NCAR/TN-XXX+IA
NCAR TECHNICAL NOTE

March 2002

RELEASE NOTES FOR PIKAIA 1.2

Paul Charbonneau

HIGH ALTITUDE OBSERVATORY

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
BOULDER, COLORADO

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The theory of punctuated equilibrium, a variation on conventional Darwinian evolution theory, states that biological species typically remain genetically (and thus phenotypically) stable for extended periods of (geological) time, and that evolutionary significant change and speciation occur in short "bursts" of evolutionary activity. I hereby propose that punctuated equilibrium also adequately characterizes the evolution of computing software.

Version 1.0 of the genetic algorithm-based optimization subroutine `PIKAIA` was released in December 1995, and except for a few minor fixes in the following few months thereafter, has remained the only version publicly available. Seven years in the computing software world is arguably equivalent to a full geological era in the physical world, so a punctuated change should be expected. Sure enough, a new software version, `PIKAIA 1.2`, is now ready for public release. Version 1.2 retains the overall format and Numerical recipes "style" of version 1.0. It only differs internally, in that it incorporates additional genetic operators and algorithmic strategies not included in the original release of `PIKAIA 1.0`.

I am not releasing version 1.2 just for the sake of satisfying my own computing software version of punctuated equilibrium; experience garnered since 1995 indicated that the performance of the original `PIKAIA` could be improved significantly, with relatively minor and user-transparent changes and additions. This is the true motivation behind the release of `PIKAIA 1.2`.

These Release Notes for `PIKAIA 1.2` are organized as follows: Chapter 1 is really all that a current `PIKAIA` user needs to know to move on to version 1.2. It describes the new operators and strategies included in `PIKAIA 1.2`, their coding implementation, and the new control variable settings that allows these new options to be used. Chapter 2 is my attempt to convince current `PIKAIA` users that switching to `PIKAIA 1.2` is worth the (small) effort. It is a performance study of various algorithmic versions of `PIKAIA`, tested again a suite of maximization problems I have been developing and using over the years whenever I experimented with new strategies and operators. Chapter 3 discusses three topics that I thought could be of interest to some current or potential `PIKAIA` users out there, although they have nothing really to do with `PIKAIA 1.2` specifically: `PIKAIA` in computing languages other than plain old dum FORTRAN-77; combinatorial optimization with `PIKAIA`; and extant parallel versions `PIKAIA`. The fourth and final chapter is rather far off left field. It is basically a short essay on Numerical Optimization, musing over some general thoughts on the topic which I always find myself throwing around when giving talks or colloquia on genetic algorithms. Parts of that essay are taken, with modifications, from the writeup of a series of lectures I gave a few years ago at a Computational Astrophysics Workshop in Oslo, Norway, and which ended up never being published (although the full text remains available on the `PIKAIA` Web Page, see §X.Y herein). Some will undoubtedly find the essay bombastic, pedant, silly, irreverent, or just plain irrelevant. It can certainly be omitted without impeding your use and enjoyment of `PIKAIA 1.2`.

I wish to thank all of my colleagues, at NCAR and elsewhere and too numerous to list here, whose questions and queries over `PIKAIA` and genetic algorithms continue to sharpen my thinking, and generally keep me on my toes. I do wish to thank explicitly Adrian Webster, Richard Boivin, Luciano Mantegazza, and Andreas Bobinger, for sending in bug reports following the original release of `PIKAIA`; Steve Tomczyk for promptly picking out and reporting a number of typos in the original `PIKAIA` User's Guide; Sarah Gibson, Scott McIntosh, and Alan Miller, for kindly making publicly available version of `PIKAIA` in computing languages other than FORTRAN-77.

Finally, I take this opportunity to thank once again my friend and former colleague Barry Knapp, now at the University of Colorado at Boulder, without whom `PIKAIA 1.0` would have never been the user-friendly software that so many people evidently have found it to be. `PIKAIA` is as much his brainchild as it is mine, and I hope that in preparing `PIKAIA 1.2` I have managed to live up to the good programming standards he tried so hard to imprint upon my thick skull back then in the previous millennium.

Paul Charbonneau

April 2002, Boulder

1. INTRODUCING PIKAIA, VERSION 1.2

This chapter contains everything needed by a user of `PIKAIA 1.0` to get going with the new version `PIKAIA 1.2`. It assumes that the reader is reasonably familiar with the basic ideas underlying a Genetic Algorithm, as well as with the operation and overall structure of the `PIKAIA` subroutine. Whenever in doubt, consult the `PIKAIA` user's guide (Charbonneau & Knapp 1995, NCAR Technical Note 418-IA; hereafter "PUG"). For good, general introductions to genetic algorithms, see Goldberg (1989), Davis (1991), Bäck 1996, or Mitchell (1996); and for `PIKAIA`-specific introductions, Charbonneau (1995, 1998).

1.1 `PIKAIA` then and now

`PIKAIA` is a general-purpose genetic algorithm-based numerical optimization subroutine, written in ANSI-standard FORTRAN-77. Version 1.0 was released in December 1995. It's main selling point, as compared to other genetic algorithm packages available commercially or in the public domain, was and remains its ease of use and "Numerical-Recipes" style. At the time of its public release, `PIKAIA 1.0` was viewed by its designers primarily a learning tool, although it had been used successfully already on a number of research applications. As of March 2002, a tally (undoubtedly incomplete) reveals 71 past or present users, and 30 papers published in refereed journals where `PIKAIA` was used as part of the work. Perhaps because `PIKAIA` was first described in the pages of *The Astrophysical Journal*, the majority of research applications remain in the space, solar and astrophysical sciences, although it has also been used in engineering design, geoseismic inversions, XXX

In the course of the past seven years it has become clear that `PIKAIA 1.0` suffers from a few simple shortcomings that can easily be remedied. Starting already in 1998, a few users reporting convergence problems were supplied with a private-release "improved" version of the original subroutine. Recurrent surfacing of these problems, and additional intermittent twiddling and testing of `PIKAIA` "improvements" by the present author, has made it clear that relatively minor modifications and additions to `PIKAIA 1.0` could lead to significantly better performance. Thus evolved `PIKAIA 1.2`.

1.2 Description of new features

PIKAIA 1.2 retains the overall structure, I/O, and calling sequence of version 1.0. Internally, it includes three new features: two-point crossover, creep mutation and distance-based adjustment of the mutation rate. The following gives brief descriptions of these features, along with the occasional difficulties encountered with PIKAIA 1.0 that motivated their inclusions in this new software release. Full listings for new code elements can be found in the Appendix.

1.2.1 Two-point crossover

The uniform one-point crossover included in PIKAIA 1.0 (see PUG, §3.6), suffers from "end-point" bias; consider the following strings resulting from the decimal encoding of a set of floating-point parameter values defining the "model" being optimized:

```
1234567890123456789012345678901 23
```

Suppose now, for the sake of the argument, that the two substrings "123" at the two ends of the string, when decoded, turn out to be advantageous, in that their give their bearer above-average fitness. This advantageous combination is basically impossible to copy intact into a single offspring string following the application of uniform one-point crossover. This would be much less of a problem for the two other 123 substrings more centrally located, where the disruption probability would here be $\simeq 0.4$.

Two-point crossover bypasses this difficulty by selecting *two* splicing points along the string, and exchanging the string portion located in between these splicing points in a manner otherwise identical to one-point crossover. For example:

```
123456789012345678901234567890123
987654321098765432109876544321098
.....|..........|................

    12345|67890123456|78901234567890123
    98765|43210987654|32109876544321098

        12345|43210987654|78901234567890123
        98765|67890123456|32109876544321098

            123454321098765478901234567890123
            987656789012345632109876544321098
```

A moment of reflection will reveal that the endpoints of the string are now much more likely to end up in the same offspring string. In fact, we have overcorrected our problem: string endpoints are now *too likely* to end up on the same offspring. The simplest way out of this dilemma is to introduce an additional probabilistic test to the crossover operation, whereby either one-point or two-point crossover is chosen, with equal probability. `PIKAIA 1.2` does precisely this, as shown in the following fragment of the new crossover subroutine `pcross`:

```
   ispl=int(urand()*n*nd)+1      [ first crossover point         ]
   if (urand().lt.0.5) then      [ pick 1-pt vs 2-pt crossover   ]
      ispl2=n*nd                 [ this for one-point crossover  ]
   else
      ispl2=int(urand()*n*nd)+1  [ second crossover point        ]
      if (ispl2.lt.ispl) then    [ ensure islp<=ispl2            ]
         tmp=ispl2
         ispl2=ispl
         ispl=tmp
      endif
   endif
   do 1 i=ispl,ispl2             [ now swap from ispl to ispl2   ]
      t=gn2(i)
      gn2(i)=gn1(i)
      gn1(i)=t
 1 continue
```

Note that two-point crossover is *not* included in most of the "improved" versions of `PIKAIA 1.0` privately distributed since 1998.

1.2.2 Creep mutation

In conjunction with the decimal encoding/decoding scheme, the uniform one-point mutation included in `PIKAIA 1.0` (see PUG, §3.7.1), is liable to getting stuck at "Hamming Walls". This problem was already noted in the context of the non-linear least-squares fit discussed in the example chapter of the PUG (§5.3), and most often occurs in later evolutionary phases, when a pretty good trial solutions is being refined by the action of mutation.

Consider the portion of a string `...4251...` encoding a floating-point parameter $a = 0.4255$. Suppose now that the optimal setting for this parameter happens to be $a^* = 0.4110$. Suppose also that the fitness landscape is smooth enough that the fitness increases gradually as a^* is approached from above or below. Evidently, many possible sequences of single-digit replacements can transform the current substring to its "target" optimum. For example,

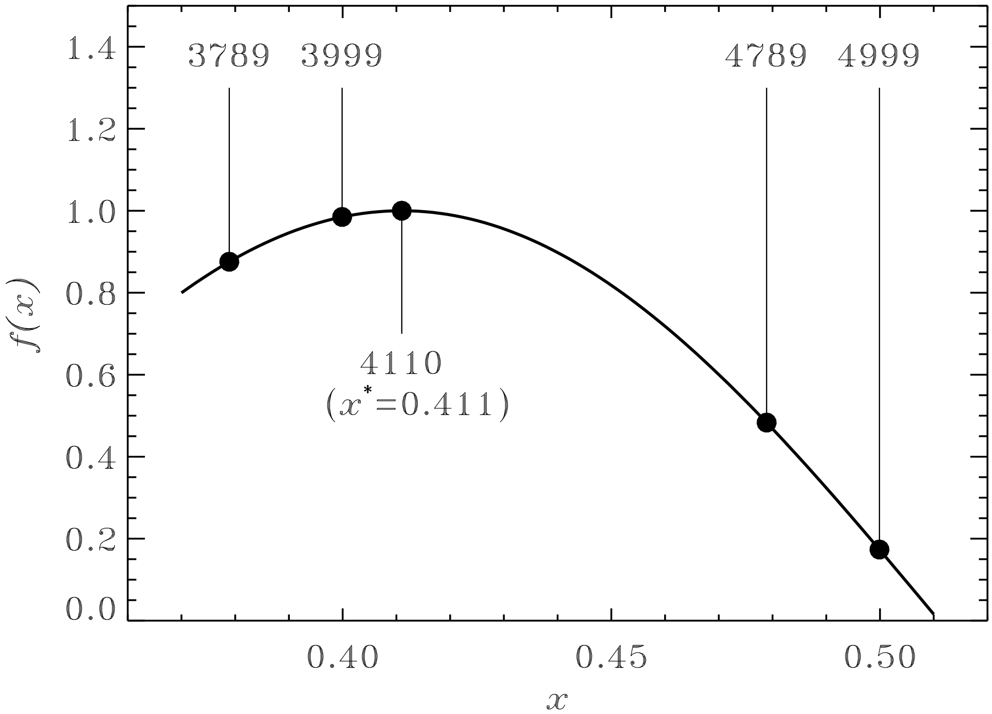

Figure 1.1: A Hamming Wall in a decimal encoding scheme. One-point mutation cannot move an decimally-encoded parameter value $x = 0.3789$ to the optimum $x^* = 0.411$ along an evolutionary favored sequence of intermediate, i.e., a sequence of gradually increasing fitness $f(x)$ (see text).

