n, high-D version of the fractal function discussed in §3.5 of Bäck (1996) is a pretty good candidate for the ultimate killer test problem.

Note that the "peak" corresponding to the global maximum covers a surface area $\pi/(4n^2)$ in parameter space. If a hill climbing scheme were used, the probability of a randomly chosen starting point landing close enough to this peak for the method to locate the true global maximum is only 1% for $n = 9$.

1.4.2 P2: maximizing a function of two variables again [2 parameters]

Test function P2, shown on Figure 5, is again a 2-D landscape to be maximized. It is defined by

$$f(x, y) = 0.8 \exp(-r_1^2/(0.3)^2) + 0.879008 \exp(-r_2^2/(0.03)^2) \ , \tag{5a}$$

$$r_1^2 = (x - 0.5)^2 + (y - 0.5)^2 \ , \tag{5b}$$

$$r_2^2 = (x - 0.6)^2 + (y - 0.1)^2 \ . \tag{5c}$$

The maximum $f(x, y) = 1$ is at $(x, y) = (0.6, 0.1)$, and corresponds to the peak of the second, narrower Gaussian. P2 is about as hard a global optimization problem as P1 (the simplex succeeds 141 times out of 10^4 trials), but for a different reason. There are now only two local maxima, with the global maximum again covering about 1% of parameter space. Unlike P1, where moving toward successively higher secondary extrema actually brings one closer to the true maximum, with P2 moving to the secondary maximum pulls solutions *away* from the global maximum. Problems exhibiting this characteristics are sometimes called "deceptive" in the optimization literature.

1.4.3 P3: maximizing a function of four variables [4 parameters]

Test problem P3 is a direct generalization of P1 to four independent variables (w, x, y, z):

$$f(w, x, y, z) = \cos^2(n\pi r) \exp(-r^2/\sigma^2) \ , \tag{6a}$$

$$r^2 = (w - 0.5)^2 + (x - 0.5)^2 + (y - 0.5)^2 + (z - 0.5)^2 \ , \qquad w, x, y, z \in [0, 1] \ , \tag{6b}$$

again with $n = 9$ and $\sigma^2 = 0.15$. Comparing performance on P1 and P3 will provide a measure of *scalability* of the method under consideration, namely how performance degrades as parameter space dimensionality is increased, everything else being equal. P3 is a *very hard* global optimization problem; the simplex method manages to find the global maximum only 6 times out of 10^5 trials.

1.4.4 P4: Minimizing a least squares residual [6 parameters]

Our fourth and final test problem is defined as a "real" nonlinear least squares fitting problem. Consider a function of one variable (x) defined as the sum of two Gaussians:

$$y(x) = \sum_{j=1}^{2} A_j \exp\left(-\frac{(x - x_j)^2}{\sigma_j^2}\right) \ . \tag{7}$$

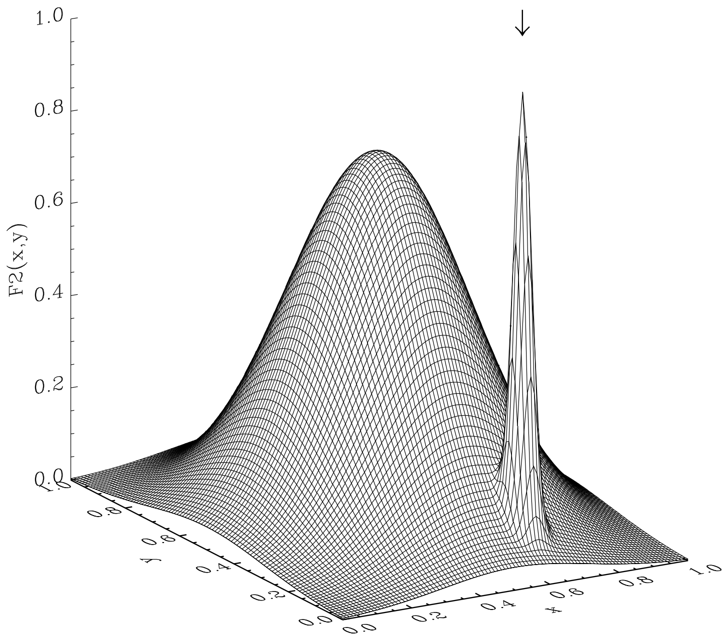

Figure 5: Test problem P2. The problem consist in maximizing a function of two variables, defined by two Gaussians (see eqs. [5]). The global maximum is $f(x, y) = 1$ at $(x, y) = (0.6, 0.1)$, and is indicated by the arrow.

Define now a "dataset" by evaluating this function for a set of K equidistant values of x_k in the interval $[0, 1]$, i.e., $y^*_k \equiv y(x_k)$, $x_{k+1} - x_k = \Delta x$, for some set values of $A_1, x_1,$ etc. Given that dataset and the functional form used to generate it (i.e., eq. [7]), the optimization problem is then to recover the parameter values for $A_1, x_1,$ etc originally used to produce the dataset. This is done by minimizing the square residual

$$R(A_1, x_1, \sigma_1, A_2, x_2, \sigma_2) = \sum_{k=1}^{K} [y^* - y(x_k; A_1, x_1, \sigma_1, A_2, x_2, \sigma_2)]^2 \ , \qquad (8)$$

with respect to the 6 parameters defining the two Gaussians. If one is told *a priori* that two Gaussians are to be fit to the data, then this residual minimization problem is obviously equivalent to a 6-D function maximization problem for $1/R$ (say), which simply defines a function in 6-D space. Figure 6 shows the dataset generated using the parameter set

$$[A_1, x_1, \sigma_1, A_2, x_2, \sigma_2] = [0.9, 0.3, 0.1, 0.3, 0.8, 0.025] \qquad (9)$$

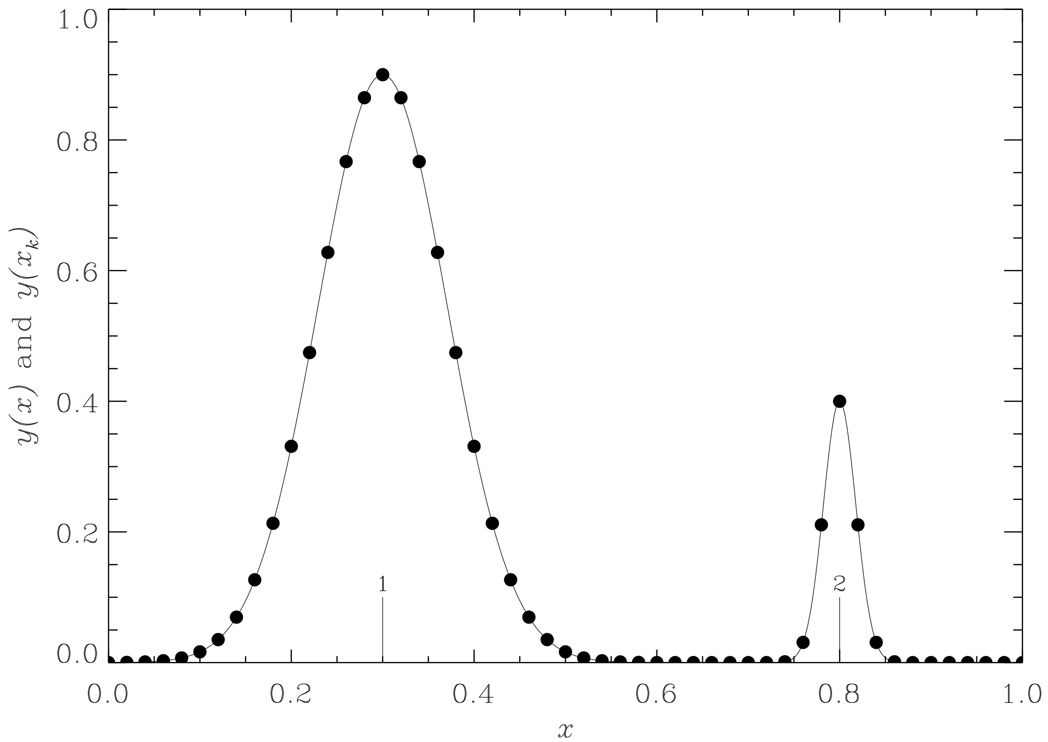

Figure 6: Test problem P4. This is a 6-parameter problem which consists in fitting two Gaussians to a "dataset" of 51 points. Note how the second Gaussian is poorly sampled by the discretization in x. The thin solid line is the underlying 2-Gaussian function defined by eq. (7).

and $K = 51$ discretization points in x. Once again the resulting minimization problem is not an easy one; given the discretization in x, the minimization is largely dominated by the need to accurately fit the broader, high amplitude first Gaussian; the second Gaussian is not only of much lower amplitude, it is also poorly sampled in x. Fitting only the first Gaussian leads to a reasonably low residual ($R \simeq 0.25$); global accuracy requires the second Gaussian to be also "detected" and fit, in which case only does $R \to 0$.

The simplex succeeds in properly fitting both Gaussians 123 out of 10^3 trials. What are the "secondary minima" on which the simplex remains stuck? They can be divided into two broad classes: (1) one of the model Gaussians fits the broad, higher amplitude component, and the other is driven to zero, either by having $A \to 0$ or $\sigma \to 0$; (2) the method returns a two Gaussians solution, where $x_1 = x_2 = 0.3$, $\sigma_1 = \sigma_2 = 0.1$, and $A_1 + A_2 = 0.9$. The 6-D parameter space contains long, flat "valleys" and "plains" of low but suboptimal residual values in which the simplex grinds to a halt.

Table I

Simplex performance measures on test problems

Test Problem	Performance	Simplex	Iterated simplex
P1	$\langle 1-f \rangle$	0.633	0.00793
	p_G	**0.0213**	**0.898**
	$\langle N_f \rangle$	37	3872
	N_t	1	100
P2	$\langle 1-f \rangle$	0.194	0.0619
	p_G	**0.0263**	**0.931**
	$\langle N_f \rangle$	44	4412
	N_t	1	100
P3	$\langle 1-f \rangle$	0.413	0.07713
	p_G	**0.00006**	**0.016**
	$\langle N_f \rangle$	70	35252
	N_t	1	500
P4	$\langle R \rangle$	0.332	0.0069
	p_G	**0.123**	**0.941**
	$\langle N_f \rangle$	753	37638
	N_t	1	20

1.5 Performance of the simplex and iterated simplex methods

Table I summarizes the performance of the simplex and iterated simplex methods on the four test problems. Each entry represents an average over at least 1000 independent runs (up to 10^5 for P3), and so should be fairly representative of the methods' behavior on each test problem. For each problem the Table gives the absolute performance, defined here as the average over all runs of either $1 - f(\mathbf{x})$, for P1, P2 and P3, or the residual R (cf. eq.[8]) for P4. The global performance is defined in terms of a probability measure (p_G) as the fraction of all runs for which the true, global extremum has been located ($f \geq 0.95$ for P1, P2 and P3) or the second, smaller Gaussian has been properly fit ($R \leq 0.1$ for P4). As a measure of relative performance the table simply lists the average number of function/model evaluations $\langle N_f \rangle$ required by each method. The last entry for each problem is the number of trials N_t executed by iterated simplex (this number is 1 by definition for the basic simplex method without restart).

At this stage only a few comments need be made on the basis of Table I. The first is that, as advertised, all four test problems are hard global optimization problems, as can be judged from the poor global performance of the basic simplex method on each. Turning to iterated simplex leads to spectacular improvement in global performance in all cases, but of course the number of required function evaluations goes up by a few orders of magnitude. In fact the global performance of iterated simplex can be predicted on the basis of the single-run simplex. The global performance on the later can be viewed as a probability (p) of locating the global maximum; the (complementary) probability of a given run *not* to do so is $1-p$; the probability of *all* iterated simplex runs *not* finding the global maximum is then $(1-p)^{N_t}$, so that the probability of *any one* of N_t iterations locating the global maximum is

$$p_G = 1-(1-p)^{N_t} \ . \qquad \text{[Iterated hill climbing]} \tag{10}$$

On the basis of eq. (10) one would predict global performances (0.884, 0.930, 0.029, 0.927) on P1 through P4, given the number of hill climbing iterations listed in the rightmost column of Table 1, which compares quite well with the actual measured global performance. One can also rewrite eq. (10) as

$$N_t = \frac{\log(1-p_G)}{\log(1-p)} \ , \qquad \text{[Iterated hill climbing]} \tag{11}$$

to predict the expected number of hill climbing iterations required to achieve a global performance level p_G; with $p \simeq 6\times10^{-5}$ for P3, requiring $p_G = 0.95$ would demand (on average) $N_t \simeq 50000$ hill climbing trials, adding up to a grand total of about 3.5×10^{6} function evaluations since a single simplex run on P3 carries out on average 70 function evaluations (cf. Table 1). Iterated hill climbing certainly works, but there really is no such thing as a free lunch...

It is easy to predict the expected global performance of iterated simplex because each trial proceeds completely independently. The improvement in global performance simply reflects the better initial sampling of parameter space associated with the initial distribution of simplex vertices. Everything else being equal, as problem dimensionality (n) increases the number of trials N_t required can be expected to scale as $N_t \propto a^n$, where a is some number characterizing in this case the fraction of parameter space covered by the global maximum. Iterated simplex is not only demanding in terms of function evaluations, but in addition it does not scale well at all on a given problem as dimensionality is increased. This, in fact, is the central problem facing iterated hill climbing in general, not just its simplex-based incarnation.

The poor scalability of iterated hill climbing stems from the fact that *each trial proceeds independently.* The challenge in developing global methods that are to outperform iterated hill climbing consists in introducing a transfer of information between trial solutions, in a manner that continuously "broadcasts" to each paratrooper in the squadron the topographical information garnered by each individual paratrooper in the course of his/her local hill climb. The challenge, of course is to achieve this without overly biasing the ensemble of trials.

A relatively well-known method that often achieves this reliably is *simulated annealing* (Metropolis *et al.* 1953; see also Press *et al.* 1992, §10.9). Simulated annealing is inspired by the global transfer of energy/information achieved by colliding constituent particles of a cooling liquid metal, which allows the substance to achieve the crystalline/metallic configuration that minimizes the total energy of the whole system. The algorithmic implementation of the technique for numerical optimization requires the specification of a *cooling schedule*, which is far from trivial: fast cooling is computationally efficient (low $\langle N_f \rangle$) but can lead to convergence on a secondary extremum (low p_G), while slow cooling improves global convergence (high p_G), but at the expense of a high $\langle N_f \rangle$. No Free Lunch, remember...

Genetic Algorithms achieve the same goal, but are inspired by the exchange of genetic information occurring in a breeding population subjected to natural selection. They can be used to form the core of very robust, global numerical optimization methods, as detailed in Section 3 below. The following Section provides a brief introduction to genetic algorithms in a more general sense.

