NCAR/TN-418+IA
NCAR TECHNICAL NOTE

December 1995

A USER'S GUIDE TO PIKAIA 1.0

Paul Charbonneau
Barry Knapp

HIGH ALTITUDE OBSERVATORY

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
BOULDER, COLORADO

TABLE OF CONTENTS

List of Figures v

List of Tables vii

Preface ix

1. Introduction: genetic algorithms and evolutionary biology
   - 1.1 Evolution as an optimization process? 1
   - 1.2 A basic genetic algorithm 2
2. Installation and quick start
   - 2.1 Obtaining a copy of `PIKAIA` 5
   - 2.2 Installation 5
   - 2.3 User support 7
   - 2.4 Bug reports 7
   - 2.5 Differences with pre-release versions 8
   - 2.6 Reference and credits 8
3. The `PIKAIA` subroutine
   - 3.1 Overview and problem formulation 9
   - 3.2 Top-level structure 10
   - 3.3 Initial population 11
   - 3.4 Selection technique 12
   - 3.5 Encoding and decoding 15
   - 3.6 The crossover operator 17
   - 3.7 The mutation operator 18
   - 3.8 Reproduction plans 20
   - 3.9 Elitism 22
4. Using `PIKAIA`
   - 4.1 Calling sequence 25
   - 4.2 Input/Output 25
   - 4.3 Fitness function 25
   - 4.4 Internal safety checks and Error/Warning messages 26
   - 4.5 The input control vector `ctrl` 26

4.6 Additional user-supplied functions and subroutines 31
4.7 Efficiency considerations 32

5. Examples

5.1 Maximizing a function of two variables 35
5.2 A linear least-squares fit 41
5.3 A non-linear least-squares fit 47
5.4 Generalized least squares fitting by distance regression 56
5.5 Data modeling using robust estimators 60
5.6 A warning concerning error estimates 65
5.7 Other applications 66

6. Where to go from here: hacking `PIKAIA`

6.1 Overall coding structure and subroutine dependencies 69
6.2 Tailoring or expanding the Output 69
6.3 Incorporating additional strategies and operators 71
6.4 Suggested further readings 74

Appendix: Code listings

A.1 Source code for `PIKAIA` 77
A.2 Random number generator and ranking subroutine 90
A.3 Sample driver and fitness function for installation check 96
A.4 Drivers for the example problems of chapter 5 98

Bibliography 107

Postface 109

LIST OF FIGURES

3.1 Adjusting Selection Pressure 14

4.1 Convergence curve for the test problem of §2.3 33

5.1 2-D surface for the installation check problem of §2.3 36
5.2 Performance of PIKAIA on the maximization problem of Fig. 5.1 39
5.3 Evolution of the population during the PK3 evolutionary run 40
5.4 Synthetic dataset 42
5.5 Genetic linear least squares fit to the dataset of Fig. 5.4 45
5.6 Evolution of solution parameters throughout the evolution 46
5.7 Genetic nonlinear least squares fit to the dataset of Fig. 5.4 52
5.8 Evolution of the best χ^2 for the solutions of Fig. 5.7 53
5.9 Synthetic dataset for the generalized nonlinear least squares problem 57
5.10 A 200-generation genetic fit to the synthetic dataset of Fig. 5.9 60
5.11 Synthetic dataset for a "damaged" circle 62
5.12 Hough transform for the dataset of Fig. 5.11 63
5.13 Robust fit to the dataset of Fig. 5.11 64

6.1 Dependency chart for PIKAIA 70

LIST OF TABLES

I Elements of input control vector 30

II Run parameters and global performance for first example problem 38

III Nonlinear least squares solutions: parameters for best individuals 50

IV Internal arrays 71

PREFACE

"Evolution is cleverer than you are" writes Francis Crick, co-discoverer of DNA and 1962 Nobel laureate. His point may well be lost on the majority of practicing scientists trained in the physical sciences, where determinism, reproducibility and predictability are central tenets of the *Weltanschauung* that is often taken to define "hard science". Yet time and time again, evolution has accomplished astonishing feats of engineering that, to this day, remain the envy of researchers at the forefront of robotics and artificial intelligence. It is also particularly noteworthy that evolution has done so without any external, "higher" guidance, through processes that are simple, local, and incorporate markedly stochastic components.