`4251` → `4221` → `4121` → `4111` → `4110`
`..2.` → `.1..` → `..1.` → `...0`

Because each intermediate step represents an increase in fitness, the above transition is evolutionary favored. But what if the starting string had been `...3789...`? The sequence

`3789` → `4789` → `4189` → `4119`, etc
`.4..` → `.1..` → `..1.`

while possible in principle, will *not* be evolutionary favorable. The problem, illustrated on Figure 1.1, is simply that the second-step individual with $a = 0.4789$ has much smaller fitness than the original $a = 0.3789$. On the other hand, the following sequence would be evolutionary favorable:

`3789` → `3799` → `3899` → `3999`

```
..9. → .8.. → .9..
```

and now we're stuck! As can be seen on Figure 1.1, the one-point mutation transition `3999` → `4999` is even less evolutionary advantageous than our earlier possibility `3789` → `4789`. At least two well-coordinated and simultaneous mutations must occur for the transition to take place with a concomittant increase in fitness, e.g., something like

```
3999 → 4099
40..
```

Because such a chance event is very unlikely to take place in a timely manner, the population will cluster at $a = 0.3999$, without further improvement in fitness. By all appearances, the solution has converged... but unfortunately, not on the global optimum; just flat against a Hamming Wall located in its vicinity.

There are two ways out of this quandary. The first is to devise an encoding scheme where successive single-digit changes at the level of the string map into smooth variations of the decoded floating-point parameter across its whole range. Gray binary coding (see, e.g., Press et al. 1992, §X.Y) is a well-known instance of this approach. However, this can rapidly become cumbersome as the string length increases. Alternately, one can simply devise mutation operators that can "carry over the one".

The creep mutation operator included in `PIKAIA 1.2` achieves this in the following way. Once a digit has been targeted for mutation, the corresponding digit is either *incremented* or *decremented* (with equal probabilities). In the case of a unit increment, if the digit happened to be a "`9`", it becomes "`0`" and that located to its left is also increment by unity (and the whole process repeats as many time as needed if the left neighbouring digit also happened to be a "`9`"). The corresponding sequences of digit operation associated with unit decrement should now be obvious to anyone having survived first-grade arithmetics.

In the case of the above example, once stuck at `3999` an evolutionary favored transition could be produced by creep mutation as follows:

```
3999 → 3909 → 3009 → 4009, etc
..+.
```

Note that the above sequence of steps represents the action of a single creep mutation event beginning at the third digit of the substring. In `PIKAIA 1.2`, creep mutation is coded up as follows in subroutine `mutate`:

```
      do 1 i=1,n                             [loop over substrings     ]
```

```
   do 2 j=1,nd                          [loop over digits          ]
    if (urand().lt.pmut) then           [mutation probability test]
     loc=(i-1)*nd+j                     [location on string        ]
     inc=nint ( urand() )*2-1           [pick mutation increment   ]
     ist=(i-1)*nd+1                     [substring left boundary   ]
     gn(loc)=gn(loc)+inc                [creep mutation            ]
     if(inc.lt.0.and.gn(loc).lt.0)then[creep decrement (-1) case]
      if(j.eq.1)then                    [at substring boundary:    ]
       gn(loc)=0                        [no creep allowed!         ]
      else
       do 3 k=loc,ist+1,-1              [creep back up substring   ]
        gn(k)=9                         [turn 0 to 9               ]
        gn(k-1)=gn(k-1)-1               [carry over the one        ]
        if(gn(k-1).ge.0)goto 4          [exit creep process        ]
3      continue
       if( gn(ist).lt.0.)then           [we popped under 0.0000...]
        do 5 l=ist,loc                  [...so we fix it up        ]
         gn(l)=0
5       continue
       endif
4      continue                         [done with decrement creep]
      endif
     endif
     if(inc.gt.0.and.gn(loc).gt.9)then[creep increment (+1) case]
      if(j.eq.1)then                    [at substring boundary:    ]
       gn(loc)=9                        [no creep allowed!         ]
      else
       do 6 k=loc,ist+1,-1              [creep back down substring]
        gn(k)=0                         [turn 9 to 0               ]
        gn(k-1)=gn(k-1)+1               [carry over the one        ]
        if( gn(k-1).le.9 )goto 7        [exit creep process        ]
6      continue
       if( gn(ist).gt.9 )then           [we popped over 9.9999....]
        do 8 l=ist,loc                  [...so we fix it up        ]
         gn(l)=9
8       continue
       endif
7      continue                         [done with increment creep]
      endif
     endif
```

```
      endif
    2  continue                              [done with this digit     ]
    1 continue                               [done with this substring ]
```

The coding ends up looking a tad intricate, because ANSI-standard FORTRAN-77 supports neither recursion or `WHILE` loops. As you examine the code, notice how extra precautions are needed to prevent the operator from creeping across from one parameter-defining substring to another: creep mutation should only operate within a substring that decodes to a single parameter, otherwise other parameters might get corrupted by "runaway creep". The large number of short `DO` loops and `IF` statements might also raise concerns regarding execution time; this is a good place to remind ourselves that in most real research GA applications, most of the CPU time is spent evaluating the fitness functions, so extra work within the GA usually has little consequence on total CPU or wallclock execution time.

Used in conjunction with strong selection pressure, creep mutation operates as a local hill climber: by design, it always generate small steps away from a current good solution (the largest jump generated in parameter space is caused by a ± 1 change in a string digit decoding into a leading digit in the corresponding floating-point parameter). An automatic consequence is then that creep mutation cannot cause large jumps in parameter space, the way one-point mutation does when it happens to operate on a digit that decodes into a leading digit in the decoded parameter. Because such large jumps are useful for the exploration of parameter space, it turns out to be advantageous to use either one-point or creep mutation, with equal probabilities. This is achieved by an additional probability test upon entering subroutine `mutate` (see full code listing in the Appendix).

1.2.3 Distance-based adjustment of the mutation rate

`PIKAIA 1.0` includes a very basic form of self-adaptation of the mutation probability p_m ($\equiv$`pmut=ctrl(6)`). At the end of each generational iteration, the degree of clustering of the population is computed using as a measure the normalized fitness difference between the best and median individuals (according to fitness-based rank; see PUG, §3.7.2):

$$\Delta_f = \frac{f^{\max} - f^{\text{med}}}{f^{\max} + f^{\text{med}}} \tag{1.1}$$

where $f^{\max} \equiv f(x^{\max})$ and $f^{\text{med}} \equiv f(x^{\text{med}})$, and $x^{\max}$, x^{med} are the parameter sets defining the best and median individuals, respectively. When $\Delta_f \to 0$, the population is strongly clustered, while if $\Delta_f \to 1$ the population is scattered across parameter space. The idea is to increase p_m in the former case, and decrease it in the latter. Doing so helps preventing premature convergence, since high p_m favors displacement away from secondary extrema, while ensuring that efficient exploration of parameter space by the crossover operator (requiring $p_m \ll 1$ to avoid excessive disruption of the building blocks being assembled by crossover) takes place whenever the population is dispersed in parameter space.

Evidently, eq. (1.1) is just one of many possible measures of population clustering. Another obvious possibility would be to use a measure of metric distance between best and median, for example:

$$\Delta_d = \frac{1}{\mathtt{n}} \left(\sum_{j=1}^{\mathtt{n}} (x^{\max} - x^{\mathrm{med}})^2 \right)^{1/2} . \tag{1.2}$$

This latter measure of population clustering turns out to be preferable to eq. (1.1) in certain cases[1]. Figure 1.2 illustrates the idea. A fitness-based criterion such as eq. (1.1) behaves as wanted for high-contrast fitness landscapes (panels A and B), but in a low-contrast case (panel C) indicates clustering ($\Delta_f \to 0$) when the best and median individuals are in fact far apart from one another. A distance-based criterion such as eq. (1.2) behaves appropriately in this latter case, but in high-contrast situation can indicate clustering prematurely ($\Delta_d \to 0$, as on panel B).

`PIKAIA 1.2` now includes the possibility of using either a fitness-based (eq. (1.1)) or distance-based (eq. (1.2)) criterion to monitor population clustering. The choice is made by appropriate setting of the mutation mode control variable `imut`≡`ctrl(5)`, a detailed in Table II. Unless one knows for a fact that a problem is characterized by low-fitness contrast across the parameter space, fitness-based mutation rate adjustment should be tried first[2]

Independently of which adjustment criterion is invoked, `PIKAIA 1.2` varies the mutation rate according to the "recipe" used in `PIKAIA 1.0` (see PUG, §3.7.2):

$$p_m \to \begin{cases} p_m \times \delta & \text{if } \Delta \leq \Delta^{\mathrm{low}}, \\ p_m & \text{if } \Delta^{\mathrm{low}} < \Delta < \Delta^{\mathrm{high}}, \\ p_m/\delta & \text{if } \Delta \geq \Delta^{\mathrm{high}}, \end{cases} \tag{2.3}$$

[1] The $1/\mathtt{n}$ normalization factor in eq. (1.2) means that Δ_d is not a true measure of metric distance. However, numerical experiments I carried out with the suite of test problems described in the next chapter indicate that such a normalization factor yields a better mutation rate adjustment.

[2] One can of course design criteria that incorporate both fitness-based and distance-based clustering measures. For example, one could deem the population clustered only IF Δ_f AND Δ_d are both small, and spread out IF Δ_f OR Δ_d are large. I played a bit with criteria of this type, and surprisingly (to me anyway), on many of the test problems described in chapter 3 below they often *degraded* performance as compared to either criterion used in isolation.

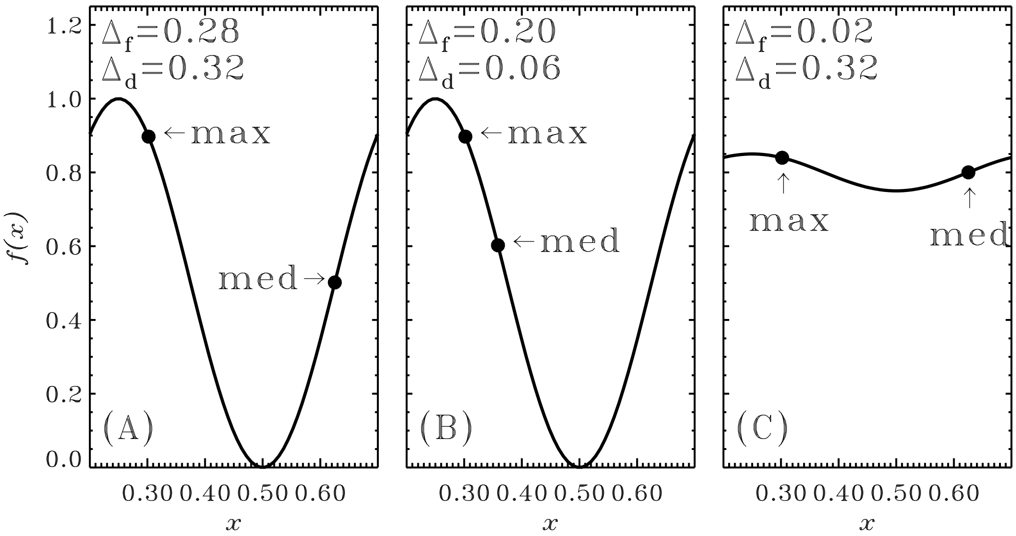

Figure 1.2: Two measures of population clustering in three distinct cases. In high fitness contrast situations (panel B), a fitness-based criterion is preferable, while in low-contrast cases (panel C) a distance-based criterion is better behaved. There are of course many situations where either criterion behaves more-or-less equally well (panel A).

where $\delta = 1.5$, $\Delta^{\rm low} = 0.05$, $\Delta^{\rm high} = 0.25$ are hardwired via a `PARAMETER` statement in subroutine `adjmut`, and Δ stands for either Δ_f or Δ_d, as the case may be. Again as in `PIKAIA 1.0`, the absolute lower and upper bounds allowed on the mutation probability p_m are set by the control variables `pmutmn`$\equiv$`ctrl(7)` and `pmutmx`$\equiv$`ctrl(8)` (see PUG, §4.5). For implementation details, see the source code for the new subroutine `adjmut` listed in the Appendix of these Release Notes.