The least you should remember from Section 1:

- Global optimization is a totally different game from local optimization.
- You should never feel lucky.
- There is no such thing as a free lunch.
- You can always design a problem that will defeat any global optimization method.

Exercises for Section 1:

(1) Look back at Figure 4. Whenever the simplex fails to achieve global convergence (i.e., $1 - f \to 0$) it seems to remain stuck at a discrete set of $1 - f$ values. What do these values correspond to?

(2) Consider again the use of iterated simplex on the test problem of Figure 2; calculate the fractional surface area of the part of the central peak that lies higher that the innermost ring of secondary maxima. On this basis what

would you predict the required number of simplex trials to be, on average, for iterated simplex to locate the central peak.

(3) Repeat the same analysis as for Exercise (2) above, but in the context of the P3 test problem. Are your results in basic agreement with Table 1? How can you explain the differences (if any)?

---

2. EVOLUTION, OPTIMIZATION, AND GENETIC ALGORITHMS

2.1 Biological evolution

The general ideas of *evolution* and *adaptation* predate Charles Darwin's *On the Origin of Species by Means of Natural Selection* (1859), but it is Darwin (and more or less simultaneously Alfred Russell Wallace) who first identified what is still considered to be the primary driving mechanism of evolution: *natural selection*.

Nature is very much oversubscribed. At almost any time in any ecosystem, far more individuals are born than can possibly survive given the ecosystem's available resources. This implies that many members of a given species will die from attrition or predation before they have a chance to reproduce. The principle of natural selection states that individuals better adapted to their environment, i.e., for whatever reason better at obtaining lunch, avoiding becoming lunch, and finding/attracting/competing for mates, will, on average, leave behind more offspring than their less apt colleagues.

For natural selection to lead to *evolution*, two more essential ingredients are required: (1) *inheritance*: offspring must retain at least some of the features that made their parents fitter than average, otherwise evolution is effectively reset at every generation; (2) *variability*: at any given time individuals of varying fitnesses must coexist in the population, otherwise natural selection has nothing to operate on.

Both these additional requirements were plainly obvious to Darwin and his contemporaries, but their underlying mechanisms remained unexplained in their lifetime. However, this situation changed rapidly in the early decades of the twentieth century, and the primary processes through which heredity is mediated and variation maintained are now basically understood. In a nutshell, the information determining the growth and development of individuals is encoded as linear sequences of genes that can each assume a finite set of "values". In sexual species, when two individuals breed, complementary portions of their genetic material are passed on to their offspring and combined to define that offspring's full genetic makeup. That's the inheritance part. In the course of "preprocessing" the genetic material to be later passed on to offspring, copy mistakes and truly random alteration of some gene values also occur occasionally. These mutation events, coupled to the fact that an offspring receives complementary genes from two parents (which is true of most animals), provides the needed source of variability.

The individual that moves, feeds and mates in real space can be looked at as an outer manifestation of its defining genes[8]. Think then of an individual's fitness as a function of the values assumed by its genes. What evolution does is to drive a gradual increase in average fitness values over the course of many generations. This is what Darwin called *adaptation*. Now that's beginning to sound like hill climbing, doesn't it? In fact evolution does not optimize, at least not in the mathematical sense of the word. Evolution is blind. Evolution does not give a damn about globally maximal fitness (n'en déplaise à Teilhard de Chardin). Even if it did, evolution must accommodate physical constraints associated with development and growth, so that not all paths are possible in genetic "parameter space". All evolution does is produce individuals of above-average fitness. Nonetheless, the basic ideas of natural selection and inheritance with variation can be used to construct very robust algorithms for global numerical optimization.

2.2 The power of cumulative selection

The idea that natural selection can lead to a form of hill climbing in fitness space may become intuitively obvious... after thinking about it for a while! What remains less obvious is the degree to which cumulative selection, i.e., selection operating on successive generations, can accelerate what would in its absence be a random search of genetic parameter space. The following example, popularized in Richard Dawkins' *The Blind Watchmaker* (1986; a book well worth reading, incidentally), makes for such a nice demonstration of this very point that it has by now found its way into at least one textbook on evolutionary genetics (Maynard Smith 1989; an excellent introduction to the topic). Consider the following sentence[9]:

`JEG SNAKKER BARE LITT NORSK`

This sentence is 27 characters long including blank spaces, and is made up of an alphabet of 30 letters if a blank character is included (please note that I *am* taking into account the famous Scandinavian letters Å, Ø, and Æ). Consider now

---

[8] This is said without at all denying that a large part of what makes us who we are arises from learning and other interactions with the environment in the course of development and growth; what genes encode is some sort of basic behavioral *Bauplan* from which these higher level processes take off.

[9] The original sentence used by Dawkins is `METHINK IT IS LIKE A WEASEL`, which, of course, is taken from Shakespeare's *Hamlet*.

the process of producing 27-character-long sentences by randomly selecting letters from the 30 available characters of the alphabet. Here's an example:

`GE YTÅUMNBGH JHØA QMWCXNESØ`

Doesn't look much like the original sentence... although careful comparison will show that two letters actually coincide. The total number of distinct 27-character-long sentences that can be made out of a 30-character alphabet is $30^{27} = 7.63 \times 10^{39}$. This is a *very* large number, even by astronomical standards. The corresponding probability of generating our first, target sentence by this random process on the first trial is then $(30)^{-27} \simeq 10^{-40}$. This is such a small number that invoking the Dirty Harry Rule at this point would be moot. Instead consider the following procedure:

(1) Generate 10 sentences of 27 randomly chosen characters;
(2) Select the sentence that has the most correct letters;
(3) Duplicate this best sentence ten times;
(4) For each such duplicate, randomly replace a few letters[10];
(5) Repeat steps (2) through (4) until the target sentence has been matched.

This search algorithm incorporates the three ingredients mentioned previously as essential to the evolutionary process. Step (2) is natural selection, in fact in a deterministic and rather extreme form since the best and only the best acts as progenitor to the next "generation". Step (3) is inheritance, again of a rather extreme form as offspring start off as exact replicas of the (single) progenitor. Step (4) is a stochastic process which provides the required variability. Note also that the algorithm operates with minimal "fitness" information; all it has available is how many correct letters a sentence contains, but not *which* letters are correct or incorrect. What is still missing is exchange of information between trial solutions, but be patient, this will come in due time.

Figure 7 illustrates the "evolution" of the best-of-10 sentence, starting from an initial ten random sentences, as described above. The mutation rate was set at $p = 0.01$, meaning that any given letter has a probability 0.01 of being subjected to random replacement. Iteration count is listed in the leftmost column, and error in the rightmost column. Error is defined here simply as the number of incorrect letters in the best sentence generated in the course of the current iteration. Note how the error decreases rather rapidly at first, but much more slowly later on; it takes about as many iterations to get the first 20 letters right as it takes to get the last one. The target sentence is found after only 918 iterations, in the course

[10] More precisely, define a *mutation rate* as the probability p ($\in [0,1]$) that a given constituent letter be randomly replaced.

Target	JEG SNAKKER BARE LITT NORSK	
1	ZEBYENÆTUVP QÅODEMI FVGHDOO	23
50	VEGÆENÆROEO QÅBDEMI FVNÅDOK	19
100	VEGÆENÆKCEO OPHZEMI FVNÅØOK	18
150	VEGWXNÆKCEO NAHADMICFNNEROK	16
200	JEGWXNPKKEO BAHA RICEÅNEROK	12
250	JEGWVNRKKE BAEA RIÅEÅN RTK	12
300	JEG RNRKKE BAET UIØØ NQRKK	10
350	JEG KNKKKE BAR UIØØ NQRMK	9
400	JEG KNVKKE BAR PIHØ NQRSK	8
450	JEG KNVKKER BAR LIDØ NÅRSK	6
500	JEG ØNVKKER BAR LISØ NKRSK	6
550	JEG ØNFKKER BARE LISI NBRSK	5
600	JEG SNFKKER BARE LIAW NBRSK	4
650	JEG SNAKKER BARE LIAW NORSK	2
700	JEG SNAKKER BARE LIAT NORSK	1
750	JEG SNAKKER BARE LIAT NORSK	1
800	JEG SNAKKER BARE LIAT NORSK	1
850	JEG SNAKKER BARE LIAT NORSK	1
900	JEG SNAKKER BARE LIYT NORSK	1
950	JEG SNAKKER BARE LITT NORSK	0

Figure 7: Accelerated Norsk learning by means of cumulative selection. Iteration count is listed in the left column, and the error, defined as the number of incorrect letters, in the rightmost column. The target sentence is found after 918 iterations.

of which 9180 trial sentences were generated and "evaluated" against the target. This is almost infinitely less than the $\sim 10^{40}$ of enumerative or purely random search.

Figure 8 shows convergence curves for three runs starting with the same initial random sentence, but evolving under different mutation rates. The solid line is the solution of Figure 7. Note how the solution with the highest mutation rate converges more rapidly at first, but eventually levels off at a finite, nonzero error level. What is happening here is that mutations are producing the needed correct letters as fast as they are destroying currently correct letters. Given an alphabet size and sentence length, there will always exist a critical mutation rate above which this will happen[11].

There are two important things to remember at this point. First, mutation

[11] This is in fact a notion central to our understanding of the emergence of life. Among a variety of self-replicating molecules of different lengths "competing" for chemical constituents in limited supply in the primaeval soup, those lying closest to the critical mutation rate can adapt the fastest to an evolving chemical

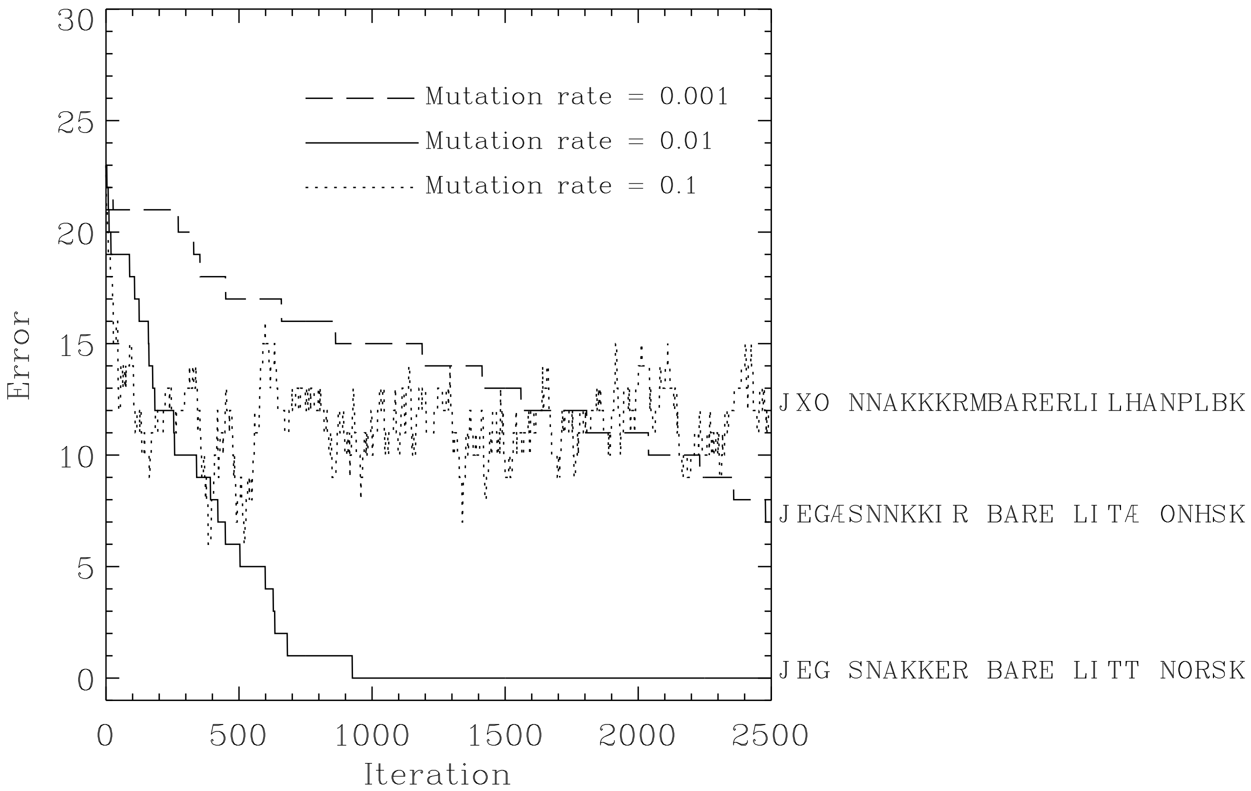

Figure 8: Convergence curves for the sentence search problem, for three different mutation rates. The curves show the error associated with the best sentence produced at each iteration. The solid line corresponds to the solution shown on Figure 7.

is a mixed blessing. It is clearly needed as a source of variability, but too much of it is definitely deleterious. Second, the general shape of the convergence curves in Figure 8 is worth noting. Convergence is rather swift at first, but then levels off. This is a behavior we will meet again and again in what follows. Time now to move on, finally, to genetic algorithms.

environment without self-destructing, and so rapidly take over the soup (see, Eigen 1971 for a comprehensive though somewhat dated review). This is conjectured to be the explanation behind the universality of the genetic code among very nearly all living organisms.

2.3 A basic genetic algorithm

Fundamentally, genetic algorithms are a class of search techniques that use simplified forms of the biological processes of selection/inheritance/variation. Strictly speaking they are not optimization methods *per se*, but can be used to form the core of a class of robust and flexible methods known as *genetic algorithm-based optimizers*.

Let's go back to a generic optimization problem. One is given a "model" that depends on a set of parameters $\mathbf{u}$, and a functional relation $f(\mathbf{u})$ that returns a measure of quality, or *fitness*, associated with the corresponding model (this could be a χ^2-type goodness of fit measure if the model is compared to data, for example). The optimization task usually consists in finding the "point" $\mathbf{u}^*$ in parameter space corresponding to the model that maximizes the fitness function $f(\mathbf{u})$. Define now a population as a set of N_p realizations of the parameters $\mathbf{u}$. A top-level view of a basic genetic algorithm is then as follows:

(1) Randomly initialize population and evaluate fitness of its members;
(2) Breed selected members of current population to produce offspring population (selection based on fitness);
(3) Replace current population by offspring population;
(4) Evaluate fitness of new population members;
(5) Repeat steps (2) through (4) until the fittest member of the current population is deemed fit enough.

Were it not that what it being cycled through the iteration is a population of solutions rather than a single trial solution, this would very much smell of iterated hill climbing. It should also give you that uncanny feeling of *déjà vu*, unless your memory is really shot or, shame on you, you have skipped over the preceding section. The crucial novelty lies with step 2: *Breeding*. It is in the course of breeding that information is passed and exchanged across population members. How this information transfer takes place is rather peculiar, and merits discussion in some detail, and not only because this is where genetic algorithms justify the "genetic" in their name.