Perhaps the most striking operational feature of living organisms is their high level of *adaptation* to their environment. The ubiquitous existence of complex adaptations in living organisms was an important factor in leading Charles Darwin to formulate his original theory of evolution by means of natural selection. Adaptation was also the focal point of John Holland's research in the late 1960's and early 1970's, which was extremely influential in originating and guiding the development of what are now usually referred to as *genetic algorithms*. Twenty years after the publication in 1975 of Holland's seminal *Adaptation in Natural and Artificial Systems*, genetic algorithms have amply demonstrated their usefulness (and robustness) in a variety of problem settings. Yet more often than not they continue to elicit mixed reactions on the part of established practitioners overly committed to conventional, deterministic optimization methods.

The genetic algorithm-based optimization subroutine `PIKAIA` described in this guide is tailored towards problems of *numerical* optimization. It fills a rather specific ecological niche; unlike the most popular genetic algorithm packages currently available commercially or in the public domain, `PIKAIA` is written in plain old standard FORTRAN 77, as opposed to the "better" languages usually favored by hard-nosed computer scientists. Furthermore, we have attempted to emulate the user-friendly style of the subroutines to be found in Press *et al.* (1992)'s *Numerical Recipes*. This, admittedly, is a high standard to aim for. Only time will tell by how far we missed. While we consider `PIKAIA` to be more of a learning instrument than a true production code, we do believe that we have succeeded in producing a piece of optimization software that is astonishingly robust by any standards.

This user's guide is organized as follows; chapter 1 is a very brief review of some aspects of the biological evolutionary process relevant to the understanding of genetic algorithms. Chapter 2 is concerned with pragmatic issues including how to obtain, install and validate the code. Chapters 3 and 4 contain most of the material traditionally found in a user's guide; chapter 3 describes the various genetic operators and ecological strategies incorporated in `PIKAIA` including, at time, fairly detailed discussions of specific implementation issues. Chapter 4 contains detailed descriptions of input parameters, calling sequence, additional routines required from the user, etc. Readers favoring the "hands on" approach to learning may get a quick start by going straight to chapter 4 upon successfully completing the installation, and only subsequently study the (important) material discussed in chapter 3. As a user's guide, the present text differs perhaps most prominently from the norm in including a lengthy chapter (32 pages), chapter 5, presenting and discussing examples of applications of the subroutine to a sequence of increasingly difficult data modeling problems, including code listings for example fitness functions. Chapter 6 provides additional information hopefully useful to users wanting to modify, expand and/or tailor the code itself, and includes an annotated bibliography pointing to what we think are good entry points in the genetic algorithm literature. Even though this user's guide is neither meant to be a tutorial nor a textbook, we did attempt to discuss relevant background material and include practical tips and general guidelines useful in dealing with real life problems, wherever most appropriate in the text and in particular in chapter 5.

While it was our original intention to avoid jargon, we ended up retaining some of the biologically inspired terminology often used in the genetic algorithms literature. In the course of writing this user's guide it became clear to us that (1) the biological terminology is actually quite useful in describing the mode of operation of genetic algorithms, and it is certainly no more obscure than the alternate computer science-based jargon; (2) a prospective user not at all exposed to this terminology and trying to look more deeply into aspects of genetic algorithms not covered in this guide may have some difficulty working through some of the relevant literature. All the required biological terminology is laid out in chapter 1, and is used minimally and in as unambiguous a way as possible.