1.2.4 New control parameter settings for `PIKAIA 1.2`

To facilitate upward compatibility, the new strategies and operators described above have been incorporated into `PIKAIA 1.2` in a manner such that the calling sequence and input control vector (`ctrl`) length of `PIKAIA 1.0` remained unchanged.

A 50/50 mixture of one-point and two-point crossover is now hardwired in `PIKAIA 1.2`. Quantitative support for this decision can be found in §3.3 below. The crossover probability `pcross` ($\equiv$`ctrl(4)`) is then the probability per breeding event of either one-point or two-point crossover to take place. The user wishing to enforce exclusive use of one-point crossover must enforce `ispl2=n*nd` in subroutine `cross`, by un-commenting the appropriate line of source code (see full listing of the new subroutine `cross` in the Appendix).

The remaining two additional operators and strategies included in `PIKAIA 1.2`, namely creep mutation and distance-based adjustment, can be invoked by setting the value of the control variable `imut` ($\equiv$`ctrl(5)`) to the values listed in the following Table:

Table I

New valid values for control variable `imut`

`imut`	Adjustable rate	Based on	Creep	Note
1	No	N/A	No	1,2
2	Yes	Fitness	No	1,3
3	Yes	Distance	No	
4	No	N/A	Yes	2
5	Yes	Fitness	Yes	
6	Yes	Distance	Yes	

Notes:
1= Only allowed settings in `PIKAIA 1.0`
2= Setting not recommended; see main text.
3= Default value in both `PIKAIA 1.0` and `PIKAIA 1.2`

All other control variable settings remain the same as in `PIKAIA 1.0`, and are specified by the user as elements of the input control vector `ctrl` (see PUG, Table I). Note that for completeness, the possibility of using 50/50 one-point and creep mutation *without* dynamical adjustment of the mutation rate is allowed (`imut`=4). However, this setting, like pure one-point mutation without rate adjustment, yields a much less-than-optimal global algorithm on most problems. Therefore, neither `imut`=1 or 4 are recommended settings for `PIKAIA 1.2`.

1.3 Compatibility with PIKAIA 1.0

`PIKAIA 1.2` is operationally compatible with `PIKAIA 1.0`, in the sense that the new subroutine can be substituted for the old one without any changes in the source code for the calling program, user-supplied fitness function `ff(n,x)`, or settings of the control variable input array `ctrl`. In particular, all default settings in `PIKAIA 1.2` are the same as in `PIKAIA 1.0`. However, even if starting from the exact same initial random population, different evolutionary paths can and will occur, for two reasons:

(1) `PIKAIA 1.2` uses two-point crossover for 50% of the breeding events which pass the initial crossover probability test upon entering subroutine `cross`.

(2) Even if the user enforces one-point crossover (by forcing `ispl2=n*nd` in subroutine `cross`), comparison of the version `1.0` and `1.2` source codes for subroutines `cross` and `mutate` will reveal three additional calls to function `urand()`. *This implies that even if starting from the same initial random population, from one generation to the next the two sequences of random deviates will become increasingly out of sync between version* `1.0` *and* `1.2`. This will lead to distinct evolutionary paths to the global optimum.

However, everything else being equal `PIKAIA 1.2` should be expected to be as performing as version `1.0` (and in fact probably more, as argued in the following chapter).

1.4 Obtaining a copy of PIKAIA 1.2

Source codes for `PIKAIA 1.2`, including the subroutine itself as well as driver programs and fitness functions for the examples discussed in chapter 5 of the PUG, can be obtained from NCAR's High Altitude Observatory via the Web or anonymous ftp. For direct Web access point your browser to

```
http://download.hao.ucar.edu/archive/pikaia
```

For download via anonymous ftp, use the address

```
ftp.hao.ucar.edu 122
```

and log in using the username `anonymous` and your e-mail address as the password (you are now on a UNIX system). Type the following command:

```
cd /archive/pikaia
```

The command `ls` can then be used to display the directory's content.

The file `README` is self-explanatory; please do read it, as it will most likely includes additional (and possibly important) information which did not make it in these release notes. The file `userguide.ps` is a standard postscript file containing the `PIKAIA 1.0` User's Guide (a.k.a. PUG in these release notes). Nearly everything in that User's Guide remains relevant to `PIKAIA 1.2`, so the novice `PIKAIA` user should definitely grab a copy. The file `pikaia.f` contains not only the genetic subroutine `PIKAIA 1.2` itself, but also a driver code, fitness function and other required routines, including a random number generator. The file is a completely self-contained source code which can be used for installation check (see PUG, §2.2). The original version `PIKAIA 1.0` remains available in the file `pikaia1.0.f`. The directory `examples` contains drivers, fitness functions and synthetic datasets for the examples discussed in chapter 5 of the PUG.

1.5 The PIKAIA Web Page

The PIKAIA Web page has been around since 1996, and can be found at the (lengthy) URL:

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

It is the place to look for Bug Reports, tutorial material, lists of known past and present users, and lists of PIKAIA-related publications. And of course, links are set up there to the most up-to-date, corrected versions of the PIKAIA User's Guide, the present Release Notes, and source code for all the software. You can also see there what a real *Pikaia* actually looks like.

1.6 Reference and credits

The GA-based optimization subroutine PIKAIA was first described in the following paper, published in *The Astrophysical Journal (Supplements)* in December 1995:

Charbonneau, P. 1995, ApJS, 101, 309.

and described in greater details in the PIKAIA User's Guide:

Charbonneau, P., & Knapp, B. 1996, *A User's Guide to PIKAIA 1.0*, NCAR Technical Note 418+IA (Boulder: National Center for Atmospheric Research)

The above two publications remain the primary reference for both PIKAIA. The present document shall serve as the "official" specific reference to version 1.2:

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

Both versions of PIKAIA remain public domain software, and so no restrictions are imposed on further distribution for research and education purposes.

2. A PERFORMANCE STUDY

The seasoned `PIKAIA` user satisfied with version `1.0` is entitled to ask: why bother switching to a new version? This is a very legitimate question[3], and this chapter makes an attempt at answering it.

2.1 Defining performance

In the computer science literature on search algorithms (which includes GAs), a distinction is often made between "off-line" and "on-line" performance. In the context of a GA, the "off-line" performance is the fitness of the very best solution found in the course of the whole evolutionary run. The "on-line" performance is a measure of how quick a good enough solution can be found. These two measures of performance are obviously related, and each can be more or less relevant depending on the problem under consideration. In the present case of global, multimodal numerical optimization using a GA, it is most important to focus on an aspect of off-line performance sometimes dubbed "global performance" or "success rate": given N runs of the GA on the same problem but starting with different initial populations, how many will succeed in finding the global maximum?

2.2 A suite of test functions

Readers with even just a little experience in global optimization will be well aware of the fact that global performance is highly problem-dependent. Consequently, the first thing that needs to be done is to assemble a suite of test problems that cover a broad range of optimization (mis)behaviors. Here all such test problems will take the form of maximization problems for functions of many (real) variables, i.e., finding the D-dimensional parameter set $\mathbf{x}^*$ such that the function

$$f(\mathbf{x}) \ , \qquad \mathbf{x} \equiv \{x_1, x_2, ..., x_D\}$$

is maximized. Following Bäck (1994), preference will be given to functions that are readily generalizable to arbitrary large D. In addition, we will require that

$$\max(f) \equiv f(x^*) = 1.0 \ , \qquad x_j \in [0.0, 1.0] \ , \quad j = 1, ..., D.$$

[3] I personally rank software upgrades very high on my most-hated list, up there with lawnmowing, suppertime telemarketers, and my neighbour's barking rat.

The first test function, plotted in its $D = 2$ version on Figure 3.1, is a simple, smooth unimodal function defined as a Gaussian centered in the middle of the spatial domain:

$$f_1(\mathbf{x}; D) = \exp(-r^2/\sigma^2), \tag{2.1a}$$

$$r^2 = \sum_{j=1}^{D} (x_j - 0.5)^2 \ , \tag{2.1b}$$

with $\sigma^2 = 0.15$. The maximum is at $x_j^* = 0.5\,\forall j$. This smooth, unimodal function is a very easy maximization problem, on which almost any local hill-climbing technique would be far more efficient than PIKAIA. It is nonetheless useful to have such a function in the test suite, in order to be able to test PIKAIA's local hill-climbing capabilities under various setting of its internal control parameters.

The second test function, plotted on Figure 2.1B again for $D = 2$, is unimodal like f_1, but step-wise discontinuous:

$$f_2(\mathbf{x}; D) = \frac{1}{D} \sum_{j=1}^{D} \frac{\mathrm{int}(10x_j - \epsilon)}{9} \ , \tag{2.2}$$

with $\epsilon = 10^{-6}$. The maximum is at $x_j \geq 0.9\,\forall j$. This is a variation on one of the test functions in the famous test suite developed by DeJong (1975) in his pioneering study of GA performance. This function is a useful test on two grounds: (1) search for a corner maximum is hard for the crossover operator; (2) only relatively large mutation can lead to fitness improvements. For small D, this function is easy for the GA, even though it would readily defeat most local gradient-based hill-climbing schemes, including the usually robust and pseudo-global downhill simplex method (see, e.g., Press *et al.* 1992, §X.Y). For large D, it becomes pretty hard even for a GA.

The third test function is the D-dimensional generalization of the multimodal 2D function distributed with PIKAIA as installation check. A plot of its $D = 2$ version is shown on Fig. 2.1C. Its D-dimensional form is defined as:

$$f_3(\mathbf{x}; D) = \cos^2(9\pi r) \exp(-r^2/\sigma^2), \tag{2.3a}$$

$$r^2 = \sum_{j=1}^{D} (x_j - 0.5)^2 \ , \tag{2.3b}$$

with again $\sigma^2 = 0.15$. The maximum is at $x_j = 0.5\,\forall j$, but is surrounded by six concentric rings of progressively higher secondary maxima centered on the central

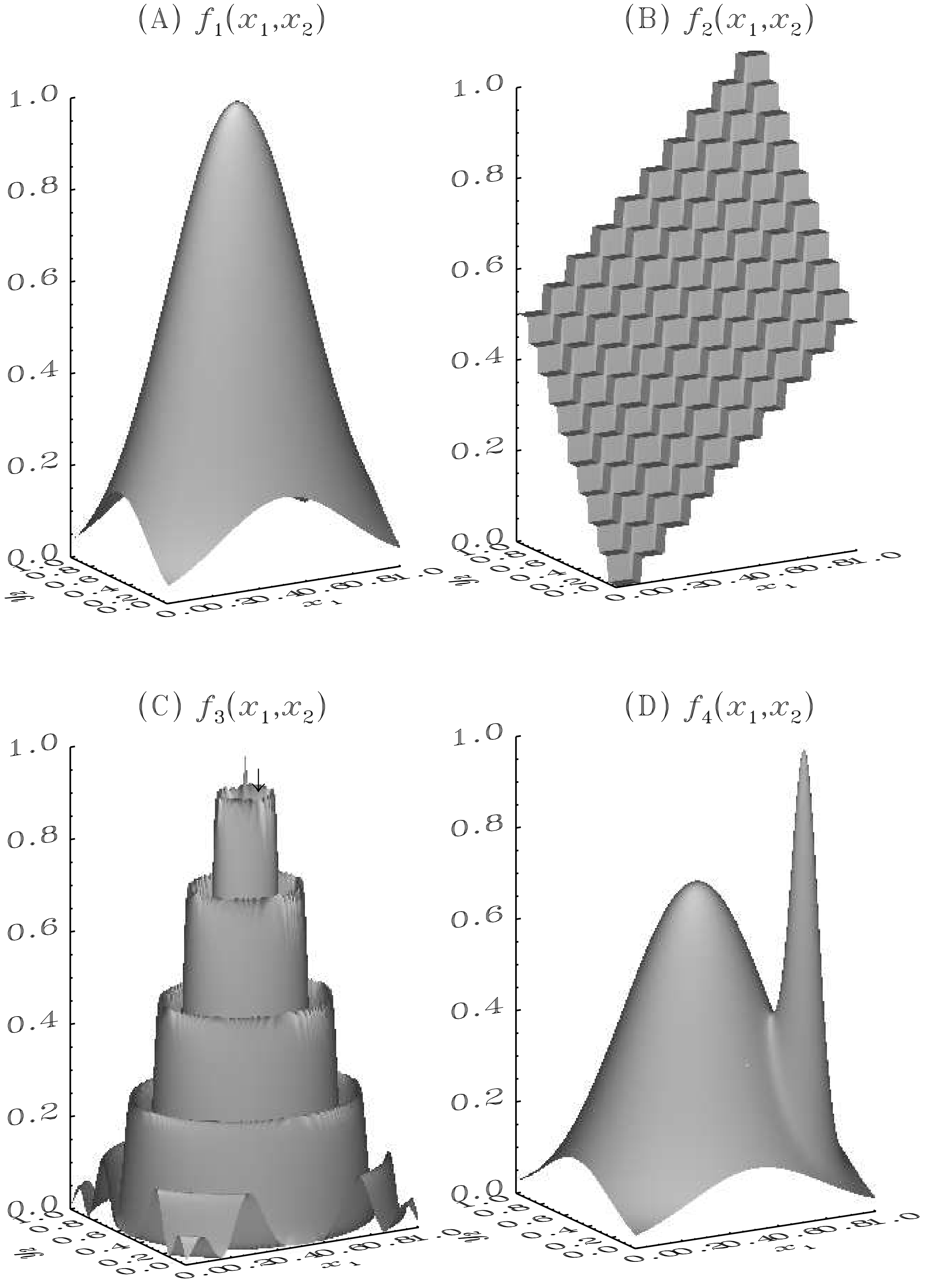

Figure 2.1: A suite of test functions. The plots show the two dimensional ($D = 2$) versions of the four functions that make up PIKAIA's test suite. The fourth test function has been rotated by 90° about the vertical, to facilitate viewing.

maximum. This is a very hard test function by any standards, and it rapidly gets a lot harder as D increases.