Figure 9 illustrates the breeding process in the context of a simple 2-D maximization problem, such as the P1 or P2 test problems. In this case an individual is a (x, y) point, and so is "defined" by two floating point numbers. The first step is to *encode* the two floating point numbers defining each individual selected for breeding. Here this is done simply by dropping the decimal point and concatenating the resulting set of simple decimal integers into a "chromosome"-like string (lines 01—06 on Figure 9). Breeding proper is a two step process. The first step is *crossover*. The two strings generated by the encoding process are laid side by side, and a cutting point is randomly selected along the length of the defining strings. The string fragments located right of the cutting point are then interchanged, and spliced onto the fragments originally located left of the cutting point (lines 07—12, for a cutting point located between the third and fourth decimal digit). The second breeding step is *mutation*. For each string produced by the crossover process, a few randomly selected digits (or "genes") are replaced by a new, randomly selected digit value (lines 13—16, for a mutation hitting the tenth digit of the second offspring string). The resulting fragments are then decoded into two (x, y) pairs, whose fitness is then evaluated, here simply by computing the function value $f(x, y)$.

Some additional comments are in order. *First*, note that offspring incorporate intact "chunks" of genetic material coming from *both* parents; that's the needed inheritance, as well as the promised exchange of information between trial solutions. However, both the crossover and mutation operations also involve purely stochastic components, such as the choice of cutting point, site of mutation, and new value of mutated digit. This is where we get the variability needed to sustain the evolutionary process, as discussed earlier. *Second*, the encoding/decoding process illustrated on Figure 9 is just one of many possible such schemes. Traditionally, genetic algorithms have made use of *binary* encoding, but this is often not particularly advantageous for numerical optimization. The use of a decimal genetic "alphabet" is no more artificial than a binary representation, even more so given that very nearly all known living organisms encode their genetic information in a base-4 alphabet. In fact, in terms of encoding floating-point numbers, both binary and decimal alphabets suffer from significant shortcomings that can affect the performance of the resulting optimization algorithms. *Third*, the crossover and mutation operators, operating in conjunction with the encoding/decoding processes as illustrated on Figure 9, preserve the total range in parameter space. That is, if the floating-point parameters defining parent solutions are restricted to the range $[0.0, 1.0]$, then the offspring solution parameters will also be restricted to $[0.0, 1.0]$. This is a very important property, through which one can effortlessly hardwire constraints such as positivity. *Fourth*, having the mutation operator act on the encoded form of the parent solution has the interesting consequence that offspring can differ very much or very little from their parents, depending on whether the digits affected by mutation decode into one of the leading or trailing digits of the corresponding floating-point number. This means that from the point of view of parameter space exploration, a genetic algorithm can carry out both wide exploration and fine tuning in parallel. *Fifth*, it takes two parents to produce (simultaneously) two offspring. One can of course devise orgiastic breeding schemes that involve more than two parents and yield any number of offspring.

	P(P1)	x=0.14429628	y=0.72317247	[01]
Encoding:	P(P2)	x=0.71281369	y=0.83459991	[02]
		↓	↓	
		71281369	83459991	[03]
	S(P2)	7128136983459991		[04]
Breeding:	S(P1)	1442962872317247		[05]
	S(P2)	7128136983459991		[06]
(a) Crossover (gene=4):				
		144 | $\boxed{2962872317247}$		[07]
		↓↑		
		712 | $\boxed{8136983459991}$		[08]
		144 $\boxed{8136983459991}$		[09]
		712 $\boxed{2962872317247}$		[10]
	S(O1)	1448136983459991		[11]
	S(O2)	7122962872317247		[12]
(b) Mutation (Offspring=O2, gene=10):				
	S(O2)	7122962872317247		[13]
		712296287 $\boxed{2}$ 317247		[14]
		712296287 $\boxed{8}$ 317247		[15]
	S(O2)	7122962878317247		[16]
Decoding:	S(O2)	7122962878317247		[17]
		71229628	78317247	[18]
		↓	↓	
	P(O2)	x=0.71229628	y=0.78317247	[19]
	P(O1)	x=0.14481369	y=0.83459991	[20]

Figure 9: Breeding in genetic algorithms. Here the process is illustrated in the context of a 2-D maximization problem (such as P1 or P2 of §1.4). An individual is an (x, y) point, and two such parent individuals are needed for breeding (denoted P(P1) and P(P2) here). The *one-point crossover* and *one-point mutation* operators act on string representations of the parents (S(P1) and S(P2)) to produce offspring strings S(O1) and S(O2), which are finally decoded into two offspring (x, y) points P(O1) and P(O2).

Experience shows that this rarely improves the performance of the resulting algorithms. *Sixth*, $f(\mathbf{u})$ must obviously be computable for all $\mathbf{u}$, but not necessarily differentiable since derivatives of the fitness function with respect to its input parameters are not required for the algorithm to operate. From a practical point of view, this can be a great advantage.

2.4 Information transfer in genetic algorithms

Time to step back and revisit the issue of information processing. Genetic algorithms achieve transfer of information through the breeding of trial solutions selected on the basis of their fitness, which is why the crossover operator is usually deemed to be *the* defining feature of genetic algorithms, as compared to other classes of evolutionary algorithms (see, e.g., Bäck 1996).

The joint action of crossover and fitness-based selection on a population of strings encoding trial solutions is to increase the occurrence frequency of substrings that convey their decoded trial solution above-average fitness, at a rate proportional to difference between the average fitness of all trial solutions including that substring in their "genotype" (i.e., the string-encoded version of their defining parameter set), and the average fitness of the whole population. The mathematical expression of the preceding mouthful, adequately expanded to take into account the possibility of substring disruption by crossover or mutation, is known as the *Schema Theorem*, and is originally due to Holland (1975; see also Goldberg 1989). As the population evolves in response to breeding and fitness-based selection, advantageous substrings are continuously sorted and combined by crossover into single individuals, leading to an inexorable fitness increase in the population as a whole. Because this involves the concurrent processing of a great many distinct substrings, Holland dubbed this property *intrinsic parallelism*, and argues that therein fundamentally lies the exploratory power of genetic algorithms.

The least you should remember from Section 2:

- Natural selection alone cannot lead to evolution; inheritance and variation are also needed.
- Cumulative selection can accelerate an otherwise random search process by a factor that is astronomically enormous.
- Genetic Algorithms are search techniques that make use of simplified forms of the biological selection/inheritance/variation triad.

Exercises for Section 2:

All exercises for this part of the tutorial aim at letting you explore quantitatively the probabilistic aspects of the sentence search example of §2.2.

(1) First some basic probability calculations, to warm up. (a) what is the probability of getting *all* of the letters wrong on an initial random trial? (b) getting at least one letter (any letter) right? (c) getting exactly one letter (any letter) right?

(2) In the run of Figure 7, it took 671 iterations to get to the point of having 26 correct letters out of 27. What is now the probability of obtaining a fully correct sentence in one of the ten mutated copies after the subsequent iteration? What is the probability of *all* mutated copies having regressed to only 25 correct letters?

(3) Given the sentence length $S = 27$, alphabet size $A = 30$, and a mutation rate p, obtain an *estimate* (i.e., not a formal calculation) for the number of iterations required, on average, to reach zero error. How does your estimate compare to Figure 7? Do you think that Figure 7 is a typical solution?

(4) Given again a sentence length S, an alphabet size A, and a mutation rate p, calculate the error level at which the sentence search algorithm will saturate (like the dotted line on Figure 9). Use this result to estimate an optimal mutation rate as a function of S and A that will, on average, lead to convergence in the smallest possible number of iterations.

(5) In terms of an analogy for biological evolution, what do you think are the most significant failings of the sentence search example?

3. PIKAIA: A GENETIC ALGORITHM FOR NUMERICAL OPTIMIZATION

3.1 Overview and problem definition

In this section we will be primarily concerned with the comparison of genetic algorithm-based optimizers with other global optimization schemes, specifically iterated hill climbing using the simplex method (§1.2). To do so we first need to settle on a specific implementation of a genetic algorithm.

`PIKAIA` is a public domain, general purpose genetic algorithm-based optimization subroutine. It is written in FORTRAN-77, is completely self-contained, and is designed to be as easy to use as the optimization subroutines found in Press *et al.*'s *Numerical Recipes* (for example their simplex routine `amoeba`). It comes with limited I/O capabilities and no fancy graphics. The software is described in great detail in the *User's Guide to PIKAIA 1.0* (Charbonneau & Knapp 1995; hereafter PUG), to which numerous references are made in what follows. The software and User's Guide can both be obtained from the `PIKAIA` Web Page:

`http://www.hao.ucar.edu/public/research/si/pikaia/tutorial.html`

This section opens with a brief overview of the operators and techniques included in `PIKAIA`. Internally, `PIKAIA` seeks to *maximize* a user-defined function $f(\mathbf{x})$ in a bounded n-dimensional space, i.e.,

$$\mathbf{x} \equiv (x_1, x_2, ...x_n) \ , \qquad x_k \in [0.0, 1.0] \quad \forall k \ . \tag{12}$$

The restriction of parameter values in the range $[0.0, 1.0]$ allows greater flexibility and portability across problem domains. This, however, implies that the user must adequately normalize the input parameters of the function to be maximized with respect to those bounds.

The maximization is carried out on a population made up of N_p individuals (trial solutions). This population size remains fixed throughout the evolution. Rather than evolving the population until some tolerance criterion is satisfied, `PIKAIA` carries the evolution over a user-defined, preset number of generations N_g.

PIKAIA offers the user the flexibility to specify a number of other input parameters that control the behavior of the underlying genetic algorithm. The subroutine does include built-in default settings that have proven robust across problem domains. All such input parameters are passed to PIKAIA in the 12-dimensional control vector `ctrl`. See Section 4 of the PUG for the allowed and default values of those control parameters.

The top-level structure of PIKAIA is the same as the sequence of algorithmic steps listed in §2.3: an outer loop controlling the generational iteration, and an inner loop controlling breeding. Since breeding involves the production of *two* offspring, the inner loop executes $N_p/2$ times per generational iteration, where N_p is the population size ($N_p = 100$ is the default value).

All parameter values defining the individual members of the initial population are assigned a random number in the range $[0.0, 1.0]$, extracted from a uniform distribution of random deviates (see §3.3 of the PUG). *This ensures that no initial bias whatsoever is introduced by the initialization.*

3.2 Minimal algorithmic components

3.2.1 Selection [PUG, §3.4]

PIKAIA uses a stochastic selection process to assign to each individual in the population a *probability* of being selected for breeding. Specifically, that probability is made linearly proportional to the fitness-based *rank* of each individual within the current population. This is carried out using a scheme known as the *Roulette Wheel Algorithm*, as detailed in §3.4 of the PUG (see also Davis 1991, chap. 1). Note that in general it is *not* a good idea to make selection probability directly proportional to fitness *value*, as this often leads to a loss of selection pressure late in the evolutionary run, once most population members have "found" the global optimum. In some cases it can also lead, early on, to a "superindividual" being selected so frequently that the population becomes degenerate through the computational equivalent of inbreeding. The proportionality constant between fitness-based rank and selection probability is specified as an input parameter to PIKAIA. The default value is 1.0.

3.2.2 Breeding [PUG, §§3.5, 3.6 and 3.7]

Once two individuals have been selected, breeding proceeds exactly as in Figure 9. The encoding process requires one to specify the number of digits to be retained in the encoding process; this is a user-specified quantity, which is set to 5 in all calculations reported upon here (this is also the default value in PIKAIA). Two additional quantities need to be specified: (1) the crossover rate, which sets the probability that the crossover operation actually takes place (default is 0.85); (2) the mutation rate, which sets the probability, for *each* digit making up the defining string of an offspring, that a mutation takes place at that digit location (default is 0.005).

3.2.3 Population replacement [PUG, §3.8]

Under `PIKAIA`'s default settings the offspring population is accumulated into temporary storage, and once the number of such offspring equals that of the current breeding population the latter is deleted and replaced by the offspring population. This is the default strategy used by `PIKAIA`, although it is possible for the user to specify other population replacement techniques (see PUG, §§3.8.2, 3.8.3).

3.3 Additional components

The components listed above define a minimal genetic algorithm. Such an algorithm can be used for numerical optimization, but as we will soon see, turns out to be far from optimal. In what follows we refer to this algorithm as GA1[12]. The following two simple additions to GA1 lead to an algorithm (to be referred to as GA2) that achieves far better performance on numerical optimization problems. So much better in fact that the use of these two additional components is the default choice in `PIKAIA`[13].

3.3.1 Elitism [PUG, §3.9]

This simply consists in storing away the parameters defining the fittest member of the current population, and later copying it intact in the offspring population. This represents a safeguard against the possibility that crossover and/or mutation destroy the current best solution, which would have a good chance of unnecessarily slowing down the optimization process. Elitism in fact becomes essential upon introducing our second, vital improvement to GA1.

3.3.2 Variable mutation rate [PUG, §3.7.2]

This one is perhaps the single most important improvement that can (and should!) be made to GA1. As discussed in §2.3, mutation is very much a mixed blessing;

[12] For those of you who might want to run `PIKAIA` to reproduce the results below, GA1 is produced by explicitly setting the following elements of `PIKAIA`'s control vector `ctrl`: `ctrl(5)=1.`, `ctrl(11)=0.`, and all other elements of `ctrl` to negative values to activate default options.

[13] This means initializing *all* elements of the control vector `ctrl` to negative values. Note that this sets a population size equal to 100 (via `ctrl(1)`), and a number of generations equal to 500 (via `ctrl(2)`).

it provides the much needed source of variability through which novel parameter values are injected into the population. However, it also leads to the destruction of good solutions. This was precisely the point of Figure 8 (dotted line). Finding the exact value for the mutation rate that achieves optimal balance between those two effects to maximize the former while minimizing the latter is of course possible[14]. However, in doing so one finds that the optimal parameter settings often end up being highly problem dependent[15].

One powerful solution to this problem is to dynamically adjust the mutation rate. The key to this strategy lies with recognizing that as long as the population is broadly distributed in parameter space, the crossover operator leads to a pretty efficient "search" as it recombines fragments of existing solutions. However, once the population has converged —whether on a secondary or absolute optimum— crossover no longer achieves much, as it leads to the exchange of fragments that are nearly identical since all parents have nearly identical parameter values. This, obviously, is where a high mutation rate is needed to reinject variability into the population.

Consider then the following procedure. At any given time, keep track of the fitness value of the fittest population member, and of the median ranked member. The fitness difference Δf between those two individuals is clearly a measure of population convergence; if Δf is large the population is presumably distributed more broadly in parameter space than if Δf is very small. Therefore, if Δf becomes too small, increase the mutation rate; if it becomes too large, decrease the mutation rate again. This is how `PIKAIA` dynamically adjusts its mutation rate during run-time. This strategy represents a simple form of *self-adaptation* of a parameter controlling the behavior of the underlying genetic algorithm. Further details and implementation issues are discussed in §3.7.2 of the PUG.