A preface is traditionally the place where authors acknowledge outside contributions, and a number of individuals certainly deserve mention here. Numerous discussions with Frank Crary, in the early developmental stages of the `PIKAIA` code, have contributed significantly in shaping the final product. Ted Kennelly was our first "friendly user" to successfully use an early version of the code to solve a real research problem, a process in the course of which much was learned about what makes (or breaks) a truly user-friendly genetic algorithms-based optimization subroutine. Useful feedback was also provided by attendees of the two-day intensive class taught by one of us (P.C.) at the Centre de Recherche en Calcul Appliqué (Montréal, Canada) in May 1995, in particular by Richard Boivin, Claude Carignan, Robert Lamontagne and Jacques Richer. We also wish to thank Tim Brown for suggesting the "damaged circle" test problem for robust estimators (§5.5), for pointers to the Hough transform as used in computer vision, and for numerous fruitful discussions of related issues in data modeling. Thanks are also due to Gene Lavely for pointers to applications of genetic algorithms in the geophysical literature, and to Ken De Jong for taking the time to clarify for us some important points related to error estimation in genetic solutions, including the warning which we repeat essentially *verbatim* at the opening of §5.6. Last but not least, we are grateful to Peter Fox for a critical reading of the final draft of this guide.

Paul Charbonneau
Barry Knapp

December 1995, Boulder

1. INTRODUCTION: GENETIC ALGORITHMS AND EVOLUTIONARY BIOLOGY

1.1 Evolution as an optimization process?

The general ideas of *evolution* and *adaptation* predate Charles Darwin's 1859 *On the Origin of Species by means of natural Selection* (see Bowler, 1983), but it is Darwin (and more or less simultaneously A.R. Wallace) who first identified what is still considered by most to be the primary driving mechanism of evolution: *natural selection*. Natural selection is the process whereby individuals better adapted to their environment (i.e., "fitter", in the wider sense of the word) tend to produce, on average, more offspring that their less well endowed competitors in the breeding population. Darwin and his contemporaries also realized that two additional ingredients are required for natural selection to lead to evolution. The first is *heredity*; an offspring must inherit, in some way, some of the characteristics that make its parents fit, otherwise evolution is effectively reset to zero with each new generation. The second ingredient is *variability*: at any given time there must exist a spectrum of fitness among population members, otherwise natural selection simply cannot operate.

Although both these aspects remained unexplained in Darwin's lifetime, the primary processes through which heredity is mediated and variation maintained are now basically understood. Each cell of each individual (or *phenotype*) contains a complete set of instructions effectively defining its physical (and possibly behavioral) makeup. This information is encoded in the form of linear gene sequences stored on pairs of homologous *chromosomes*, which constitute the individual's *genotype*. Sexual reproduction involves the combination of genetic material from both parents, one half of each chromosome pair coming from each parent. A fundamental aspect of this breeding process is that the relationship between phenotype and genotype is unidirectional; a given individual can be thought of as an external manifestation of its genotype (although there exist environmental influences in development and growth that are beyond genetic control), but the individual cannot influence its own genetic makeup. It can, however, influence the genetic makeup of subsequent generations through differential reproductive success, which is of course where natural selection plays its crucial role. To a large extent, variability turns out to be maintained by the machinery of heredity itself. The production of reproductive cells often entails the recombination of genetic material across homologous chromosomes through the processes of *crossover* and *inversion*. Copying mistakes and/or true random events also occasionally introduce *mutations* in the genotype. The ensemble of all genes existing at a given time in the breeding population makes up the *gene pool*. For a given gene associated with a chromosomal locus, there exist in general more than one allowed "gene value" (or *allele* in biological terminology). Evolution can be thought of —and mathematically modeled, see e.g. Maynard Smith (1989)— in terms of temporal changes in allele frequencies throughout the gene pool.