The fourth and final test function is the sum of two D-dimensional Gaussians, as shown on Figure 2.1D (for $D = 2$ as usual). It's general form is:

$$f_4(\mathbf{x}; D) = A_1 \exp(-r_1^2/\sigma_1^2) + A_2 \exp(-r_2^2/\sigma_2^2) \ , \tag{2.4a}$$

$$r_1^2 = \sum_{j=1}^{D} (x_j - 0.5)^2 \ , \qquad r_2^2 = \sum_{j=1}^{D} (x_j - 0.2)^2 \ , \tag{2.4b}$$

with

$$A_1 = 0.7, \qquad A_2 = 1 - 0.7 \exp(-r_1^2/\sigma_1^2) \ , \tag{2.4c}$$

and $\sigma_1^2 = 0.15$, $\sigma_2^2 = 0.005$. This is a *very* hard problem, and increasingly so the larger D gets. Almost invariably, early in the evolutionary run the central, broad lower Gaussian draws towards it most trial solutions. The challenge is then for the prematurely converged population to find and move onto the taller, narrower Gaussian which is the true, global maximum.

2.3 Experimental design

The idea here is primarily to examine the improvement (or degradation) in performance associated with the new operators and strategies included in `PIKAIA 1.2`, namely distance-based mutation rate adjustment, two-point crossover, and creep mutation. This is in fact impossible to establish in a robust, problem-independent manner, since the setting of the various GA control variables interact with each other in complex ways, with resulting very non-linear effects on performance. In other words, examining performance as one control parameter at a time is varied is useful, but care is warranted in making grand sweeping conclusions on the basis of such experiments. With this important caveat firmly in mind, we proceed nonetheless.

We will test six distinct implementations of `PIKAIA 1.2`, labeled very originally PK1 through PK6. All implementations use a population size `np`=`ctrl(1)`= 50, 5-digit encoding `nd`=`ctrl(3)`= 5, crossover probability `pcross`=`ctrl(4)`= 0.85, full-generational-replacement `irep`=`ctrl(10)`= 1, elitism `ielite`=`ctrl(11)`= 1, full pressure `fdif`=`ctrl(9)`= 1.0, and variable mutation rate in the range $0.0005 \leq p_m \leq 0.25$, initially set at $p_m = 0.005$ (as specified by `ctrl(6)`, `ctrl(7)`, and `ctrl(8)`). The six implementations differ in whether or not they use two-point crossover, creep mutation, and whether they adjust the mutation rate according to the fitness-based or distance-based criterion (eq. (2.1) or (2.2) herein). Note that in this context PK1 corresponds to the default settings on `PIKAIA 1.0`.

2.4 Off-line performance Results

To get interesting and meaningful numbers in a reasonable amount of CPU time, we'll pick $D = 10$ and 15 for f_1 and f_2, but $D = 3$ and 4 for the much harder f_3 and f_4. Likewise, we'll only run out to 100 generations with the first two, but up to 2500 generation with the latter pair. To get representative statistics, each `PIKAIA` implementations listed in Table II is run 100 times, each time with distinct initial random seeds, and thus with different initial random population.

The results are listed in Table II. The top part of the Table details which operators and adjustment strategy are used in which of the six `PIKAIA` implementation. Remember that a "yes" on two-point crossover or creep mutation means that these are used 50% of the time, with one-point crossover and uniform one-point mutation used for the other 50%. For the unimodal f_1 and f_2, the numerical entries correspond to the 100-run average of the best individual found at the end of each run. On the other hand, for f_3 and f_4 the entries give the success rate, e.g., 58/100 means that the global maximum was found in 58 out of 100 independent trials. This is by far the most relevant performance measure for these hard multimodal functions.

Table II

Off-line Performance results

	PK1	PK2	PK3	PK4	PK5	PK6
2-pt. cross.	No	Yes	No	Yes	Yes	Yes
Creep	No	No	No	No	Yes	Yes
p_m-adj.	fitn.	fitn.	dist.	dist.	fitn.	dist.
$f_1(D = 10)$	0.9921	0.9929	0.8558	0.8717	0.9893	0.9232
$f_1(D = 15)$	0.9668	0.9714	0.5118	0.5401	0.9656	0.7408
$f_2(D = 10)$	0.9906	0.9912	0.9929	0.9949	0.9983	0.9984
$f_2(D = 15)$	0.9586	0.9629	0.9542	0.9544	0.9813	0.9822
$f_3(D = 3)$	43/100	44/100	64/100	66/100	91/100	100/100
$f_3(D = 4)$	37/100	25/100	7/100	10/100	27/100	27/100
$f_4(D = 3)$	93/100	93/100	70/100	100/100	67/100	58/100
$f_4(D = 4)$	2/100	3/100	14/100	14/100	2/100	2/100

There is a lot of information, trends, and pseudo-trends that can be extracted from Table II. The most obvious trend is that performance rapidly degrades as problem dimensionality (D) is increased (for constant population size and number of generation). Perhaps the most noteworthy other general observation is that:

(1) Two-point crossover often helps, and very rarely hurts (cf. PK1 vs PK2, and PK3 vs PK4, on all f's)

Other trends in Table II are harder to interpret, or just plain confusing or apparently inconsistent. As a small selection, consider:

(2) Creep mutation helps a lot on some problems (PK2 vs PK5 and PK3 vs PK6 on f_3), but degrades it on others (cf. f_4)
(3) Creep mutation degrades performance on f_1 when used with fitness-based mutation rate adjustment (PK2 vs PK5), but improves it when used with distance-based adjustment (PK4 vs PK6).
(4) PK3 did worst on $f_3(D = 4)$, but best (ex aequo with PK4) on $f_4(D = 4)$
(5) On f_3, PK1 did worst at $D = 3$, but degraded the least —and did the best— at $D = 4$.

The sources of these apparent inconsistencies are of course that (1) performance is highly problem-dependent, and (2) The GA's internal strategies and operator interact in very complex and nonlinear ways.

It is on the basis of these —and other— numerical experiments that the decision was made to hardwire 50% two-point crossover in `PIKAIA 1.2`, but to leave the choice of creep mutation and distance- vs fitness-based mutation rate adjustment in the hand of the user, through the setting of the control variable `imut` ($\equiv$`ctrl(5)`), as described in Table I herein.

2.5 On-line performance Results

On-line performance, i.e., how fast is an "acceptable" solution found, is mostly an issue in applications relating to real-time control, and is rarely critical in the types of numerical global optimization modeling applications that most `PIKAIA` users are likely to find themselves involved with. Nonetheless, a quick look at on-line performance on our suite of test functions is useful to appreciate various behavioral differences in the six $\widehat{\text{P}}$IKAIA versions used to generate the off-line performance results of table II.

Figure 2.2 depicts average convergence curves for two of the test problems of Table II, namely $f_1(D = 15)$ and $f_2(D = 10)$. Each curve corresponds to the average of 100 "best fitness vs generation" convergence curves, and so can be expected to be statistically representative.

Consider first Figure 2.2A, for the unimodal f_1 test function. Initially (first 15 generations, say), all `PIKAIA` versions show more or less the same rapid increase

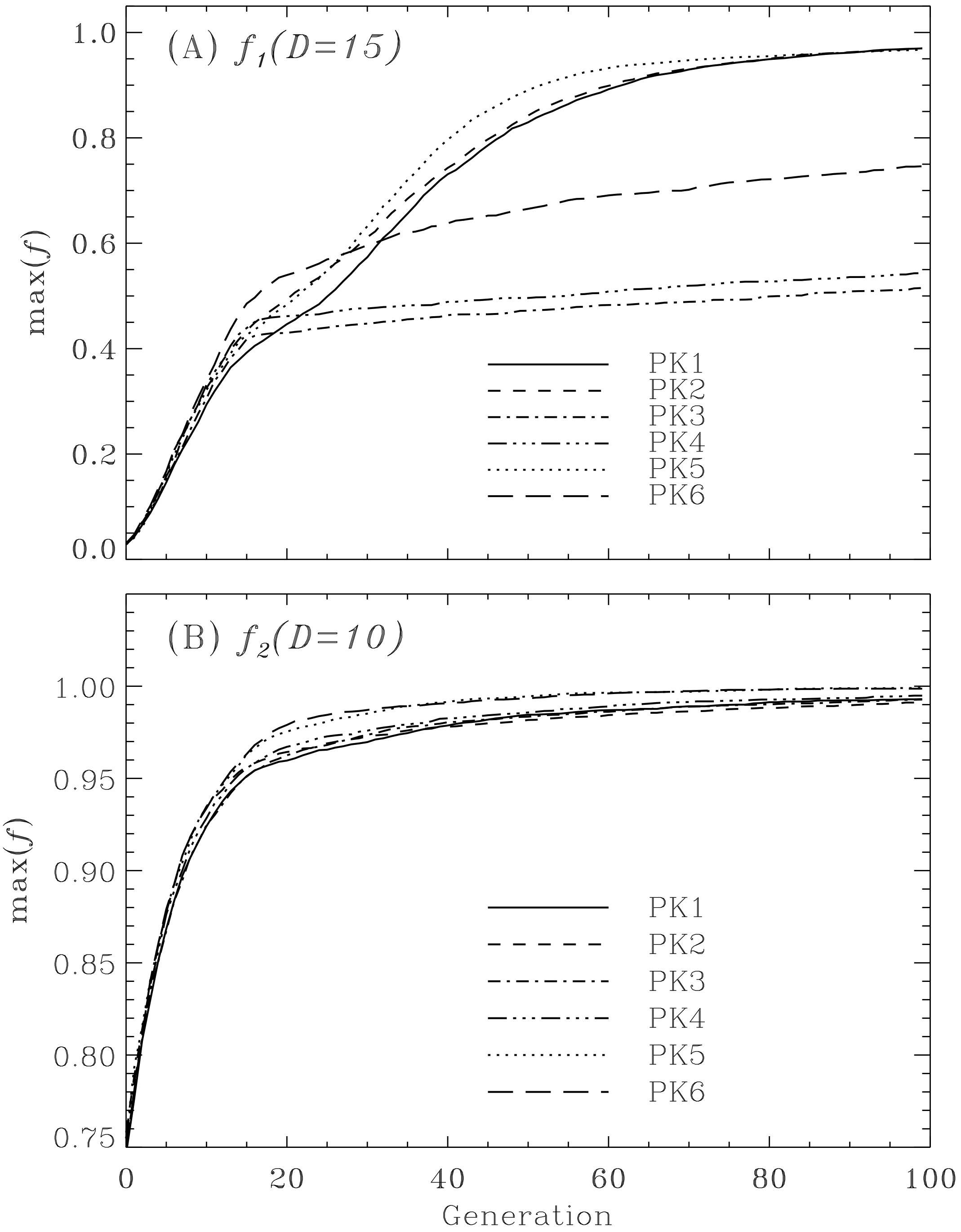

Figure 2.2: On-line performance results on $f_1(D = 15)$ and $f_2(D = 10)$. Note the different vertical scales on each panel.

in fitness, which is here primarily due to the exploratory action of the crossover operator. Then a marked divergence can be seen between the algorithms using fitness-based adjustment of the mutation probability (PK1, PK2, and PK5) and those using distance-based adjustment (PK2, PK4, and PK6). This reflects the fact that f_1 is a sharply-peaked function, for which a fitness-based criterion (eq. (1.1)) provides a measure of population clustering superior to that computed using a distance-based criterion. Note how the use of creep mutation (PK2 vs PK5) makes little difference in the former case, but a significant one in the latter (PK4 vs PK6). Here the use of two-point crossover seems to make little difference in either case (cf. PK1 vs PK2, and PK3 vs PK4).