3.4 A case study: GA2 on P1

It will prove useful to first take a detailed look at the behavior of the genetic algorithm in the context of a simple problem. Figure 10(A) shows ten convergence curves for GA2 working on P1 with $N_p = 50$. What is plotted is one minus the

[14] In fact this is often done by letting the mutation rate (and other controlling parameters of the algorithm) evolve under the control of a second, higher level genetic algorithm, with fitness being then defined as the performance of the genetic algorithm defined by those parameters on the problem under consideration. Pretty cute but, as you might imagine, rather time consuming.

[15] Just as the optimal mutation rate you (hopefully) worked out in Problem 4 of §2 is rather sensitively dependent on the sentence length and alphabet size.

fitness value of the *fittest* individual versus generation count, for 10 separate runs of GA2. Figure 10(A) should be compared to Figure 4, showing the convergence of the simplex on the same problem. Early on, the curves have qualitatively similar shapes[16]; either convergence occurs relatively quickly (much more quickly for simplex, when it does converge), or solutions remain "stuck" on one of the rings of secondary extrema (cf. Fig. 2), which leads to the error leveling off at a fixed value. *Unlike simplex, however, GA2 is able to pull itself off the secondary extrema rings.* It does so primarily through mutation, although crossover between two parents properly positioned in parameter space can achieve the same effect. Mutation being a fundamentally stochastic process, it is then not surprising to see different GA2 runs requiring different generation counts before the needed favorable mutation takes place.

Clearly mutation plays a critical role here. Figure 10(B) shows the fitnesses of the best (solid line) and median-ranked (dashed line) individuals in the population as a function of generational count, for the GA2 run plotted with a thicker line on panel (A). The dotted line shows the variation of the mutation rate. Figure 11 shows the distribution of the population in 2-D parameter space[17], at the epochs indicated by solid dots on Fig. 10(B).

To start with, note on Fig. 11(A) that no individual in the initial random population has landed anywhere close enough to the central peak for hill climbing to work. The first few generational iterations see the population cluster itself closer and closer to center (Fig. 11[B]), but the fitness difference between best and median is still quite large. The mutation decreases slightly from its initial (low) value, but then remains constant. By the 15^{th}generation (Fig. 11[C]) most of the population has converged somewhere on the inner ring of secondary extrema ($f = 0.9216$), so that the fitnesses of the best and median are now comparable. This leads to a sharp increase of the mutation rate (between the $12^{\text{th}}$and 20^{th}generations). The high mutation rate results in offspring being knocked all over parameter space in the course of breeding (Fig. 11[D]). While some mutant individuals do land regularly on the slope of the central peak, it is only by the 55^{th}generation that one such mutant is catapulted high enough to become the fittest of the current population (Fig. 11[E]). Further breeding during subsequent generations brings more and more individuals to the central peak and further increases in fitness of

---

[16] You might notice that GA2 already starts off doing significantly better than the simplex method; this merely results from the initial random population of GA2 having "sampled" 50 points in parameter space, compared to only 3 for the simplex.

[17] An animation of the evolving population for this solution can be viewed on the Tutorial Web Page.

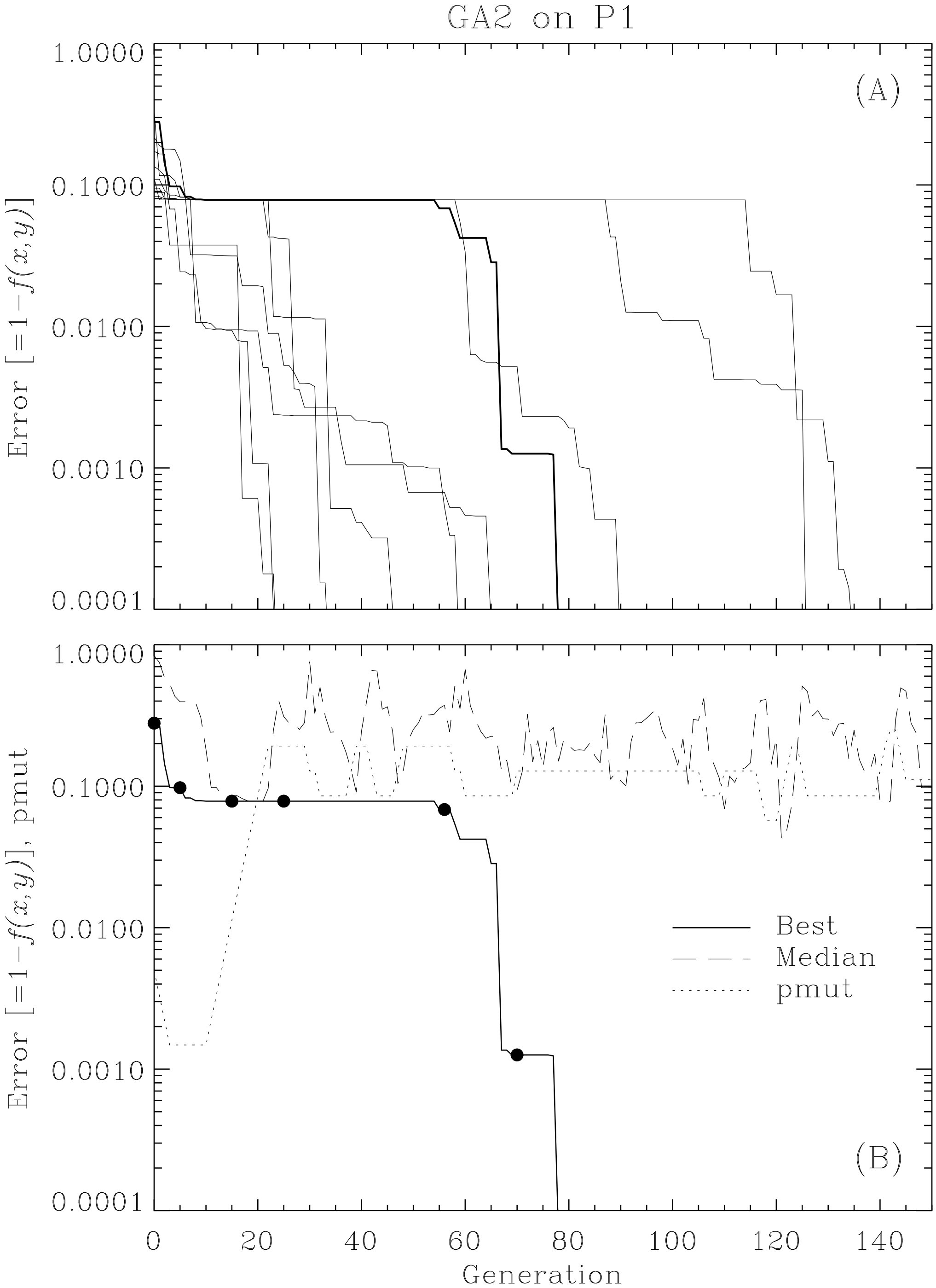

Figure 10: Panel (A) shows convergence curves for 10 distinct runs of GA2 on P1. As before the error is defined as $1-f(x,y)$. Panel (B) shows, for the single run plotted with a thicker line on panel (A), the variations with generation count of the best individual of the population (solid line), median-ranked individual (dashed line), and mutation rate (dotted line).

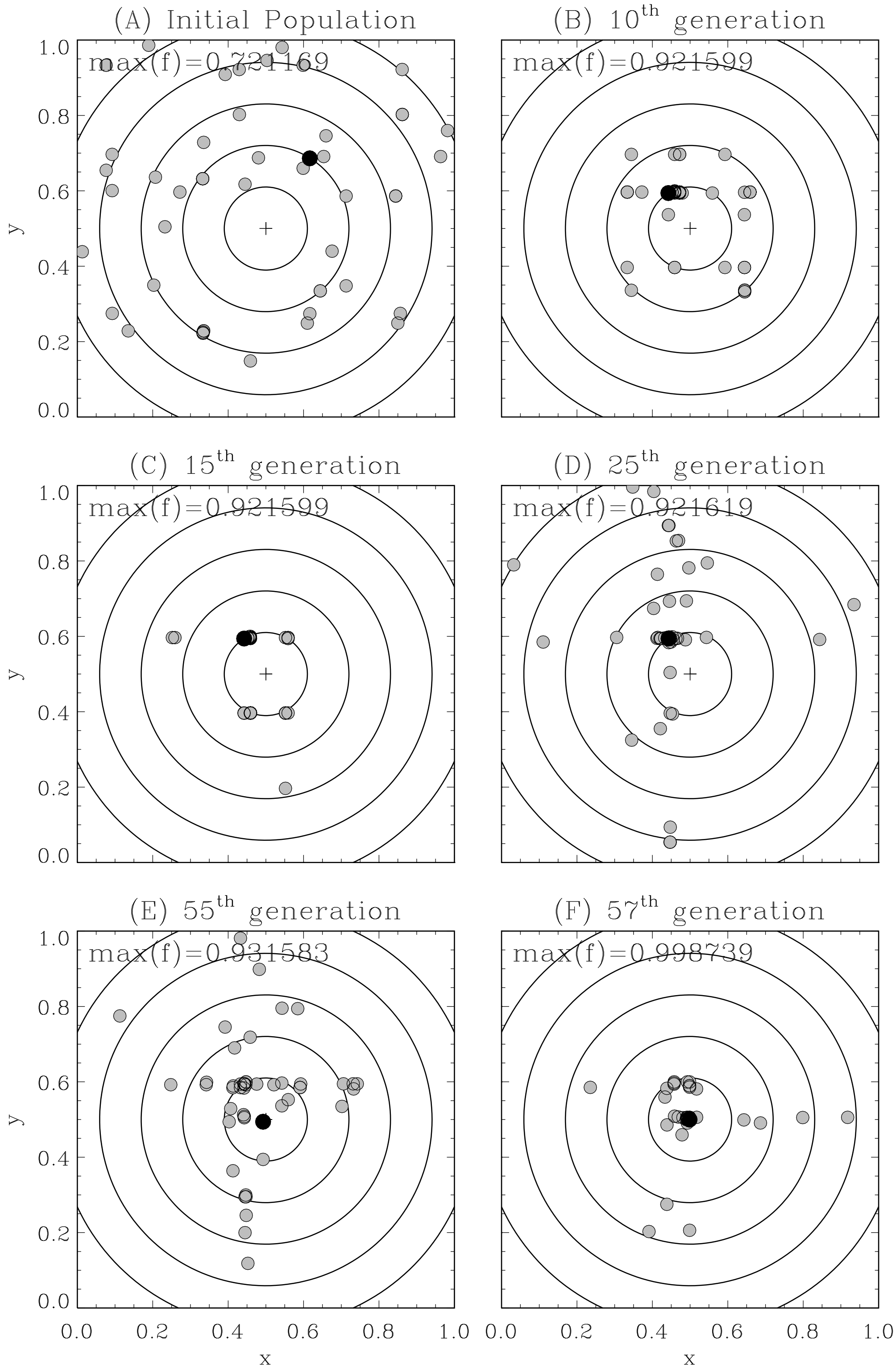

Figure 11: Evolution of the population of trial solutions in parameter space, for the GA2 run shown as a thicker line on Fig. 10. The concentric circles indicate the rings of secondary maxima, and the larger, solid black dot is the fittest solution of the current generation.

the current best via both crossover and mutation (Fig. 11[F]). Note how elitism is essential here, otherwise the "mutant" having landed on the slopes of the central peak would have a low likelihood of replicating itself intact into the subsequent generation, in view of the high mutation rate.

GA1 basically behaves in exactly the same way, with the important exception that many more generations are needed for the favorable mutation to show up; this is because GA1 operates with a fixed, low mutation rate, while GA2 lets this rate vary depending on the degree of convergence of the population (cf. §3.3.2).

3.5 Hamming walls and creep mutation

We are doing pretty well with GA2, but we still need to correct a fundamental shortcoming of the one-point mutation operator arising from the decimal encoding scheme of Fig. 9. Consider a problem where the sought-after optimal solution requires the following substring to be produced by the evolutionary process:

`..........21000..........;`

decoding into the floating point number 2.1000; now, early in the evolutionary run an individual having, say,

`..........19123..........`

will likely be fitter than average, and so this genetic material will spread throughout the population. After a while, following favorable mutations or crossover recombinations, the substring might look like, say

`..........19994..........`

which is admittedly quite close to `21000`. However, two very well coordinated mutations are needed to push this towards the target `21000`: the "`1`" must mutate to a "`2`" *and* the first "`9`" to a "`0`". Note that either mutation occurring in isolation, and/or mutating to a different digit value, takes us farther from the target floating point number. Mutation being a slow process, the probability of the needed pair of mutations occurring simultaneously will in general be quite small, meaning that the evolution would have to be pushed over many generations for it to happen. The population is getting "piled up" at internal boundaries of the encoding system.

These boundaries are called *Hamming walls*. They can be bypassed by choosing an encoding scheme such that successive single mutations can always lead to a continuous variation in the decoded parameter. This is why the so-called Gray binary coding (e.g., Press *et al.* 1992, §20.2) is now used almost universally in genetic algorithms based on binary encoding. Another possibility is to devise mutation operators that can jump over Hamming walls.

Creep mutation does precisely this. Once a digit on the encoding string has been targeted for mutation, instead of simply replacing the existing digit by a randomly chosen one, just add either $+1$ or -1 (with equal probability), and if the resulting digit is < 0 (because a "0" has been hit with "-1") or > 9 (because a "9" has been hit with "$+1$"), carry the one over to the next digit on the left. Just like in grade school. So, for example, creep mutation hitting the middle "9" with $+1$ in the last substring above would lead to

```
..........20094..........
```

which achieves the desired effect of "jumping" the wall.

The one thing creep mutation does *not* allow is to take large jumps in parameter space. As argued before, jumping is actually a needed capability; consequently, in practice for each offspring individual a probability test will decide whether one-point or creep mutation is to be used (with equal probabilities).

Creep mutation is *not* included in the original release of `PIKAIA` (now known as `PIKAIA 1.0`, although it is in version `1.2`, which has been released in April 2002 (see the `PIKAIA` Web Page and the *Release Notes for PIKAIA 1.2*, NCAR Technical Note 451-STR). The results described in what follows were obtained using a modified version of `PIKAIA 1.0`, GA3, which includes creep mutation but is otherwise identical to GA2.

3.6 Performance on test problems

Time now to turn loose our algorithms on the suite of test problems of §1.4. We have three versions of genetic algorithm-based optimizers : GA1, which represents a minimal algorithm, and GA2, which is identical to GA1 but includes in addition elitism and dynamic adjustment of the mutation rate; and GA3, including creep mutation but otherwise identical to GA2. As a comparison algorithm we retain iterated hill climbing using the simplex method, as described in §1.3. As will soon become evident, GA1 is actually not a very good optimizer, so that the more interesting comparison will be among GA2, GA3, and iterated simplex.

Before getting too carried away let's pause and reflect on what we are trying to achieve here. Ideally, one wants a method that achieves convergence to the global optimum with high probability ($p_G \gtrsim 0.95$, say), while requiring the smallest possible number of model (or function) evaluations in doing so. This latter point can become a dominant constraint when dealing with a real application, where evaluating the "fitness" of a given trial solution is computationally intensive[18].

---

[18] Consider the helioseismic inversions described in Charbonneau *et al.* (1998) using a genetic algorithm; given a set of parameters defining a trial solution, fitness

Such considerations are easily quantified. Let N_p and N_g be the population size and generation length of a run; the required number of function evaluations, N_f, is obviously

$$N_f = N_p \times N_g \ , \qquad [\text{GA1, GA2, GA3}] \qquad (13a)$$

while for iterated simplex N_f is the number of hill climbing trials (N_t) times the average number of function evaluations required by a single simplex run (N_s; this quantity is run- and problem-dependent):

$$N_f = N_t \times N_s \ . \qquad [\text{Iterated simplex}] \qquad (13b)$$

So we play the following game: we run iterated simplex and GA2 for increasing numbers of generations/iterations, and check whether global convergence is achieved; to get statistically meaningful results we do this 1000 times for each method and each generation/iteration count. This allows us to empirically establish the probability of global convergence (p_G, $\in [0, 1]$) as a function of generation/iteration count. In doing so, to decide whether or not a given run has globally converged we use again the criteria $f \geq 0.95$ for P1, P2 and P3, and $R \leq 0.1$ for P4. The results of this procedure, applied to each test problem, is shown in Figure 12.

It should be easy to convince yourself of the following: (1) on P1 and P2, both iterated simplex and GA2 perform equally well on all aspects of performance when pushed long enough to have $p_G \gtrsim 0.9$; (2) P3 is a hard problem, and neither technique performs satisfactorily on it. Still, GA2 largely outperforms iterated simplex on global performance. (4) On P4, GA2 and iterated simplex do equally well up to $p_G \sim 0.5$, but then GA2's performance starts to lag behind as the solutions are pushed to $p_G \gtrsim 0.95$.

An obvious conclusion to be drawn at this juncture is that iterated hill climbing using the simplex method makes for a pretty decent global optimization scheme. Not quite what you were expecting as a sales pitch for genetic algorithm-based optimization, right? This is in part a consequence of the relatively low dimensionality of our test problems. Recall from §1.5 that iterated simplex leads to improved performance (with respect to single run simplex) primarily as a consequence of the better sampling of parameter space associated with the initial (random) distribution of simplex vertices; given enough trials, one is almost guaranteed to have one initial simplex vertex landing close enough to the

---

evaluation involves (1) the construction of a 2-D rotation curve, (2) a large matrix-vector multiplication, (3) the calculation of a χ^2 against some 600 data points. This adds up to about half a CPU-second on a Cray J90. All test problems of §1.4 require very little computation in comparison.

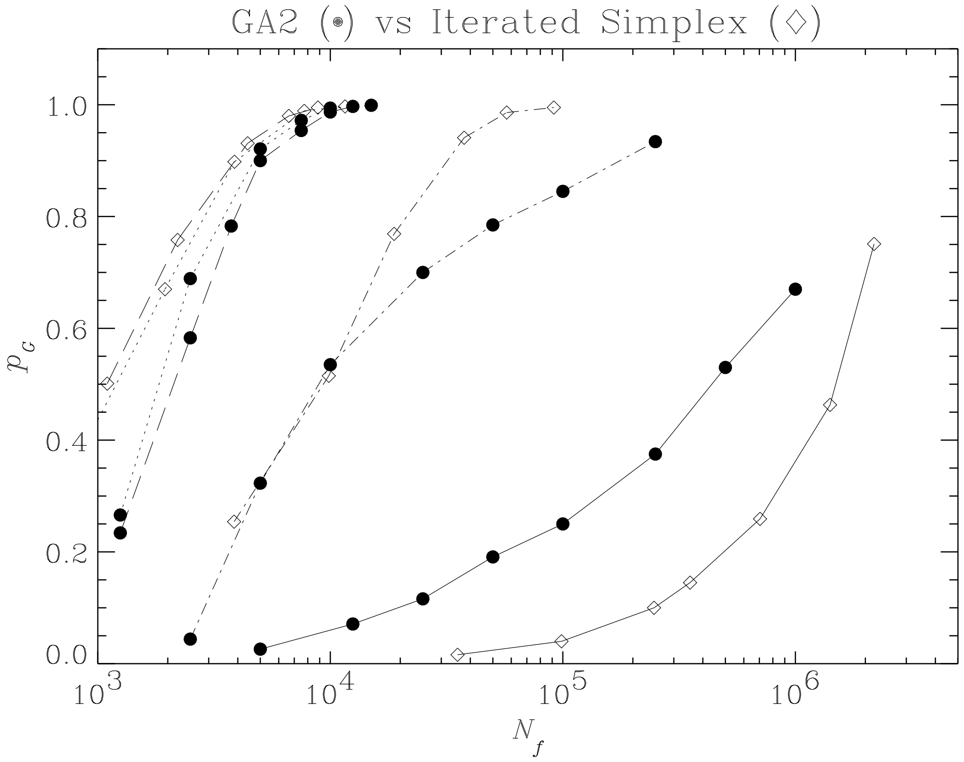

Figure 12: Global convergence probability as a function of the number of function evaluations N_f required by iterated simplex (diamonds) and GA2 (solid dots) on the four test problem (P1: dotted line; P2: dashed line; P3: solid line; P4: dash-dotted line). The probabilities were estimated from 1000 distinct trials and, in the case of iterated simplex, N_f is an average over the 1000 trials.

global maximum to ensure subsequent global convergence. In low dimensional search spaces, iterated simplex thus ends up being quite competitive. Figure 12 already indicates that this "edge" does not carry over to higher dimensionality (compare results for P1 and P3).

GA2's performance on P2 is actually a delicate matter. Take another look at Figure 5 and consider what happens once the population has converged to the broad, secondary maximum (as it does early in the run for nearly every single trial); for mutation to propel a solution from $(x, y) \simeq (0.5, 0.5)$ to the narrow peak $(x, y) = (0.6, 0.1)$, two very well coordinated mutations must take place simultaneously, otherwise mutant solutions end up in regions of rather low elevations and do not contribute much to the next generation. This is a low probability occurrence even at relatively high mutation rates[19], so the process takes time. GA2's global performance on P2 then results from an interplay between one-point

[19] Notice on Figure 11(D)—(F) how few solutions show up in the corners of the

mutation and the rather direct relationship that exists here between a solution's defining parameters and its string representation, on which mutation and crossover operate. If the narrow Gaussian is centered on $(x, y) = (0.5, 0.9)$, then a single mutation can propel a solution from the broad, central Gaussian to the narrow one. Not surprisingly, on this modified problem GA2 outperforms iterated simplex to a significant degree: $p_G = 0.987$ with only $N_f = 2500$, i.e., faster than iterated simplex by a factor of 5. Encoding is a tricky business, with potentially far-reaching consequences on performance.

Iterated simplex's superior performance on P4 is certainly noteworthy, yet reflects in part the peculiar structure of parameter space defined by the Gaussian fitting problem, which is relatively well accommodated by the simplex method's pseudo-global capabilities. Other local optimization methods do not fare nearly as well. For a detailed comparison of genetic algorithms and other methods on fitting Gaussian profiles to real and synthetic data, see McIntosh *et al.* (1998).

It is really only with very hard problems, such as P3, that GA2 starts showing its worth. By any standards, P3 is a very hard global optimization problem. While on its 2-D version GA2 and iterated simplex do about as well, as dimensionality is increased the global performance of iterated simplex degrades much more rapidly than GA2. This is in fact where the power of genetic algorithm-based optimizers lies, although for search spaces of high dimensionality ($n \gtrsim 20$, say) the one-point crossover and mutation operators described in §2.3 are usually suboptimal and must be improved upon[20].

In some sense, a fairer comparison of the respective exploratory capabilities of GA2 and iterated simplex can be carried out by setting the number of trials in iterated simplex so that the original distribution of simplex vertices samples parameter space with the same density as GA2's initial random population; in other words, using the notation of eqs. (13), we set

$$N_t = N_p/(n+1) \ , \tag{14}$$

where n is the dimensionality of parameter space, and compare the results of the resulting iterated simplex runs to some "standard" GA2 and GA3 runs. Such a comparison is presented in Table II, in a format essentially identical to Table I. Performance measures are also listed for a set of GA1 runs extending over the

---

domain.

[20] `PIKAIA 1.2`, to be released in April 2002, includes a two-point crossover operator, which generally improves performance for problems involving many parameters. See, e.g., Section 3 of the *Release Notes for PIKAIA 1.2* (Charbonneau 2002).

Table II

Performance on test problems (with eq. (14) enforced)

Test Problem	Performance	Iter. Simplex	GA1	GA2	GA3
P1	$\langle 1-f \rangle$	0.05471	0.0753	0.0071	0.0024
	p_G	**0.302**	**0.119**	**0.914**	**0.971**
	$\langle N_f \rangle$	671	5000	5000	5000
	N_t or N_g	17	100	100	100
P2	$\langle 1-f \rangle$	0.11583	0.1961	0.0212	0.0289
	p_G	**0.353**	**0.018**	**0.883**	**0.840**
	$\langle N_f \rangle$	745	5000	5000	5000
	N_t or N_g	17	100	100	100
P3	$\langle 1-f \rangle$	0.15547	0.166	0.0634	0.0614
	p_G	<**0.0001**	**0.001**	**0.191**	**0.230**
	$\langle N_f \rangle$	727	25000	50000	50000
	N_t or N_g	10	500	1000	1000
P4	$\langle R \rangle$	0.0619	0.199	0.039	0.076
	p_G	**0.619**	**0.149**	**0.845**	**0.698**
	$\langle N_f \rangle$	5593	50000	50000	50000
	N_t or N_g	7	1000	1000	1000

same number of generations as the GA2 and GA3 runs. Once again performance measures are established on the basis of 1000 distinct runs for each method.

Evidently GA2 and GA3 outperform GA1 on all aspects of performance to a staggering degree. GA1 is not much of a global numerical optimization algorithm. Comparison with Table I shows that its global performance exceeds somewhat that of the simplex method in single-run mode, but the number of function evaluations required by GA1 to achieve this is orders of magnitude larger.

What is also plainly evident on Table II is the degree to which GA2 and GA3 outperform iterated simplex *for a given level of initial sampling of parameter space.* Although the number of function evaluations required is typically an order of magnitude larger, both algorithms are far better than iterated simplex at actively exploring parameter space. *This is plain evidence for the positive effects of transfer of information between trial solutions in the course of the search process.*

The worth of creep mutation can be ascertained by comparing the global performance of the GA2 and GA3 solutions. The results are not clear-cut: GA3 does better than GA2 on P1 and P3, a little worse on P2, and significantly worse on P4. The usefulness of creep mutation is contingent on there actually being Hamming walls in the vicinity of the global solution; if there are, creep mutation helps, sometimes quite a bit. Otherwise, it effectively decreases the probability of taking large jumps in parameter space, and so can be deleterious in some cases. This is what is happening here with P2, where moving away from the secondary maximum requires a large jump in parameter space to take place, from $(x, y) = (0.5, 0.5)$ to $(0.6, 0.1)$.

At any rate, the above discussion amply illustrates the degree to which *global performance is problem-dependent*. This cannot be overemphasized. You should certainly beware of any empirical comparisons between various global optimization methods that rely on a small set of test problems, especially of low dimensionality. You should also keep in mind that GA2 is one specific instance of a genetic algorithm-based optimizer, and that other incarnations may behave differently —either better or worse— on the same test problems.

The least you should remember from Section 3:

- Through random initialization of the population, genetic algorithms introduce no initial bias whatsoever in the search process.
- For numerical optimization, elitism and an adjustable mutation rate are two crucial additions to a basic genetic algorithm.
- Iterated hill climbing using the simplex method makes a pretty decent global optimization technique, especially for low-dimensionality problems.
- Performance measures of any global optimization method are highly problem-dependent.

Exercises for Section 3:

(1) Look back at Figure 10. The dynamically adjusting mutation rate levels off at a value of about 0.1. One could have predicted this average value before running the code. How? (Hint: re-read §2.2)

(2) Code up a 3-D, 5-D and 6-D version of P1. Using `PIKAIA` in its GA2 form (default settings except for generation count), investigate how global performance degrades with problem dimensionality. Keep the generation count fixed at 2500 (`ctrl(2)=2500`). How does this compare to iterated simplex?

---

4. A REAL APPLICATION: ORBITAL ELEMENTS OF BINARY STARS

4.1 Binary stars

More than half of all stars observed in the solar neighborhood are components of binary systems. This is presumed to be a consequence of angular momentum conservation leading to fragmentation in the later stages of collapse of protostellar clouds. Some binary stars can be resolved optically with even a small telescope; the first such *visual binary* system was discovered in 1650 by G.B. Riccioli. Binarity can also be established spectroscopically, by measuring the small Doppler shift in narrow spectral lines caused by the component V along the line-of-sight of the orbital velocity about the system's center of mass. For non-relativistic orbital speeds the wavelength shift is

$$\frac{\Delta\lambda}{\lambda} \simeq \frac{V}{c} \ , \tag{15}$$

where c is the speed of light. The first such *spectroscopic binary* was discovered in 1889 by E.C. Pickering[21]. Current hardware and analysis techniques now allow us to measure stellar radial velocity with useful accuracy down to 2—3 meters per second[22]. This level of accuracy is what has made possible the recent spectacular discovery of extrasolar planets. Figure 13 shows radial velocity measurements of the star η Bootis, a "classical" spectroscopic binary star. From these data one can determine the orbital parameters of the system.

[21] Amusingly, this spectroscopic binary is the brighter component of the first visual binary discovered by Riccioli: the star Mizar A, in the constellation *Ursa Majoris*. Even better, it was later realized that Mizar B is also a spectroscopic binary.

[22] This figure is for 2002; back in 1998, when this paper was originally written, it was given as $10\,\mathrm{m\ s}^{-1}$. Pretty remarkable improvement, in just a little over three years...