Results for test function f_2, shown on Fig. 2.2B, show a totally different pattern, with now PK5 and PK6 coming out slightly ahead, and all four other implementations doing more-or-less the same. This indicates a total lack of dependence on the criterion used to measure population clustering and control the mutation rate. This is a natural consequence of the fact that the unimodal function f_2 is globally linear in each parameter (see Fig. 2.1B), so that there exists a linear relationship between fitness difference and metric distance. The superior on-line performance of PK5 and PK6 indicates the enhanced local hill-climbing capabilities of creep mutation, over uniform one-point mutation.

On-line results for the multimodal f_3 and f_4 (not shown) show no obvious trends. All versions do about equally well over the first 50–100 generations, following which the convergence curves mostly reflect the success rate at locating the true extrema. On $f_3(D = 3)$, PK4 does marginally better early on, with PK5 and PK6 taking a slight lead from generation 100 onward, and ultimately achieving the best off-line performance (see Table II).

3. SELECTED SPECIAL TOPICS

This chapter discusses three special topics that, although not specifically related to `PIKAIA 1.2`, might be nonetheless of interest to the `PIKAIA` user community.

3.1 `PIKAIA` in other computing languages

3.1.1 PIKAIA in IDL

Sarah's IDL version; Scott's version; Gordon Chin's version;
`http://zeus.nascom.nasa.gov/ scott/ga.html`

3.1.2 PIKAIA in f90

Alan Miller, a Honorary Research Fellow at the CSIRO/Mathematical and Information Sciences in Australia, has produced a FORTRAN-90 version of `PIKAIA 1.0`, which he kindly makes available at the following Web site:

`http://users.bigpond.net.au/amiller`

and see under "Miscellaneus code".

3.2 `PIKAIA` for combinatorial optimization

This section describes the use of *PIKAIA* for a combinatorial optimization problem. It is based on a chapter of the PhD thesis of Scott McIntosh, then a graduate student at the University of Glasgow, Scotland.

The physical problem Scott was working on involved inferring physical properties of the solar outer atmosphere, using measurements of the intensities I_l of various spectral lines observed by space-borne solar telescopes (see McIntosh *et al.* 2000). Mathematically, the problem is defined by an integral equation of the form

$$I_m = \int_T K_m(T)\xi(T)\mathrm{d}T \ , \qquad m = 1, ..., M \ . \tag{3.1}$$

Here the quantity $\xi(T)$, known as the *differential emission measure*, is the quantity to be inverted from the M observed lines and is assumed to depend only on the temperature T of the solar atmospheric plasma. The Kernel $K_m(T)$ is a known quantity involving a lot of atomic physics, as well as properties of the observing instrument. Discretizing K_m and ξ on a temperature mesh of size J, and introducing a suitable quadrature formula for the integral, turns eq. (3.1) into the matrix system:

$$[K]\{\xi\} = \{I\} \ , \tag{3.2}$$

where the matrix $[K]$ is of size $M \times J$. In practice, even if one chooses $J = M$ the linear system described by eq. (3.2) is ill-conditioned. This occurs because of the smoothing properties of the integral operator in eq. (3.1), and is compounded by the fact that the Kernels K_m are not localized enough in T to avoid overlap (see Craig & Brown 1986 for an accessible introduction to these matters).

The technique of choice for dealing with eq. (3.2) is of course Singular Value Decomposition (see Press *et al.* 1992 for a lucid, no-nonsense introduction). However, Scott was interested in a method that would allow him to simply select a subset of lines I_m that would yield the least ill-conditioned version of eq. (3.2), while allowing additional, physically-based constraints to be imposed on line choices. The reduction in dataset size was also appealing to him, because of the massive amount of data produced by current solar spectral imaging instruments (one spectrum per pixel, at a high temporal cadence). Already familiar with `PIKAIA` from a prior modeling study (McIntosh *et al.* 1998), Scott decided to modify it to tackle his line selection problem.

3.2.1 Modifying PIKAIA's encoding scheme

What Scott was facing is a combinatorial optimization problem than involves identifying the subset of N *distinct* labels from a search list of M $(> N)$ possible choices, so as to minimize some computable quantity, in his case the Condition Number of the matrix $[K]$ in eq. (3.2). An important simplifying aspect of this combinatorial problem is that the *ordering* of selected elements within the optimal subset is unimportant.

The only non-trivial modification to `PIKAIA` involved the encoding scheme. For Scott's problem, an individual is defined as a list of N integers with values $1 \leq n \leq M$ defining with of the M available spectral lines measurements are to be used to carry out the inversion, as opposed to `PIKAIA`'s floating point array `x(1:n)`. The user-supplied fitness function `ff(n,x)` must then accept as input one such integer array, and compute the Condition Number of the resulting matrix system (now of size $N \times J$) given by eq. (3.2). The primary difficulty was to devise an encoding scheme that always ensures that the action of crossover and mutation produces offspring lists of labels that always contain *distinct* entries.

3.2.2 Standard Ordinal Representation

The construction of a list of distinct elements from a reference list with possible non-distinct entries is readily carried out using the *ordinal representation* scheme (Michalewicz 1994, §X.Y). The task is to extract N distinct labels from a master list $\mathbf{S}$ of M possible values, where $S_m = m$, $m = 1, ..., M$. An *ordinal vector* $\mathbf{e}$ is made up of N elements (e_n) with values in the range $1 \leq e_n \leq M - n + 1$. The corresponding *element vector* $\mathbf{E}$ is constructed according to the following iterative procedure:

do $n = 1, N$

$E_n := S_{e_n}$

$S_k := S_{k+1}, \qquad k = n, ..., M - n - 1$

enddo

The second algorithmic step has the consequence that S_{e_n} is removed from the list, and the list size is reduced by one at each iteration. This ordinal representation scheme has some attractive characteristics. It is quite simple to implement, and having the e_j's uniformly distributed in their allowed bounds results in a uniform distribution of E_j's. However, one can easily verify that when pairs of $\mathbf{e}$ are acted upon by the uniform one-point crossover operator, the e_j's located right of the splicing point can decode into E_j's not originally coded by the parent $\mathbf{e}$'s. Consider for example a situation where $N = 10$ distinct elements must be extracted from a reference list of $M = 40$ possible values. The following two ordinal vectors:

$$\mathbf{e}_1 = (4, 5, 1, 29, 26, 11, 31, 8, 22, 5) \tag{3.3a}$$

$$\mathbf{e}_2 = (3, 5, 35, 36, 8, 17, 19, 8, 23, 1) \tag{3.3b}$$

decode into the two lists

$$\mathbf{E}_1 = (4, 6, 1, 32, 29, 14, 37, 11, 27, 8) \tag{3.4b}$$

$$\mathbf{E}_2 = (3, 6, 37, 39, 10, 20, 23, 11, 29, 1) \tag{3.4b}$$

Now carry our crossover of the two ordinal vectors (3.3a), (3.3b), with a splicing point at the third element, yielding the offspring ordinal vectors:

$$\mathbf{e}_3 = (4, 5, 1, 36, 8, 17, 19, 8, 23, 1) \tag{3.5a}$$

$$\mathbf{e}_4 = (3, 5, 35, 29, 26, 11, 31, 8, 22, 5) \tag{3.5b}$$

These decode into the two offspring lists:

$$\mathbf{E}_3 = (4, 6, 1, 39, 11, 21, 24, 12, 30, 2) \tag{3.6a}$$

$$\mathbf{E}_4 = (3, 6, 37, 31, 28, 13, 36, 19, 26, 7) \tag{3.6b}$$

Note that the offsprings lists now contain many labels that were not present in either of the parent lists (3.5a), (3.5b). This difficulty is a direct consequence of the leftward shifting associated with the encoding procedure[4]. This is conceptually at variance with the expected behavior of crossover, which should lead to exchange of intact subset of the labels E_j's. The only tolerable exception is when the crossover operation introduce additional duplicate entries in the pair of $\mathbf{e}$ resulting from the crossover operation. Likewise, under the ordinal representation uniform one-point mutation can potentially alter *all* elements of $\mathbf{E}$. This made the standard ordinal representation unsuitable for Scott's application.

3.2.3 Ranked Ordinal Representation

Scott and I came up with a simple modification of standard ordinal representation, which we dubbed Ranked Ordinal Representation (ROR) that can bypass the above problem. It consists in *ranking* the ordinal vector in decreasing order (so that $e_{n+1} \leq e_n$) prior to applying the ordinal algorithm. Under this scheme the ordinal vector (3.2a) now decodes into

$$(31, 29, 26, 22, 11, 8, 5, 6, 4, 1) \tag{3.7}$$

Clearly, if $\mathbf{e}$ does not contain duplicate entries then $\mathbf{E} = \mathbf{e}$ (with $\mathbf{e}$ ranked). If however entry $e_n = e_{n+1}$ for some n $(< N)$, then $E_n = e_n$ and $E_{n+1} = e_n + 1$.

This modified scheme behaves far better under one-point crossover. Under ROR, the two parent ordinal vectors $\mathbf{e}_1, \mathbf{e}_2$ (eqs. (3.3a,b) above) now decode into:

$$\mathbf{E}_1 = (31, 29, 26, 22, 11, 8, 5, 6, 4, 1) \tag{3.8a}$$

$$\mathbf{E}_2 = (36, 35, 23, 19, 17, 8, 9, 5, 3, 1) \tag{3.8b}$$

and the resulting two offspring ordinal vectors $\mathbf{e}_3, \mathbf{e}_4$ (eqs. (3.6a,b) above) then decode into:

$$\mathbf{E}_3 = (36, 23, 19, 17, 8, 9, 5, 4, 1, 2) \tag{3.9a}$$

$$\mathbf{E}_4 = (35, 31, 29, 26, 22, 11, 8, 5, 6, 3) \tag{3.9b}$$

Clearly the crossover operation has had the desired effect, namely the interchange of a subset of E_n's from each parent $\mathbf{E}$ (remember that element ordering within

[4] Note in fact that $\mathbf{E}_3 = \mathbf{E}_2 + 1,\ \forall n \geq 5$, Likewise, $\mathbf{E}_4 = \mathbf{E}_1 - 1,\ \forall n \geq 5$ except $n = 8$ where a duplicate entry caused an additional shift.

each **E** is irrelevant here). Similarly, one-point mutation has now only a local effect.

To ensure that a statistically uniform distribution of e_n's translates into a similarly uniform distribution of E_n's, it is necessary to let $1 \leq e_n \leq M, \forall n$. This, in turns, requires that $S_{e_n} \leq M - n + 1$; to achieve this we modify the algorithm in the following way:

do $n = 1, N$

$E_n := \min(M - n + 1, S_{e_n})$

$S_k := S_{k+1}, \qquad k = n, ..., M - n - 1$

enddo

This actually introduces a slight bias toward high values of E_n, but for relatively small population sizes ($N_p \leq 100$, say) it remains statistically insignificant as compared to to the realization noise ($\propto \sqrt{N_p}$).

A more serious problem arises when crossover is repeatedly applied to *randomly* selected members of an initially randomly distributed population of **e**'s, when N_p is small. Because the occurrence of duplicates (e_n, e_n)'s in newly generated offspring always translates into a pair (E_n, E_{n+1}); in the absence of mutation, any inhomogeneity in the original distribution of e_n's across the population (inevitable for finite population size) will tend to grow until the whole population is clustered around N allowed values for the search list.