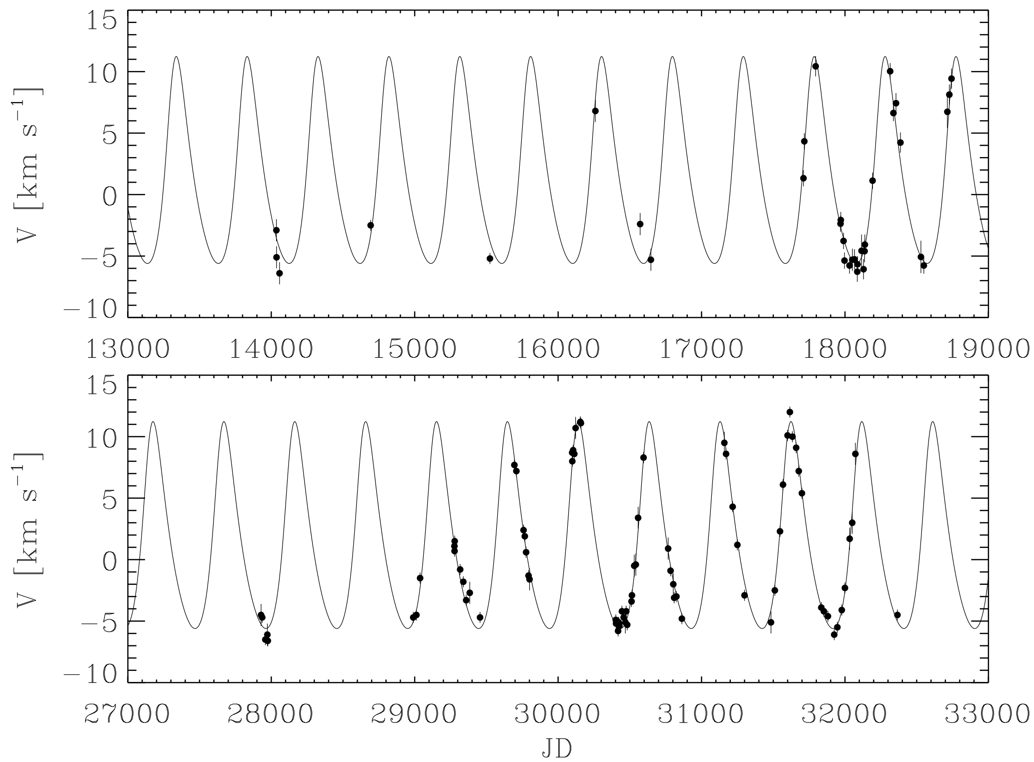

Figure 13: Radial velocity variation observed in the spectroscopically visible component of the binary star η Bootis. The time axis is given in units of Julian Date (one JD = one solar day). Data are from Bertiau (1957), with one lone datum at JD= 23175 missing on this plot. The solid line is the best-fit solution obtained later in this section. The asymmetrical shape of the curve is due to the eccentricity of the orbit; a circular orbit would lead to a purely sinusoidal radial velocity variation.

4.2 Radial velocities and Keplerian orbits

If the shape and size of an orbit are known, as well as the orientation of its semi-major axis with respect to the line of sight, the expected radial velocity variations can be computed and compared to observations. There are actually a few subtleties involved. Determining the radial velocity variation associated with the motion of a binary component in an arbitrarily positioned elliptical orbit about the common center of mass of the system is a straightforward but somewhat messy problem in spherical trigonometry. The procedure is laid out in great gory details in many astronomical monographs (see, e.g., Smart 1971). We shall simply write down the resulting expression here:

$$V(t) = V_0 + K(\cos(\omega + v(t)) + e\cos(\omega)) \ . \tag{16}$$

The quantity V_0 is the radial velocity of the binary system's center of mass, and eq. (16) only holds once the Earths' orbital motion about the Sun as been subtracted out. Note that the quantity V_0 *cannot* be simply "read off" the radial velocity curve, unless the orbit is perfectly circular. The velocity amplitude K is a function of other orbital parameters:

$$K = \frac{2\pi}{P} \frac{a \sin i}{(1 - e^2)^{1/2}} , \tag{17}$$

where P is the orbital period, e the orbital eccentricity, a the semi-major orbital axis, and i the inclination angle of the orbital plane with respect to the plane of the sky ($i = 0$ is an orbit in the plane of the sky, $i = 90°$ an orbit seen edge-on). Because i usually cannot be inferred from the velocity curve (unless the system happens to also be an eclipsing binary), the velocity amplitude K is usually treated as a single parameter. The so-called true anomaly v is the time-like variable, and corresponds to the angle between a radius vector joining the star to the center-of-mass and that joining the orbital perihelion to the center-of-mass. The longitude of the perihelion (ω) is the angle subtended by the latter line segment to the line segment defined by the intersection of the orbital plane with the plane of the sky (see Smart 1971, §195 and Figure 132).

The chief complication arises from the fact that for an elliptical orbit the angular velocity about the center of mass is not constant, but obeys instead Kepler's second Law (orbital radius vector sweeps equal areas in equal time intervals). The true anomaly v is related to the so-called eccentric anomaly E via the relation

$$\tan \frac{v}{2} = \sqrt{\frac{1+e}{1-e}} \tan \frac{E}{2} . \tag{18}$$

The eccentric anomaly, in turn, is related to time via Kepler's equation (see Smart 1971, §68):

$$E - e \sin(E) = \frac{2\pi}{P}(t - \tau) , \tag{19}$$

where τ is the time of perihelion passage, by convention the zero-point of the orbit. At the risk of oversimplifying, in essence what E measures is the deviation from constant angular velocity (as in a circular orbit) due to the orbit's eccentricity. Note that eq. (19) is transcendental in E, i.e., it cannot be solved analytically for E as a function of t. Of course it can be solved using any of the classical methods for nonlinear root finding, such as bisection (Press et al. 1992, §9.1). Kepler, of course, did it all by hand.

Going over the preceding expressions, one can identify 6 parameters that need to be determined to relate the radial velocity curve to the orbital elements and related quantities. These six parameters are:

(1) P, the orbital period;

(2) τ, the time of perihelion passage;

(3) ω, the longitude of the perihelion;

(4) e, the orbital eccentricity;

(5) K, the orbital velocity amplitude;

(6) V_0, the system's radial velocity.

For later use we will group these six parameters into a vector

$$\mathbf{u} = (P, \tau, \omega, e, K, V_0) \ . \tag{20}$$

A number of methods have been devised to infer these parameters from a given radial velocity curve (see, e.g., Smart 1971, §197; Petrie 1962). In what follows we treat this fitting problem as a nonlinear least-squares minimization problem. We seek to find the parameter set $\mathbf{u}$ that minimize the reduced χ^2 given N data points $V_j^{\rm obs} \equiv V(t_j)$ with associated error estimates σ_j:

$$\chi^2(\mathbf{u}) = \frac{1}{N-6} \sum_{j=0}^{N-1} \left(\frac{V_j^{\rm obs} - V(t_j;\mathbf{u})}{\sigma_j} \right)^2 \ , \tag{21}$$

where the normalization factor $(N-6)^{-1}$ is the number of degrees of freedoms of the fit; under this normalization $\chi^2 \lesssim 1$ indicates an acceptable fit.

The minimization problem defined by eq. (21) can —and will— be solved using a genetic algorithm-based optimizer, specifically GA3's version of `PIKAIA`. At this stage you should perhaps only note that performing this minimization using an explicitly gradient-based method would be a real mess. If you need to be convinced of this try differentiating eq. (16) with respect to e, and see how you like it... This, however, is not the only difficulty one encounters in carrying out the fit. The most serious problem is related to the existence of solution degeneracies, i.e., widely differing sets of fitting parameters that lead to very similar radial velocity curves, for some classes of orbits. This can become a severe problem for noisy and/or poorly sampled radial velocity curves; the search space then becomes markedly multimodal in χ^2, and a global method is essential. The η Bootis data of Fig. 13 offers a moderately difficult global optimization problem. So let's give it a go using `PIKAIA`.

4.3 A genetic algorithm solution using `PIKAIA`

4.3.1 Normalization of input parameters

Since `PIKAIA` operates internally in a search space bound in $[0, 1]$ in all dimensions, the first thing to do is to normalize the components of the parameter vector $\mathbf{u}$. Reasonable choices would be:

P	$\equiv \mathbf{u}(1)$	$200 \leq P \leq 800$	[in JD; from data]
τ	$\equiv \mathbf{u}(2)$	$t_0 \leq \tau \leq t_0 + P$	[in JD]
ω	$\equiv \mathbf{u}(3)$	$0 \leq \omega \leq 2\pi$	[radian; full allowed range]
e	$\equiv \mathbf{u}(4)$	$0 \leq e \leq 1$	[dimensionless; full allowed range]
K	$\equiv \mathbf{u}(5)$	$0 \leq K \leq \max(V_j^*) - \min(V_J^*)$	[$\mathrm{km\,s^{-1}}$; from data]
V_0	$\equiv \mathbf{u}(6)$	$\min(V_j^*) \leq V_0 \leq \max(V_j^*)$	[$\mathrm{km\,s^{-1}}$; from data]

You might think that some of these bounds could be significantly reduced by even casual examination of the data (P, for example); while this might well be the case and would likely accelerate convergence, it is definitely *not* recommended practice in general. One of the great advantage of genetic algorithms is the complete lack of initial bias they introduce in the search process, in view of the fully random nature of the initial population (cf. §3.1). The last thing you want to do is reintroduce a bias by overly constraining the parameter ranges to be explored. Whenever firm physical or mathematical bounds exist, use them. This is what is done here. The eccentricity e and perihelion longitude ω are both by definition bound to $[0, 1]$ and $[0, 2\pi]$, respectively. For *a priori* unbounded variables, use the data to establish bounds that are as wide as reasonable, while remaining physically meaningful. For example, in view of eq. (16) it should be clear that V_0 must be bound between the smallest and largest velocity values occurring in the dataset, and that K cannot exceed the data's peak-to-peak spread (although it might be wise to extend these bounds by some small amount if the radial velocity curve is very noisy or poorly sampled by the data.). Note that the encoding process allows us to use some parameter values to set meaningful bounds on other parameters; this is done here for the time of perihelion passage τ, which cannot exceed a time interval equal to the orbital period.

4.3.2 Defining fitness

Fitness is the only point of contact between the genetic algorithm and the problem being solved. Because time is determined much more accurately than observed radial velocities, the natural quantity to use here is the usual χ^2 measure of goodness-of-fit (cf. eq. [21]), though of course other statistical estimators can be used.

Given a trial solution, as defined by a 6-vector $\mathbf{u}$, computing a χ^2 requires the construction of a synthetic radial velocity curve evaluated at the N data abcissa t_j. Once the input parameters have been properly rescaled (§4.3.1), for each of the t_j's, the steps involved are:

(1) Given a t_j and trial period P, eccentricity e, and time of perihelion passage τ, solve Kepler's equation (19) for E. This defines a nonlinear root finding problem for which the bisection method is well-suited;

(2) Now knowing E, calculate the true anomaly v using equation (18);

(3) Now knowing v, and given the trial velocity amplitude K, system velocity V_0 and perihelion longitude ω, compute the radial velocity V using equation (16);

(4) Once V has been computed for all t_j, calculate χ^2 using equation (21).

One final step is required, to relate χ^2 to fitness, in order to set the selection probability of the trial solution. `PIKAIA` is set up to *maximize* the user-defined function, and *requires* fitness to be a positive definite quantity. We thus set

$$\text{Fitness} = (\chi^2)^{-1} . \tag{22}$$

Because `PIKAIA` uses ranking to set selection probability, you need not worry about the functional form you impose between fitness and χ^2; making fitness proportional to $(\chi^2)^{-1/2}$ (say) would lead to the same rank distribution, and so to the same selection probabilities. Naturally you'd better make sure that the relationship you define between fitness and goodness-of-fit is single-valued and monotonic. Otherwise you can't expect `PIKAIA` to produce anything sensible. Please do not set fitness equal to $-\chi^2$, as `PIKAIA`'s implementation of the Roulette Wheel Algorithm for parent selection *requires* fitness to be a positive-definite quantity. This has been a common initial mistake for `PIKAIA` users attempting χ^2 minimization.

4.3.3 Setting `PIKAIA`'s internal parameters

Unless you have good reasons to do otherwise, use `PIKAIA`'s default parameter settings. This is done by initializing all twelve elements of the control vector `ctrl` to some nonzero negative value, and will result in what we have been calling GA2. The one parameter you most likely want to set explicitly is the generation count N_g. This corresponds to the second element of the control vector, so that for example setting `ctrl(2)` = `2000` would force `PIKAIA` to run for 2000 generations, instead of its default value of 500. As you hopefully have figured out by now, the required number of generations is very much problem-dependent. Just be sure to remember the Dirty Harry Rule; if you're not sure how to set N_g, err on the high side.

4.3.4 Running `PIKAIA`

The first thing to do is to write a FORTRAN function that is given a trial solution parameter vector **u**, and returns a fitness. This is really the only interface between `PIKAIA` and the problem at hand. The argument specification of the function are hardwired into `PIKAIA`, so that the beginning of the function *must* look like

```
real function orbit(n,x)
dimension x(n)
```

where `n` is the dimension of parameter space and `x(n)` is a vector of `n` floating point numbers defining a trial solution. Of course the function's name, here `orbit`, can be whatever you like, but do declare it `external` in the calling program. For the orbital element fitting problem we have `n`= 6. The function itself basically goes through the sequence of steps listed in §4.3.2 to compute a χ^2, which is then used to define a fitness as per eq. (22).

One important thing relates to the scaling of the input parameters. The scaled versions of the `x(n)`'s (cf. §4.3.1) must be stored in new variables local to the fitness function. Storing the rescaled parameters back into the `x(n)`'s, i.e.,

```
x(1)=200.+x(1)*600.      [*****NEVER DO THIS*****]
```

is guaranteed to have disastrous consequences (besides being poor programming style). This has also been a relatively common initial mistake of `PIKAIA` users so far.

Prior to calling `PIKAIA` itself three things need to be done: (1) Read the time and radial velocity data and making them accessible to the fitness function through an appropriately defined `COMMON` block (for example). (2) initialize the random number generator[23], and (3) initialize `PIKAIA`'s control vector. The last two steps are carried out as follows:

```
seed=123456
call urand_init(seed)
do i=1,12
   ctrl(i)=-1
enddo
```

You can of course pick any seed value other than 123456, as long as it is a nonzero positive integer. Initializing all components of `ctrl` to some negative value, as done here, forces `PIKAIA` to use its internal default settings. This yields GA2,

[23] `PIKAIA` is distributed with a random number generator which is *deterministic*, meaning that it must be given a seed value, from which it will always produce the same sequence of random deviates (on a given platform).

evolving a population of $N_p = 100$ individuals over $N_g = 500$ generations. For other possible settings (and their algorithmic consequences) see §4.5 of the PUG. With the fitness function defined as described above, a call to `PIKAIA` looks like

```
call pikaia(orbit,n,ctrl,xb,fb,status)
```

Upon successful termination (indicated by `status`$=0$ on output), the `n`-dimensional array `xb` contains the parameters (scaled to $[0,1]$) of the best trial solution of the last generation, and its fitness is given by the scalar output variable `fb`. By default this is the only output returned by `PIKAIA`, although additional run-time output can be produced by appropriately setting `ctrl(12)` to 1 or 2 (see PUG, §4.5).

4.3.5 Results

Figure 14 shows results for a typical GA3 run. Part (A) shows convergence curves, namely the χ^2 value for the best (solid line) and median (dashed line) individual as a function of generation count. Part (B) shows the corresponding variations of the six parameters defining the best solution (scaled to $[0,1]$). Most parameters undergo rapid variations over the first ten generational iterations, but subsequently tend to remain pretty stable until favorable mutations or crossover produce better individuals. The first such "key" mutation occurs at generation 184, when a period close enough to the true period is finally produced. Note how the solution then remains "stuck" on a $e=0$ secondary minimum up to generation 430. The subsequent evolution is characterized by a gradual increase in e, ω and τ, accompanied by smaller adjustments in K and V_0. Notice again on Fig. 14(A) how, especially in the first few hundred generations, the mutation rate (dotted line) is highest when the best solution is "stable", and decreases again following significant changes in best fitness.

The final, best solution vector after 1000 generations is

$$(P,\tau,\omega,e,K,V_0) = (494.20, 14299.0, 326.86, 0.26260, 8.3836, 1.0026)\ , \qquad (23)$$

with units as in §4.3.1. This best-fit GA3 solution, with $\chi^2 = 1.63$, is plotted as a solid line on Fig. 13. It turns out that for this problem the use of creep mutation is advantageous, as detailed in the following section. The best-fit solution differs slightly from the best-fit solution of Bertiau (1957), but lies well within Bertiau's one-σ range. The fact that the solution has a χ^2 significantly larger than 1 should not be deemed extremely alarming, as error estimates on V given in Bertiau (1957) are based in part on a subjective assessment of the "quality" of his photographic plates.

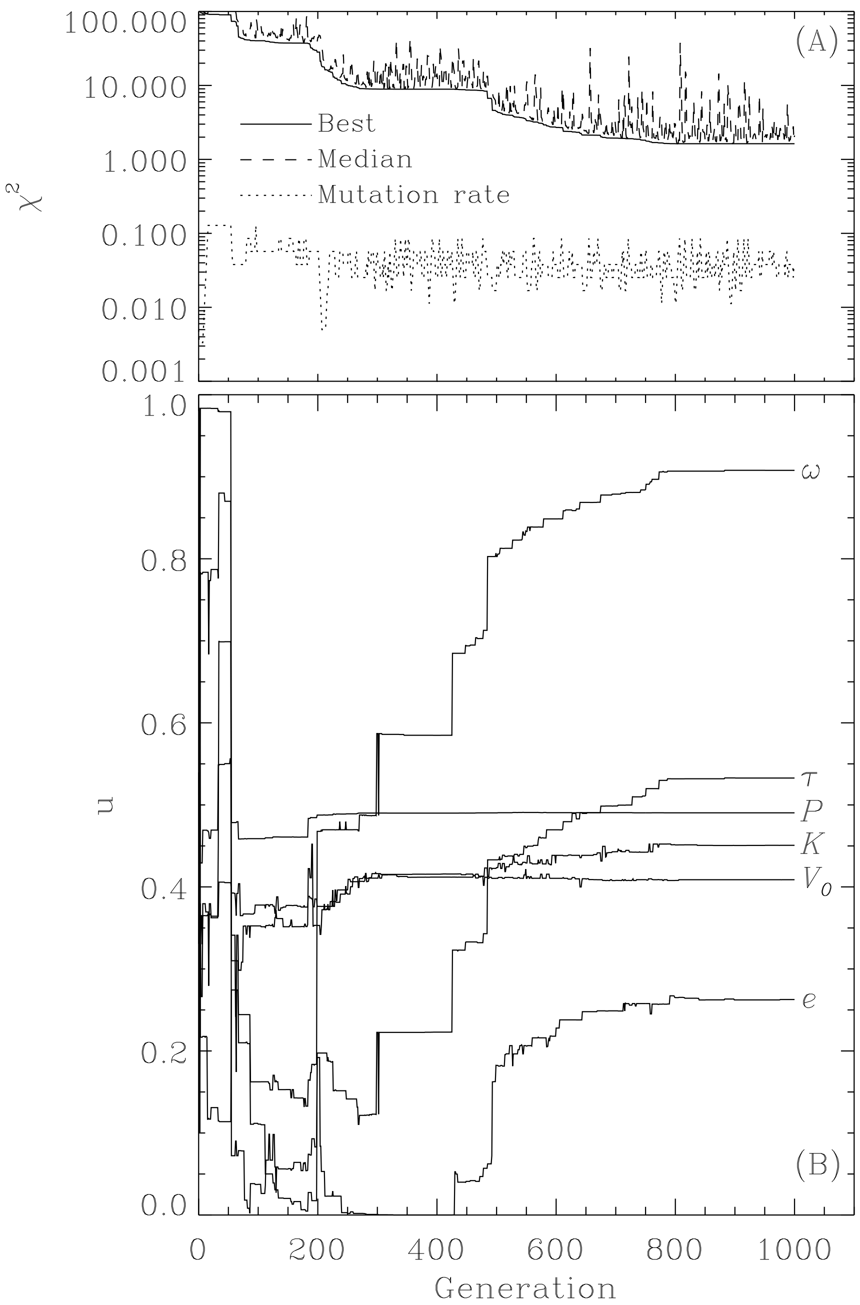

Figure 14: Evolution of a typical solution to the binary orbit fitting problem. Part (A) shows the χ^2 (inversely proportional to fitness) of the best and median individuals in the population, as well as the mutation rate (dotted line). Part (B) shows the corresponding variations of the six parameters defining the best individual (scaled to $[0, 1]$).

4.3.6 Error estimates

You might think we're done, but we certainly are not; our allegedly global solution of §4.3.5 is almost worthless until we can specify error bars of some sort on the best fit model parameters.

The traditional way derivative-based local hill climbing methods compute error bars "automatically" is via the Hessian matrix of second derivatives evaluated at the best-fit solution (e.g., Press *et al.* 1992, §15.5). This local curvature information is not particularly useful when dealing with a global optimization problem. What we want is some information about the shape and extent in parameter space of the region where solutions with $\chi^2 \leq 1$ are to be found. This is usually done by Monte Carlo simulation, by perturbing the best fit solution and computing the χ^2 of these perturbed solutions (see, e.g., Bevington & Robinson 1992, §11.6; Press *et al.* 1992, §15.6). This is undoubtedly the most reliable way to get error estimates. All it requires is the ability to compute a χ^2 given a set of (perturbed) model parameters; if you have found your best-fit solution using a genetic algorithm-based optimizer such as `PIKAIA`, you already have available the required computational machinery: it is nothing else than your fitness function.

In relatively low-dimensionality parameter spaces such as for our orbital fitting problem, it is often even simpler to just construct a hypercube centered about the best-fit solution and directly compute $\chi^2(\mathbf{u})$ at some preset spatial resolution across the cube. Figure 15 shows the result of such an exercise, in the context of our orbital fitting problem. The Figure shows χ^2 isocontours, with the best-fit solution of §4.3.5 indicated by a solid dot. A strong well-defined error correlation between ω and τ is seen on panel (D). This "valley" in parameter space becomes longer and flatter as orbital eccentricities approach zero. The gradual, parallel increase in ω and τ visible on Fig. 14(B) corresponds to the population slowly "crawling" along the valley floor; this process is greatly facilitated by the use of creep mutation. Weaker error correlations are also apparent between e, V_0 and K.

The diamonds are a series of best-fit solutions returned by a series of GA2 runs extending over 2500 generations. Only the runs having returned a solution with $\chi^2 \leq 1.715$ are shown. The dashed lines are the means of the inferred parameter values. Notice, on Panels (A) and (B), the "pileup" of solutions at $K = 8.3696$, and a similar such accumulation at $\omega = 324$ on panel (D). These parameter values map onto Hamming walls in the scaled $[0, 1]$ parameter range used internally by `PIKAIA`. It just so happens that for these data and adopted parameter range, two such walls lie close to the best-fit solution; this does not prevent GA2 from converging, but it slows it down significantly; in this case the solutions stuck at walls still lie well within the 68.3% confidence level, but GA2 needs to be pushed to a few thousands of generations to reliably locate the true χ^2 minimum. Since

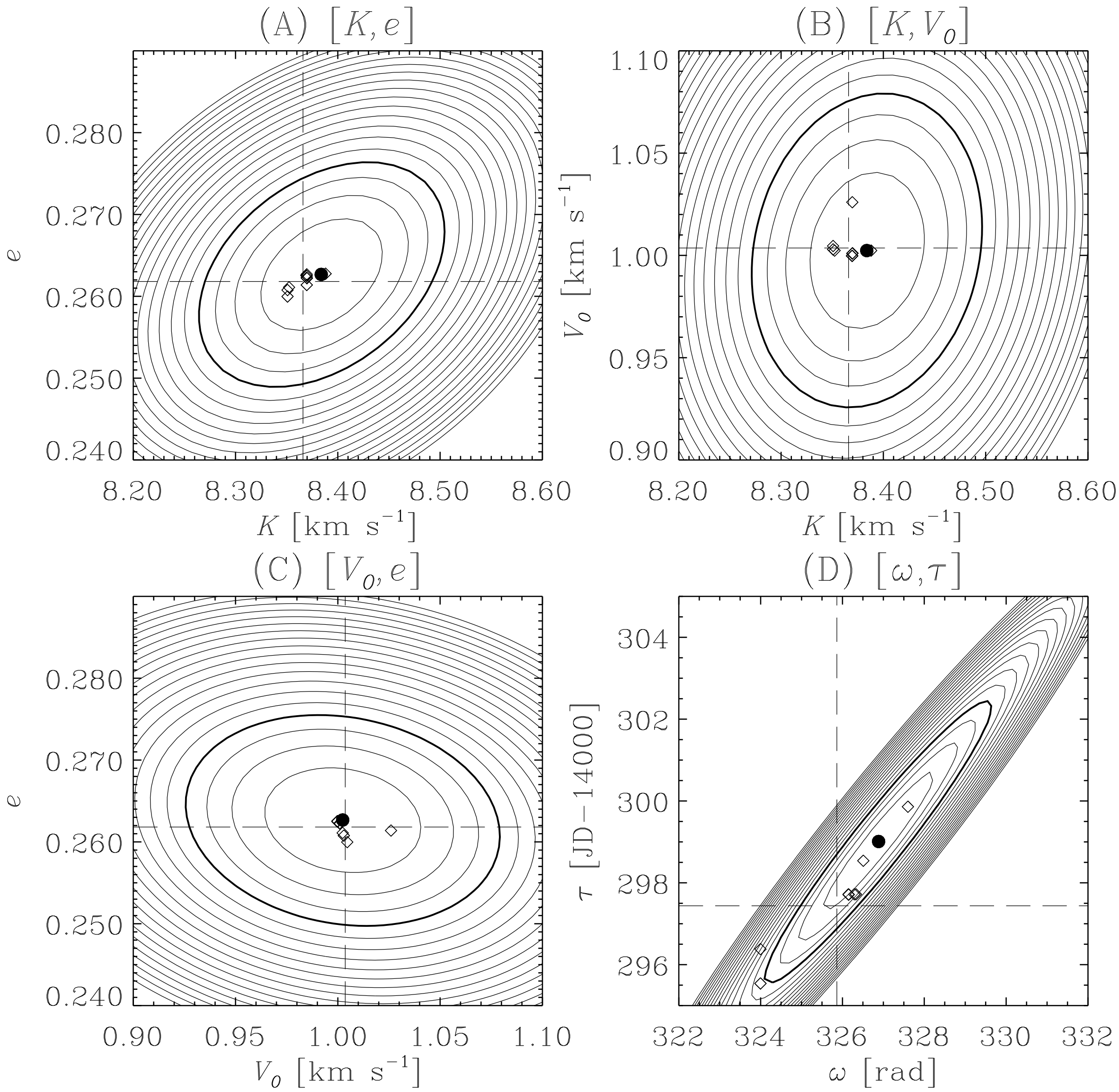

Figure 15: χ^2 isocontours in four hyperplanes of parameter space. Contour spacing is $\Delta\chi^2 = 0.0053$, and the thicker contours corresponds to the 68.3% confidence interval for 2-parameter joint probability distribution ($\chi^2 = 1.6513$; conceptually similar to a one-σ region, see Press *et al.* 1992, §15.6). The solid dots mark the best-fit solution of §4.3.5. Note the clear error correlation between ω and τ on panel (D). The diamonds are best-fit solutions returned by a series of GA2 runs. The dashed lines indicate the corresponding mean parameter values.

most of the GA2 solutions would be deemed acceptable on the basis of their χ^2 value, you might have missed this (I certainly did at first) unless you chose, as you should, to heed the Dirty Harry Rule and err on the high side in setting the generation count N_g. So here is a case where the use of creep mutation is definitely advantageous.

The error-estimation procedure described above is straightforward though (CPU) time-consuming. It certainly works if the computing costs associated with evaluating one model are sufficiently low to allow the calculation of many additional solutions once the genetic algorithm has converged. If this is not the case other strategies must be used. One possibility is to accumulate information about χ^2 isosurfaces *in the course of a single evolutionary run.* After all, while the population evolves through many hundreds (or thousands) of generations a significant (but non-homogeneous) sampling of parameter space is taking place. Throughout the evolutionary run, one stores away all solutions that fall below whichever χ^2 value is deemed to indicate an adequate fit. This information is then used *a posteriori* to construct χ^2-isosurfaces, without having to carry out additional model evaluations. Gibson & Charbonneau (1998) describe one such technique, in the context of a coronal modeling problem. We actually had to take steps to *slow down* the convergence of our genetic algorithm, in order to achieve a suitable sampling of parameter space in the course of the evolutionary run (see the Gibson & Charbonneau paper for more details).

One thing you should definitely *not* do is use your population of trial solutions at the *last* generational iteration to establish error bounds; while the final population will evidently be distributed about the best-fit solution if the algorithm has converged, the way population members are distributed in parameter space is greatly influenced by features of the genetic algorithm, notably the value of the mutation over the last few generations. Such extraneous factors are clearly unrelated to the structure of parameter space in the vicinity of the best-fit solution.

The least you should remember from Section 4:

- Set upper and lower bounds on allowed parameter values that are as wide as possible, while remaining physically meaningful.
- When using `PIKAIA`, always make your fitness a positive-definite quantity; in your fitness function, always store your rescaled input parameters in local variables, rather than back into the input parameter vector `x(n)`.
- Whenever in doubt as to the number of generations through which you should let your solutions evolve, err on the high side.
- With global optimization, *a posteriori* Monte Carlo simulation is the safest way to get reliable error estimates on the global solution.

Exercises for Section 4:

(1) This Exercise lets you have a go at a real orbital fitting problem. Your target is the star ρ CrB (meaning, the 17^{th} brightest star in the constellation *Corona Borealis*), one of the first stars around which a planet was detected (see Noyes *et al.* 1997). You can obtain radial velocity data from the Tutorial Web page under Data. These data were obtained using the Advanced Fiber Optic Echelle (AFOE) spectrograph. Follow the procedure outlined in §4.3 to obtain a set of best-fit orbital elements.