This is illustrated in Figure 3.1, showing the original distribution of S_m's corresponding to a population of $N_p = 50$ iterating over as little as 10 generations, for $N = 30$ and $M = 133$. Panel A shows the original distribution of S_m's (thick histogram), and the resulting distribution after 10 iterations (thin histogram). The dotted line is the expected average count for each S_m, and the dashed lines indicate the standard deviation expected from Poisson counting statistics. Panel B is similar in format, but now mutation (with probability of occurrence of $p_m = 0.02$ for each e_n) also included. Mutation reduces, but does not eliminate, the clustering instability.

3.3 Parallel `PIKAIA`

It is often said that genetic algorithms readily lend themselves to a parallel implementation, because fitness evaluation, usually the most computationally demanding step, can be carried out independently for each individual in the population. This seems obvious, but to develop an actual working parallel genetic algorithm implementation is quite another matter. In this section I give an overview of one specific parallel application of `PIKAIA` which I found both elegant and impressive.

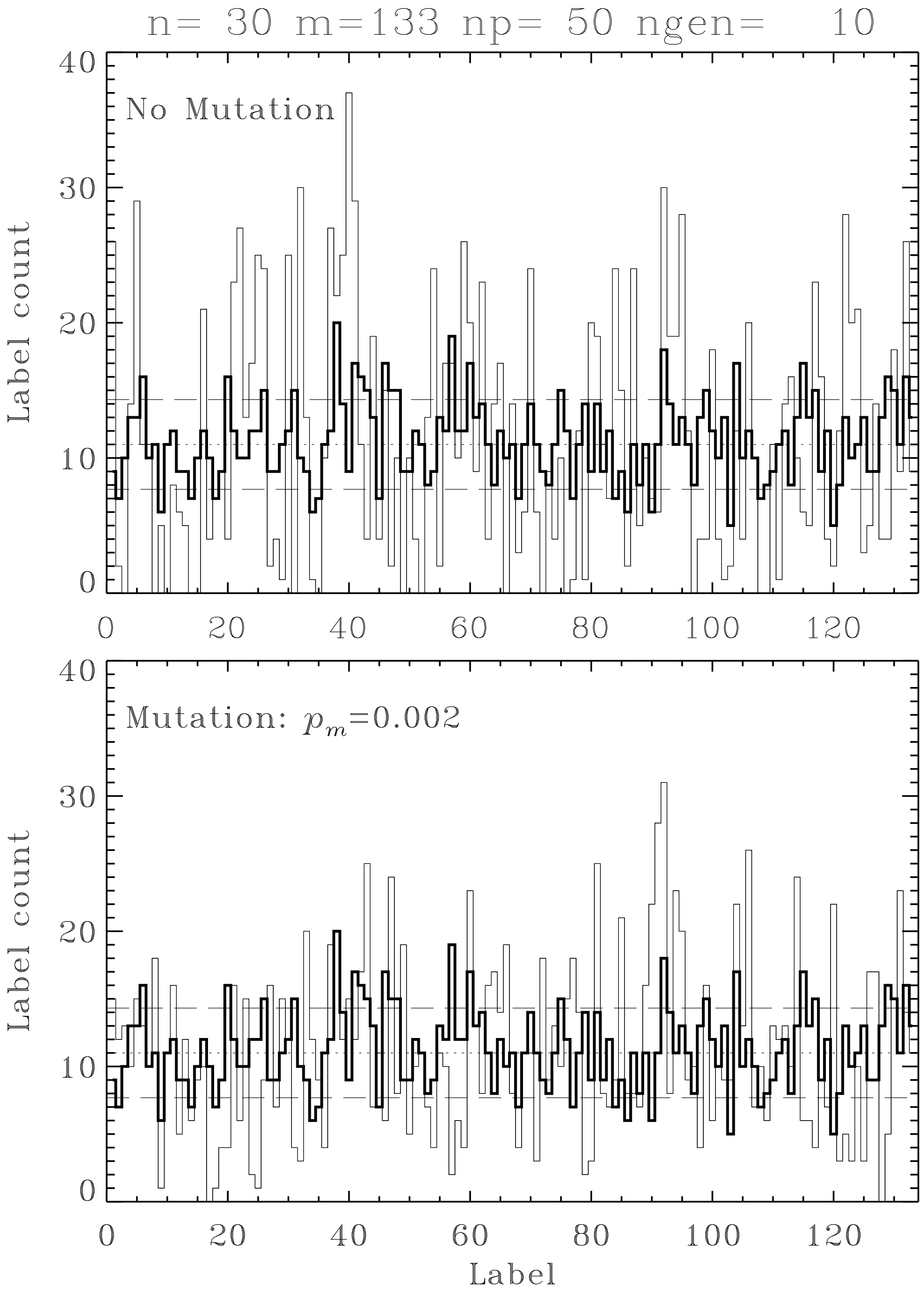

Figure 3.1: Clustering instability for ROR without selection pressure.a The thick-line histograms show the frequency distribution of selected lines in the initial random population, and the thin histograms that having arisen after application of the crossover operator with random selection for ten generations, with (panel B) and without (panel A) use of the mutation operator. The horizontal dotted line is the expected mean count fori purely random selection, with the dashed lines indicating the one-sigma range expected from Poisson counting statistics.

Travis Metcalfe, then a graduate student at the University of Texas at Austin, was facing a tough modelling problem in stellar structure theory in the context of his PhD thesis work: what can be learned about the physics of white dwarf interiors from astronomical observations of their pulsation frequencies (for a great introduction to the physics of white dwarf interior, see Fontaine *et al.* 2001). White dwarfs are the ultimate enpoints of stellar evolution for the vast majority of stars, including our Sun. They have exhausted all their nuclear fuel, and gradually cool and fade to black by radiating their thermal energy content into space. Their interior structure is a historical record of their life as thermonuclear furnaces. Travis was trying to obtain a best-fit to pulsation frequencies of one particularly well-observed white dwarfs, matriculated GD358. To do so he had to (1) pick a set of global stellar parameters such as mass, internal chemical composition, etc.; (2) evolve a stellar model from the beginning of the white dwarf phase to the observed surface temperature of GD358; (3) solve an eigenvalue problem for the pulsation frequency of his cooled model; and finally (4) compute the root-mean-square residual between these theoretical frequencies and the observed ones. Finding the set of stellar parameters that minimizes this residual defined the optimization problem facing Travis. For a given parameter set, the above sequence of modeling steps added up to about a minute of CPU time on a typical fast workstation processor. Faced with a vast and likely multimodal parameter space, it rapidly became clear to Travis that he needed an efficient global optimization scheme that could be parallelized. He wisely opted for genetic algorithms, and settled on `PIKAIA` as a specific software.

Travis' next judicious move was to decide that if he was going to go parallel, he might as well go all the way. He managed to obtain no less than 64 minimal "slave" PCs, all of which he connected to a "master" PC via a local network operating under LINUX (see Metcalfe & Nather 1999). With executables of the stellar evolution and pulsation codes as well as observed frequency data copied onto each of the slave PC's, Travis ran a slightly modified version[5] of an early pre-release incarnation of `PIKAIA 1.2` on the master PC, using the higher-level PVM language to manage the communications between master and slaves. All

[5] This version included distance-based mutation rate adjustment and creep mutation. The former lead to significant performance improvement for Travis' modeling problem, but the latter did not. Here distance-based mutation rate adjustment helped a lot because Travis was dealing with a low contrast parameter space; his best solutions did not have a fitness value (inverse residual) differing a whole lot from moderately good or even so-so solutions. The difference was however physically meaningful, given the high accuracy of the pulsation frequencies observed in white dwarfs.

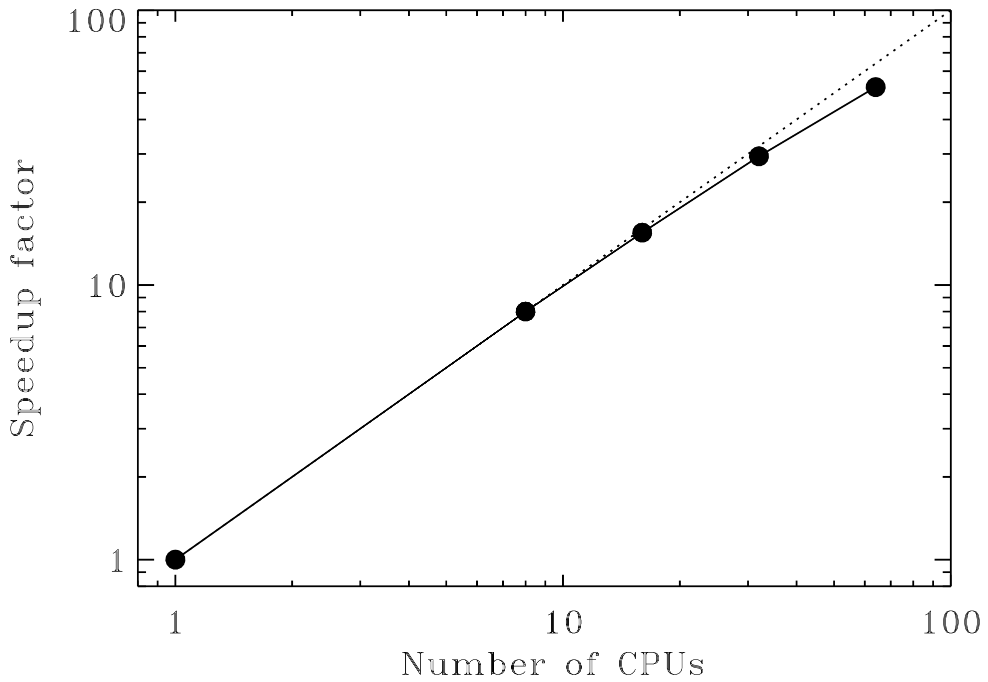

Figure 3.2: Speedup factor for parallel `PIKAIA`. The curve shows the speedup factor (inverse wallclock time) achieved by Travis' parallel implementation of `PIKAIA` on his fully distributed hardware architecture, as a function of the number of processor used in the calculation. The theoretical maximum, speedup $\propto$ number of processors, is shown by the dotted line.

the crucial little details are included in his PhD thesis, which is available online at

`http://ceti.as.utexas.edu/metcalfe/`

(see in particular his Chapter 3 and Appendix C). Travis went on to reap a rich scientific harvest from his customized hardware/software system (see Metcalfe *et al.* 2000; Metcalfe *et al.* 2001; Metcalfe & Charbonneau 2002). From a computational point of view, one truly remarkable thing was the speedup factor achieved using his fully distributed hardware and software design. Figure 3.2 shows a scalability plot going from one to 64 slave processors. The speedup factor reaches 52.7 at 64 processor. And perhaps even more remarkably, the scaling curve shows little sign of leveling off. Scaling up a parallel software very rarely looks that good. Figure 3.2 stands as a testimony of the truly robust parallelization properties of genetic algorithms.

As far as `PIKAIA` itself is concerned, Travis actually had very little to do, once he decided to restrict his reproduction plan to full-generational-replacement (`irep`≡`ctrl(10)=1`). In `PIKAIA` running under full-generational-replacement, at each generation the offspring solutions are bred and stored in the array `newph`, and only after the breeding loop terminates are the fitness calculated in a **do** loop within subroutine `newpop` (see PUG, §§3.2 and A.1). Travis pulled out the call to the user-supplied fitness function `ff(x,n)` from the `do 2 i=1,np` loop in `newpop`, and inserted a single call to a new subroutine that evaluate the fitnesses of population members in parallel (a similar modification was made to the `do 1 ip=1,np` loop in the main subroutine, to compute the fitnesses across the initial random population).

4. ON NUMERICAL OPTIMIZATION

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. 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 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 Figures 4.1(A) through (D), 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 (Fig. 4.1 [A]–[B]). One then determines the local steepest uphill direction, moves a certain distance in that direction (Fig. 4.1 [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 (Fig. 4.1 [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 on which the rabid paratrooper of Fig. 4.1(A) is about to deposit himself. Unfortunately, life is not always that simple. Consider instead the 2-D landscape shown on Figure 4.2; 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 4.2 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? [6] Your answer to this question is embodied in the First Rule of Global Optimization, also known as

> THE DIRTY HARRY RULE:
>
> "You should never feel lucky"

Faced with the landscape of Figure 4.2 and without a good starting guess in hand, the most straightforward solution lies with a technique called *iterated hill climbing*.