(2) Physically, what do you thing causes the $[\tau, \omega]$ degeneracy so obviously apparent on Fig. 15(D)?

5. FINAL THOUGHTS AND FURTHER READINGS

5.1 To cross over or not to cross over?

That is indeed the question at the center of what has been at times a heated debate between proponents of two classes of evolution-inspired search techniques, namely Genetic Algorithms and the so-called Evolution Strategies. The fact that proponents of each class of techniques were also originally segregated geographically did not exactly facilitate the early phases of the debate. The initial study and development of genetic algorithm is due to John Holland and collaborators at the University of Michigan in the 1960's and early 1970's. Evolution Strategies, on the other hand, were originally developed independently and more or less simultaneously by a group of researchers at the Technical University of Berlin.

Both classes of techniques make use of the natural selection analogy, and incorporate inheritance and variation in some form. They differ primarily in how they breed selected solutions; usually, classical evolution strategies use deterministic schemes for selection and population replacement, and make use of perturbation and recombination operators that are functionally equivalent to mutation and crossover. These operators are usually self-adapting, and are designed to be significantly "smarter" than the simple biologically inspired one-point crossover/mutation described in §2.3 (see Michalewicz 1996, §5.3.2, and Bäck, §2.1, for some nice examples of such "smart" mutation operators). Equally important, these operators are applied directly on the floating-point representation of the parameters defining the model being optimized.

Classical genetic algorithms, on the other hand, have tended to stick to relatively simple selection and population replacement techniques, and to simple genetic operators that act on a string-encoded version of the model parameters. Mutation is usually seen as a secondary operator, needed to continuously inject variability in the population being acted upon by crossover and fitness-based selection.

At any rate, the distinction between genetic algorithms and evolution strategies is becoming increasingly blurred with time, as most current codes incorporate components from both. This is the case for `PIKAIA`, which carries through breeding in a manner essentially in accord with classical Genetic Algorithms, but uses a dynamically adjustable mutation rate in a manner reminiscent of Evolution Strategies. In evolutionary phases where the mutation rate is low, `PIKAIA` operates pretty much like a classical genetic algorithm; when the mutation rate is high, `PIKAIA` functions more like a stochastic hill-climber. This is a powerful algorithmic combination indeed. As for the ultimate worth of classical crossover, you are hereby encouraged to form your own opinion by doing Exercise 5.1 below.

5.2 Hybrid methods

In the preceding pages we had a few occasions to refer to the characteristic shape of convergence curves for genetic algorithm-based optimizer: fast decrease of the error in the early phase of the evolution, followed by long periods of more or less constant error value punctuated by episodes of rapid error decrease, triggered by the appearance and subsequent spread in the population of favorable mutations. This is quite unlike classical local hill climbing schemes which —when they do converge— exhibit a more or less steady decrease of the error, at a characteristic rate for a given algorithm (see for example the two converged simplex runs on Fig. 4).

Local hill climbing is fast but local; genetic algorithms are slow but global. This dichotomy is at the very core of the No Free Lunch Rule, but can actually be made to work to one's advantage by *combining* both techniques. This may involve running `PIKAIA` until no improvement is made to the best individual in the previous $N_g/3$ generations (say), and then using this individual to initialize the simplex method (or any other local hill climbing method for that matter). Such a *hybrid scheme* combines the good exploratory capabilities of genetic algorithms and the superior convergence behavior of other methods in the vicinity of an extremum. This will often represent the optimally efficient use of `PIKAIA` when high accuracy (dubbed "absolute performance" in §1.1) is required on real-life problems. Care must still be taken not to stop the genetic algorithm too soon in the sake of economy in the number of function evaluations, otherwise Dirty Harry will end up catching up up with you one of these days. I know you have now heard it a few times already, but once again, *there really is no such thing as a free lunch!!*

5.3 When should you use genetic algorithms

Because they rely only on a single scalar quantity, fitness, to carry out their task, genetic algorithm-based optimizers are easy to use for very wide classes of problems. You can use a genetic algorithm to solve *anything* that can be formulated as a minimization/maximization task —and in principle anything described by an equation can. This, of course, does not at all mean that you should. If you already have something that works well enough for you, *don't mess with it!* This is in fact the First Rule of Scientific Computing, also known as

> HAMMING'S FIRST RULE:
>
> "The purpose of computing is insight, not numbers"

When, then, should you consider using genetic algorithms on a real-life research problem? There is no clear-cut answer to this questions, but based on my own relatively limited experience I would offer the following list:

(1) Markedly multimodal optimization problems where it is difficult to make a reliable initial guess as to the approximate location of the global optimum;

(2) Optimization problems for which derivatives with respect to defining parameters are very hard or impossible to evaluate in closed form. If a reliable initial guess is available, the simplex method is a strong contender; if not, genetic algorithms become the method of choice;

(3) Ill-conditioned data modeling problem, in particular those described by integral equations;

(4) Problems subjected to positivity or monotonicity constraints than can be hardwired in the genetic algorithm's encoding scheme.

This is of course not meant to be exclusive of other classes of problems. One constraint to keep in mind is the fact that genetic algorithm-based optimization can be CPU-time consuming, because of the large number of model evaluations typically required in dealing with a hard problem. The relative ease with which genetic algorithms can be parallelized can offset in part this difficulty: on a "real life" problem most work goes into the fitness evaluation, which proceeds completely independently for each individual in the population (see Metcalfe and Charbonneau 2002 for a specific example). Never forget that your time is always more precious than computer time.

In the introductory essay opening his book on genetic programming, Koza (1992) lists seven basic features of "good" conventional optimization methods: correctness, consistency, justifiability, certainty, orderliness, parsimony, and decisiveness. He then goes on to argue that genetic algorithms embody none of these presumably sound principles. Is this ground to reject optimization methods based on genetic algorithms? Koza does not think so, and neither do I. From a practical point of view the bottom line always is: use whatever works. In fact, that is precisely the message conveyed, loud and clear, by the biological world.

I would like to bring this tutorial to a close with a final, Third Rule of Global Optimization. Unlike the first two, you probably would *not* find something equivalent in optimization textbooks. In fact I did not come up with this rule, although I took the liberty to rename it. It originates with Francis Crick, co-discoverer of DNA and 1962 Nobel Prize winner. So here is the Fourth Rule of Global Optimization, also known as[24]

THE NO-GHOST-IN-THE-MACHINE RULE:

"Evolution is cleverer than you are"

5.5 Further reading

There are now quite a few textbooks or monographs available on evolution strategies and genetic algorithms. The following three are my personal favorites:

Goldberg, D.E. 1989, *Genetic Algorithms in Search, Optimization & Machine Learning*, Reading: Addison-Wesley,

Davis, L. 1991, *Handbook of Genetic Algorithms*, New York: Van Nostrand Reinhold,

Bäck, T. 1996, *Evolutionary Algorithms in Theory and Practice*, Oxford: Oxford University Press.

Goldberg's book is a textbook, complete with exercises and examples worked out in great detail. Bäck obviously enjoys abstruse mathematical notation far more than I do, but his book is definitely recommended reading. Not the least of its merits is its unifying presentation of genetic algorithms and evolution strategies as different incarnations of Evolutionary Algorithms, which helps a lot to appreciate their similarities and differences. Bäck also has collected a set of really nasty test

[24] For reasons best known to himself, Crick calls this "Orgel's Second Rule". If any reader of this Tutorial happens to know what "Orgel's First Rule" might be, please let me know.

functions. I have retained a soft spot for Davis' book because this is the book I used to teach myself genetic algorithms many years ago. It is far less comprehensive in its coverage than Goldberg's or Bäck's books, but has an application-oriented, no-nonsense flavor that I found and continue to find very refreshing. Two other books well worth looking into are

Michalewicz, Z. 1996, *Genetic Algorithms + Data Structures = Evolution Programs*, third ed., New York: Springer,

Mitchell, M. 1996, *An Introduction to Genetic Algorithms*, Cambridge: MIT Press.

Both of these put more emphasis on the type of non-numerical optimization problems that are closer to the hearts of most computer scientists, such as the Traveling Salesman Problem. Mitchell's book has a nice chapter describing applications of genetic algorithms in modeling evolutionary processes in biology. Genetic algorithms were originally developed in a much broader context, centering on the phenomenon of *adaptation* in quite general terms. Anybody serious about using genetic algorithms for complex optimization tasks should make it a point to work through the early bible in the field:

Holland, J.H. 1975, *Adaptation in Natural and Artificial Systems*, Ann Arbor: The University of Michigan Press (second ed. 1992, MIT Press).

If you want to know where the name *Pikaia* comes from, or if you think you just might enjoy an excellent book on evolution, read

Gould, S.J. 1989, *Wonderful Life. The Burgess Shale and the Nature of History*, New York: W.W. Norton & Company.

Finally, here are some samples from the computing science literature that I found useful and intellectually stimulating in pondering over some of the more "philosophical" material contained in this tutorial:

Bäck, T., Hammel, U., and Schwefel, H.-P. 1997, Evolutionary computation: comments on the history and current state, in *IEEE Transactions on Evolutionary Computation*, **1**, 3-16,

Culberson, J.C. 1998, On the futility of blind search: an algorithmic view of "No Free Lunch", *Evolutionary Computation*, **6**, 109-127,

DeJong, K.A. 1993, Genetic algorithms are NOT function optimizers, in *Foundation of genetic algorithms 2*, ed. L.D. Whitley (San Mateo: Morgan Kaufmann), Hamming, R.W. 1962, *Numerical Methods for Scientists and Engineers*, New York: McGraw-Hill, chap. $N+1$,

Holland, J.H. 1995, *Hidden Order: How Adaptation builds Complexity* (Reading: Addison-Wesley),

Koza, J.R. 1992, *Genetic Programming: on the programming of computers by means of natural selection* (Cambridge: MIT Press), chap. 1,

Wolpert, D.H., and Macready, W.G. 1997, No Free Lunch theorems for optimization, in *IEEE Transactions on Evolutionary Computation*, **1**, 67-82.

The least you should remember from Section 5:

- If you need high accuracy in fitting parameters, use a hybrid scheme.
- If it works, don't mess with it.
- Don't be a purist; use whatever works best.
- Your time is more valuable than CPU time.
- The purpose of computing is insight, not numbers.
- Evolution is cleverer than you are.

Exercises for Section 5:

(1) Write fitness functions for the P1, P2 and P3 test problems defined in §1.4. Solve these problems using `PIKAIA`'s default settings, except for (1) setting the number of generation (`ctrl(2)`) to the values listed in Table 2, and (2) turning off crossover. This is most easily done by simply setting `ctrl(4)=0`. Compare your global convergence results to those listed in Table 2 for GA2. Is crossover a good thing?

(2) You should not try this exercise unless you have already done a few other exercises where you had to use `PIKAIA`. The idea to explore is to let the selection pressure parameter vary dynamically in the course of the run, in a similar way as `PIKAIA` varies the mutation rate. The first thing you need to do is to read carefully §3.4 of the PUG, to fully understand how `PIKAIA` relates fitness-based rank to selection probability, and §3.7 to fully understand how it dynamically varies the mutation rate. Use the ideas developed in §3.7.2 of the PUG to let the parameter `fdif` ($\equiv$`ctrl(9)`) vary in response to the spread

of the population in parameter space. Run this modified `PIKAIA` on the four test problems of §1.4 and compare the results to the GA2 algorithm[25].

---

[25] There is no answer to this exercise on the Tutorial Web Page, only a few hints; but if you do get interesting results (namely, significantly enhanced performance that remains robust across problem domain), I would very much like to hear about it.

BIBLIOGRAPHY

Bäck, T. 1996, *Evolutionary Algorithms in Theory and Practice*, Oxford: Oxford University Press.

Bertiau, F.C. 1957, *Astrophys. J.*, **125**, 696

Bevington, P.R., & Robinson, D.K. 1992, *Data Reduction and Error Analysis for the physical Sciences*, second ed., New York: McGraw-Hill

Charbonneau, P. 2002, *Release Notes for PIKAIA 1.2*, NCAR Technical Note 451+STR, Boulder: National Center for Atmospheric Research (PUG)

Charbonneau, P., & Knapp, B. 1995, *A User's Guide to PIKAIA 1.0*, NCAR Technical Note 418+IA, Boulder: National Center for Atmospheric Research (PUG)

Charbonneau, P., Tomczyk, S., Schou, J., & Thompson, M.J. 1998, *Astrophys. J.*, **496**, 1015

Darwin, C. 1859, *On the Origin of Species by Means of Natural Selection, or the Preservation of favoured Races in the Struggle for Life*, London: J. Murray

Dawkins, R. 1986, *The Blind Watchmaker*, New York: W.W. Norton

Eigen, M. 1971, *Die Naturwissenschaften*, **58**(10), 465

Gibson, S.E., & Charbonneau, P. 1998, *J. Geophys. Res.*, **103**(A7), 14511

Goldberg, D.E. 1989, *Genetic Algorithms in Search, Optimization & Machine Learning*, Reading: Addison-Wesley

Hamming, R.W. 1962, *Numerical Methods for Scientists and Engineers*, New York: McGraw-Hill

Holland, J.H. 1975, *Adaptation in Natural and Artificial Systems*, Ann Arbor: The University of Michigan Press (second ed. 1992, MIT Press)

Jenkins, F.A., & White, H.E. 1976, *Fundamentals of Optics*, fourth ed., New York: McGraw-Hill

Koza, J.R. 1992, *Genetic Programming: on the programming of computers by means of natural selection*, Cambridge: MIT Press

Maynard Smith, J. 1989, *Evolutionary Genetics*, Oxford: Oxford University Press

McIntosh, S.W., Diver, D.A., Judge, P.G., Charbonneau, P., Ireland, J., & Brown, J.C. 1998, *Astron. Ap. Suppl.*, **132**, 145

Metcalfe, T., & Charbonneau, P. 2002, *J. Comp. Phys.*, submitted

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., & Teller, E. 1953, *J. Chem. Phys.*, **21**, 1087

Michalewicz, Z. 1996, *Genetic Algorithms + Data Structures = Evolution Programs*, third ed., New York: Springer

Nelder, J.A., & Mead, R. 1965, *Computer J.*, **7**, 308

Noyes, R.W., Jha, S., Korzennik, S.G., Krockenberger, M., Ninenson, P., Brown, T.M., Kennelly, E.J., & Horner, S.S. 1997, *Astrophys. J. Lett.*, **483**, L111

Petrie, R.M. 1962, in *Astronomical Techniques*, ed. W.A. Hiltner; vol. II of *Stars and Stellar Systems*, eds. G.P. Kuiper & B.M. Middlehurst, Chicago: University of Chicago Press, chap. 23

Press, W.H., Teukolsky, S.A., Vetterling, W.T., & Flannery, B.P. 1992, *Numerical Recipes*, Second Ed., Cambridge: Cambridge University Press

Smart, W.M. 1971, *Textbook on Spherical Astronomy*, fifth ed., Cambridge: Cambridge University Press