[6] Well, do you, punk?

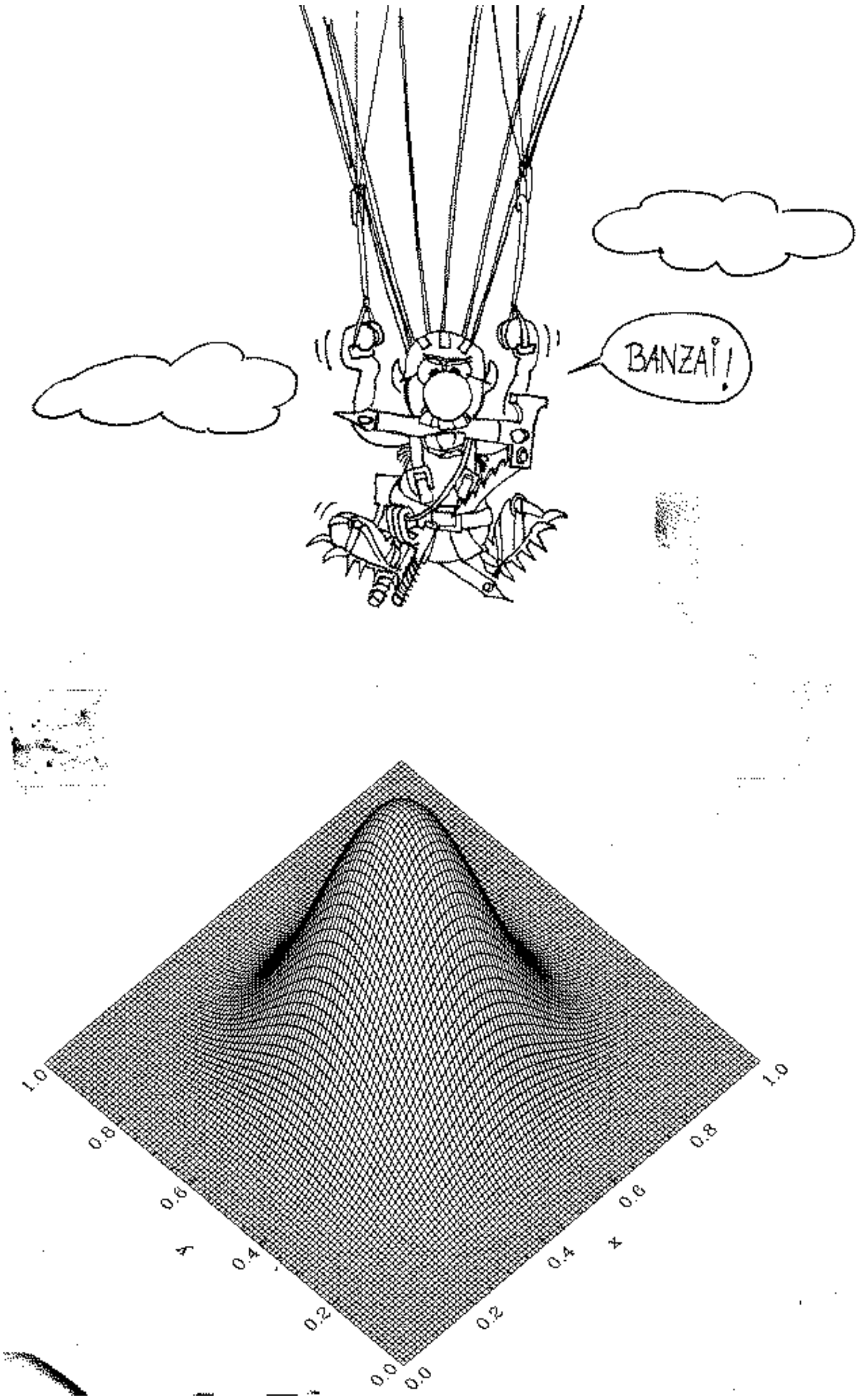

Figure 4.1(A): A hill-climbing optimization scheme. I. Initialization.

Figure 4.1(B): A hill-climbing optimization scheme. II End of Initialization.

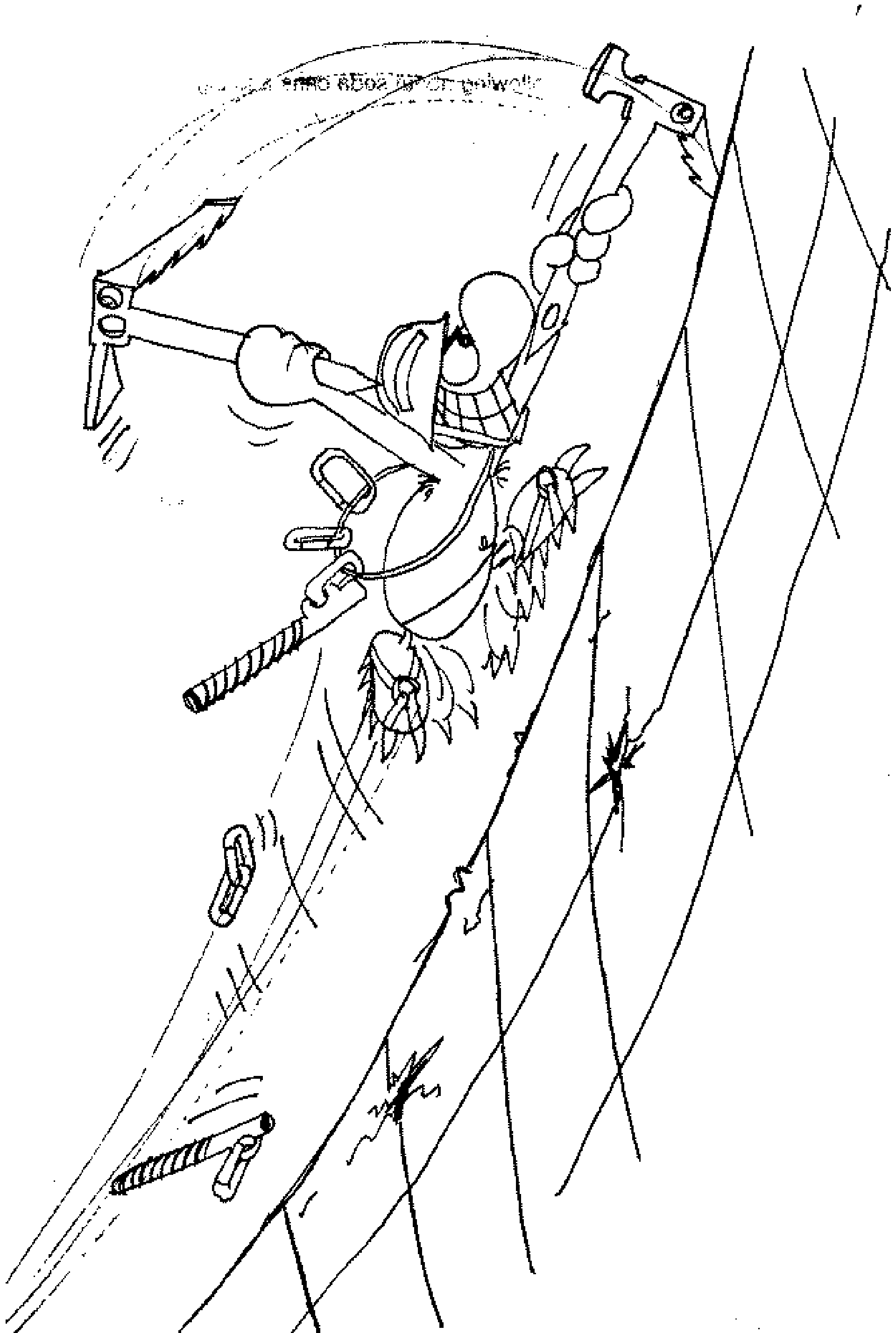

Figure 4.1(C): A hill-climbing optimization scheme. III Upward Iteration.

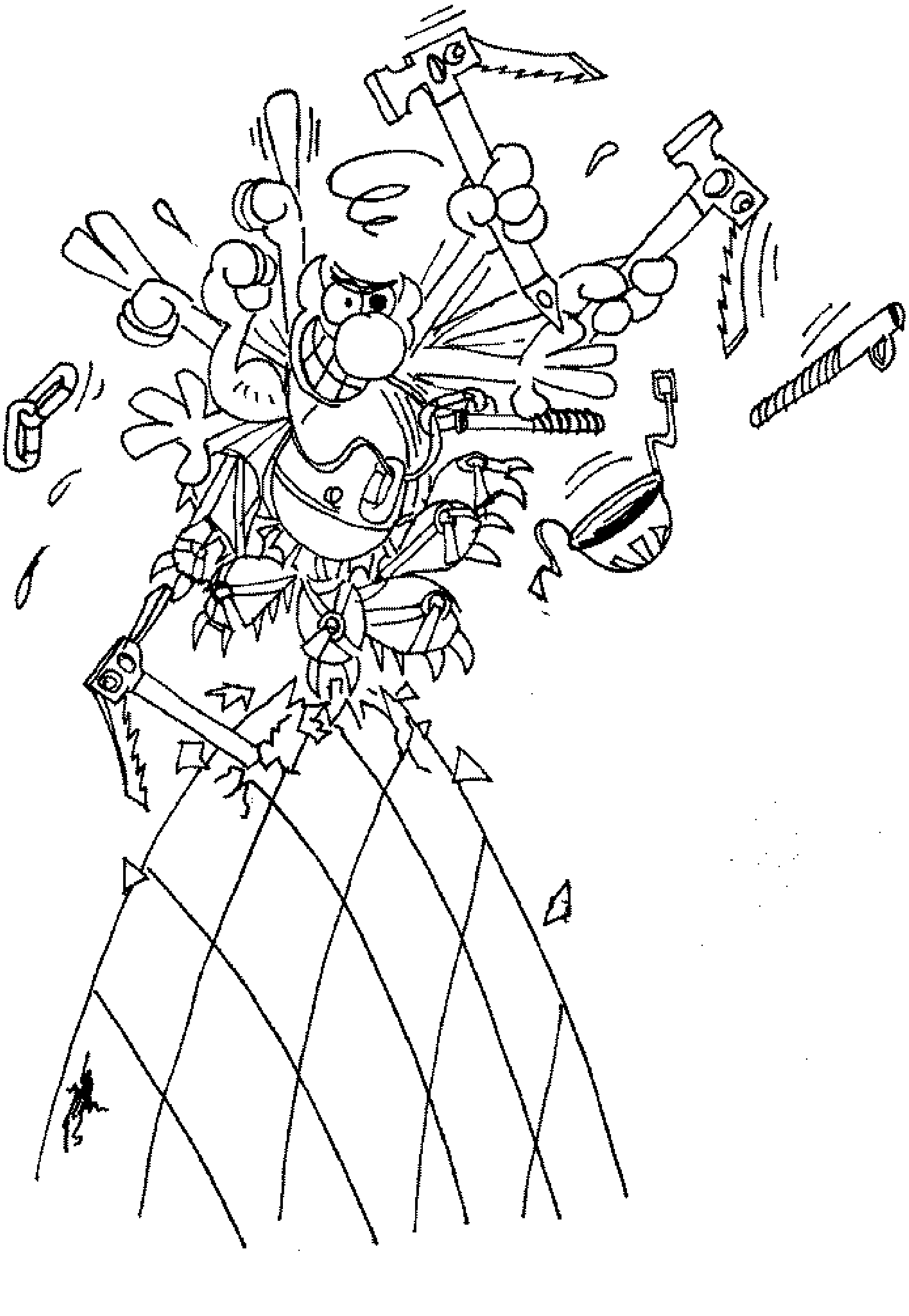

Figure 4.1(D): A hill-climbing optimization scheme. IV. Termination.

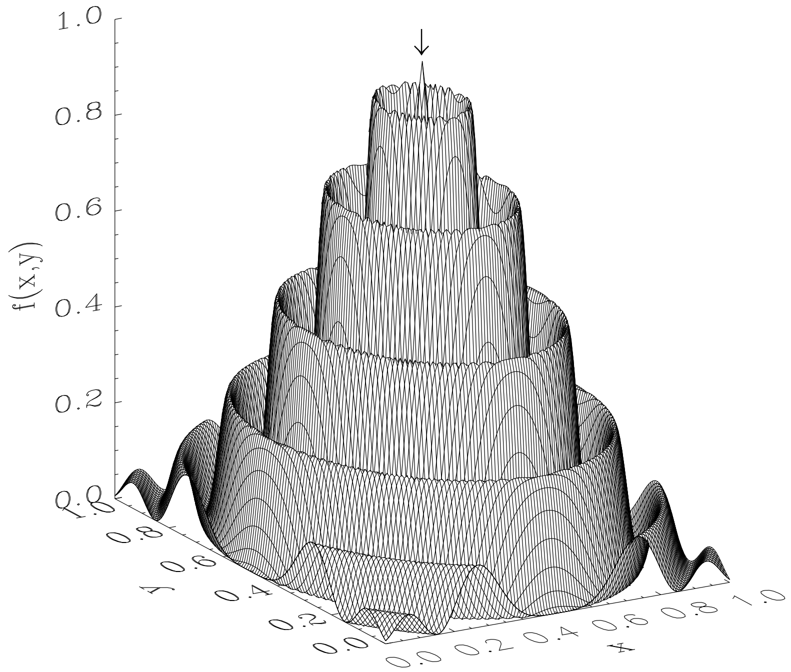

Figure 4.2: A multimodal optimization problem. The global maximum is $f(x, y) = 1$ at $(x, y) = (0.5, 0.5)$ is indicated by the arrow.

This is a fancy name for something very simple, as illustrated on Figure 4.3. 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 well imagine, deciding when to stop is the crux of this otherwise straightforward procedure.

In the case of the 2D landscape of Figure 4.2, the fraction of the 2D domain area corresponding to landing sites from which local hill-climbing would lead to the true, central maximum is about 1%. Consequently, 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. Of course, this kind of information is rarely available *a priori*. Even if it were, 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

Figure 4.3: An iterated hill-climbing optimization scheme.

THE NO FREE LUNCH RULE:

"If you really want the global optimum, you will have to work for it"

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 climbing run. The challenge, of course is to achieve this without overly biasing the ensemble of trials. And this, of course, is precisely what genetic algorithms do.

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. 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, 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). 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.

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 a genetic algorithm until no improvement is made to the best individual in some preset number of generation, and then using this individual to initialize your favorite local hill climbing scheme. 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 most efficient use of genetic algorithms when relatively good accuracy 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. Once again, *there really is no such thing as a free lunch.*

When, then, should you consider using genetic algorithms on a real-life research problem? There is no clear-cut answer to this questions. You certainly can use a genetic algorithm to solve *anything* that can be formulated as a minimization/maximization task —and in principle anything described by equations 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, arguably, is in fact the First Rule of Scientific Computing, also known as

HAMMING'S FIRST RULE:

"The purpose of computing is insight, not numbers"

Correctness; consistency; justifiability; certainty; orderliness; parsimony; and decisiveness; these are the seven basic features of "good" conventional optimization methods identified by Koza (1992) in the introductory essay of his book on Genetic Programming. He then goes on to argue that genetic algorithms embody none of these presumably sound principles. Is this ground to shy away from 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 essay 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[7]

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

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

"Evolution is cleverer than you are"

Suggested assorted further reading

In case some of the above ramblings have intrigued you a bit, here is a small assortment of readings that I found useful in musing over these and related pseudo-philosophical matters.

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

APPENDIX: SOURCE CODE

A.1 New subroutines for PIKAIA 1.2

The following are reduced listing of the three subroutines in `PIKAIA 1.2` that present significant changes from `PIKAIA 1.0`. As with all `PIKAIA 1.0` code listings included in the PUG, the listings are "reduced" in that the corresponding subroutines in `PIKAIA 1.2` many more explanatory comments lines than the listing given here; all operational lines are included as they appear in the source code; and all `REAL`, `INTEGER`, etc, statements have been regrouped independently of their local/input/output status.

```
      subroutine cross(n,nd,pcross,gn1,gn2)
      implicit none
      integer        n, nd
      real           pcross
c=====================================================================
c     breeds two parent chromosomes into two offspring chromosomes
c     Either uniform one-point or two-point crossover is used,
c     chosen anew with equal probability at each breeding event
c=====================================================================
c     USES: urand
      integer        i, ispl, ispl2, itmp, t
      real           urand
      external       urand
c     Use crossover probability to decide whether a crossover occurs
      if (urand().lt.pcross) then
c        Compute first crossover point
         ispl=int(urand()*n*nd)+1
c        Choose between one-point and two-point crossover
         if (urand().lt.0.5) then
            ispl2=n*nd
         else
            ispl2=int(urand()*n*nd)+1
c un-comment following line to enforce one-point crossover
c           ispl2=n*nd
            if (ispl2.lt.ispl) then
               itmp=ispl2
               ispl2=ispl
               ispl=itmp
            endif
         endif
c        Swap genes from ispl to ispl2
         do 10 i=ispl,ispl2
            t=gn2(i)
            gn2(i)=gn1(i)
            gn1(i)=t
   10    continue
      endif
      return
      end
c*********************************************************************
      subroutine mutate(n,nd,pmut,gn,imut)
      implicit none
      integer        n, nd, imut, gn(n*nd)
      real           pmut
c=====================================================================
c     Generalized uniform one-point and creep mutation operator
```

```
c     Mutations occur at rate pmut at all gene loci
c        imut=1    Uniform mutation, constant rate
c        imut=2    Uniform mutation, variable rate based on fitness
c        imut=3    Uniform mutation, variable rate based on distance
c        imut=4    Uniform or creep mutation, constant rate
c        imut=5    Uniform or creep mutation, variable rate based on
c                  fitness
c        imut=6    Uniform or creep mutation, variable rate based on
c                  distance
c======================================================================
c     USES: urand
      integer        i,j,k,l,ist,inc,loc,kk
      real           urand
      external       urand
c     Decide which type of mutation is to occur
      if(imut.ge.4.and.urand().le.0.5)then
c     CREEP MUTATION OPERATOR
c     Subject each locus to random +/- 1 increment at the rate pmut
         do 1 i=1,n
            do 2 j=1,nd
               if (urand().lt.pmut) then
c     Infer location on string, substring boundary, and pick +1 or -1
                  loc=(i-1)*nd+j
                  inc=nint ( urand() )*2-1
                  ist=(i-1)*nd+1
                  gn(loc)=gn(loc)+inc
c                 Now we carry over the one, if needed
c                 Decrement case
                  if(inc.lt.0 .and. gn(loc).lt.0)then
                     if(j.eq.1)then
                        gn(loc)=0
                     else
                        do 3 k=loc,ist+1,-1
                           gn(k)=9
                           gn(k-1)=gn(k-1)-1
                        if( gn(k-1).ge.0 )goto 4
    3                   continue
                        if( gn(ist).lt.0.)then
                           do 5 l=ist,loc
                              gn(l)=0
    5                      continue
                        endif
    4                   continue
                     endif
                  endif
c                 Increment case
```

```
                if(inc.gt.0 .and. gn(loc).gt.9)then
                   if(j.eq.1)then
                      gn(loc)=9
                   else
                      do 6 k=loc,ist+1,-1
                         gn(k)=0
                         gn(k-1)=gn(k-1)+1
                         if( gn(k-1).le.9 )goto 7
    6                 continue
                      if( gn(ist).gt.9 )then
                         do 8 l=ist,loc
                            gn(l)=9
    8                    continue
                      endif
    7                 continue
                   endif
                endif
             endif
    2     continue
    1   continue
      else
c     UNIFORM MUTATION OPERATOR
c     Subject each locus to random mutation at the rate pmut
         do 10 i=1,n*nd
            if (urand().lt.pmut) then
               gn(i)=int(urand()*10.)
            endif
   10    continue
      endif
      return
      end
c**********************************************************************
      subroutine adjmut(ndim,n,np,oldph,fitns,ifit,pmutmn,pmutmx,
     +                  pmut,imut)
      implicit none
      integer        n, ndim, np, ifit(np), imut
      real           oldph(ndim,np), fitns(np), pmutmn, pmutmx, pmut
c======================================================================
c     dynamical adjustment of mutation rate;
c     imut=2 or imut=5 : adjustment based on fitness differential
c                        between best and median individuals
c     imut=3 or imut=6 : adjustment based on normalized metric distance
c                        between best and median individuals
c======================================================================
      integer        i
      real           rdif, rdiflo, rdifhi, delta
```

```
      parameter      (rdiflo=0.05, rdifhi=0.25, delta=1.5)
      if(imut.eq.2.or.imut.eq.5)then
c     Adjustment based on fitness differential
         rdif=abs(fitns(ifit(np))-fitns(ifit(np/2)))/
     +           (fitns(ifit(np))+fitns(ifit(np/2)))
      else if(imut.eq.3.or.imut.eq.6)then
c     Adjustment based on normalized metric distance
         rdif=0.
         do 1 i=1,n
            rdif=rdif+( oldph(i,ifit(np))-oldph(i,ifit(np/2)) )**2
    1    continue
         rdif=sqrt( rdif ) / float(n)
      endif
      if(rdif.le.rdiflo)then
         pmut=min(pmutmx,pmut*delta)
      else if(rdif.ge.rdifhi)then
         pmut=max(pmutmn,pmut/delta)
      endif
      return
      end
```

A.2 Other minor coding changes in `PIKAIA 1.2`

This section described other minor coding changes made throughout `PIKAIA 1.2` to accommodate the new operators and strategies described in these release notes. The only changes occur in the main program `pikaia`, and in subroutine `setctl`.

A.2.1 main program

Because subroutine `mutate` must know if creep mutation is to be used, and subroutine `adjmut` whether fitness-based or distance-based adjustment is invoked, the control variable `imut` is now passed through both subroutine's argument lists. Likewise, to compute metric distance subroutine `adjmut` must be passed the population array `oldph`. The corresponding calls in subroutine `pikaia` now look like:

```
      call mutate(n,nd,pmut,gn1,imut)
      call mutate(n,nd,pmut,gn2,imut)
      if (imut.eq.2.or.imut.eq.3.or.imut.eq.5.or.imut.eq.6)
     +   call adjmut(NMAX,n,np,oldph,fitns,ifit,pmutmn,pmutmx,
     +               pmut,imut)
```

A.2.2 subroutine setctl

The safety checks on the value of `imut` carried out in subroutine `setctl` now reflects the new valid numerical values for this control variable:

```
      if(imut.ne.1.and.imut.ne.2.and.imut.ne.3.and.imut.ne.4
     +    .and.imut.ne.5.and.imut.ne.6) then
         write(*,10)
         status = 5
      endif
   10 format(' ERROR: illegal value for imut (ctrl(5))')
```

And that is all!

BIBLIOGRAPHY

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

Charbonneau, P. 1995, *The Astrophysical Journal Supplement*, **101**, 309-XXX

Charbonneau, P. 1998, An introduction to genetic algorithms for numerical optimization (`www.hao.ucar.edu/public/research/si/pikaia/pikaia.html`, → "Oslo Tutorial")

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)

Craig, I.J.D., & Brown, J.C. 1986, *Inverse Problems in Astronomy* (Bristol, UK: Adam Hilger).

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

Fontaine, G., Brassard, P., & Bergeron, P. 2001, *Publication of the Astronomical Society of the Pacific*, **113**, 409-XXX

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

Holland, J.H. 1992, *Adaptation in Natural and Artificial Systems*, Second Edition (Cambridge: MIT Press; first ed. 1975)

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)

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

McIntosh, S.W., Charbonneau, P., & Brown, J.C. 2000, *The Astrophysical Journal*, **529**, 1115-1130

Metcalfe, T., & Nather, R. 1999, *Linux Journal*, **65**, 58-XX

Metcalfe, T., Nather, R., & Winget, D. 2000, *The Astrophysical Journal*, **545**, 974-XXX

Metcalfe, T., Winget, D., & Charbonneau, P. 2001, *The Astrophysical Journal*, **557**, 1021-XXX

Michalewicz, Z. 1994, *Genetic Algorithms + Data Structures = Evolution Programs* (New York: Springer)

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