Genetic Algorithms (hereafter "GA") are a class of heuristic search techniques that incorporate these ideas in a setting that is computational rather than biological. Strictly speaking, genetic algorithms do not optimize, and neither does biological evolution (Holland 1992, preface; De Jong 1993); evolution uses whatever material that is at its disposal to produce above-average individuals. Evolution is blind. Evolution has no ultimate goal of "perfection". 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". One could perhaps argue than evolution performs a form of highly constrained optimization, but even then it certainly does not optimize in the mathematical sense of the word. Nevertheless, genetic algorithms form the basis of a class of extremely robust optimization method known as *GA-based optimizers*. The GA-based subroutine `PIKAIA` is one such optimizer.

1.2 A basic genetic algorithm

In their simplest incarnation, genetic algorithms make use of the following reduced version of the biological evolutionary process; the gene pool —and its associated phenotypic population— evolves in response to

(1) differential reproductive success in the population,
(2) genetic recombination (crossover) occurring at breeding,
(3) random mutations affecting a subset of breeding events.

Consider then the following generic optimization problem. One is given a "model" that depends on a set of parameters $\mathbf{a}$, and a functional relation $f(\mathbf{a})$ that returns a measure of quality for 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 consists in finding the "point" $\mathbf{a}^*$ defining a model that maximizes the quality measure $f(\mathbf{a})$. For the sake of the argument suppose that one has available a target value F and define a tolerance criterion ε (> 0) such that a "solution" $\mathbf{a}^*$ satisfying

$$|F - f(\mathbf{a}^*)| \leq \varepsilon \tag{1.1}$$

corresponds to a model deemed acceptable. Define now a population A as a set of K realizations of the parameters $\mathbf{a}$:

$$A \equiv \{\mathbf{a}_k\}, \qquad k = 1, 2, ..., K \tag{1.2}$$

and an operator $\mathcal{R}$ that, when applied to a given population A^{n-1}, produces a new population A^n. From these building blocks a basic genetic algorithm could be constructed as follows:

Initialize: $A^0 \equiv \mathbf{a}_k^0, \qquad k = 1, ..., K$
Compute: $f_k^0 \equiv f(\mathbf{a}_k^0)$
$n := 0$
do while $|F - \max(f_k^n)| \geq \varepsilon$
  $n := n + 1$
  $A^n = \mathcal{R}(A^{n-1})$
  Compute: $f_k^n, \qquad k = 1, ..., K$
end do
$\mathbf{a}^* = \mathbf{a}_{k(max)}^n$

Were it not that the operator $\mathcal{R}$ acts on a population rather than on a single individual, this would look very much like some generic Monte Carlo algorithm. The crucial difference lies with the definition of the operator $\mathcal{R}$. First, $\mathcal{R}$ does not operate directly on the parameters $\mathbf{a}_k$ (the phenotypes), but rather on their encoded versions (the genotypes). Second, $\mathcal{R}$ applies crossover and mutation operations on the genotypes, processes that involve markedly stochastic aspects, as opposed to fully deterministic schemes such as averaging. Third, $\mathcal{R}$ does not operate on the existing population in a homogeneous fashion, but selects a subset of the population on the basis of their quality measure $f(\mathbf{a})$ (their fitness). In analogy with biological systems, a phenotype is encoded in the form of a string (or chromosome) of digits. In contrast to biological systems, in basic genetic algorithms it is common to fully encode a phenotype on a single chromosome, as opposed to groups of homologous chromosome pairs with dominant/recessive character. A genotype then is made of a single chromosome, and both terms can be used interchangeably (to the probable dismay of evolutionary biologists...).

These considerations notwithstanding, the algorithm listed above still only defines an *adaptive plan* or *evolution strategy*. It pretty much ensures gradual improvement over successive iterations, but does not guarantee absolute maximization in anything approaching the strict mathematical sense of the word. GA-based optimizers typically implement additional strategies and techniques to improve performance in the context of numerical optimization. A number of these, incorporated in the GA-based optimization subroutine `PIKAIA`, are described in detail in chapter 3 below.

Genetic algorithms have been used successfully to solve a number of difficult optimization problems arising in computer science, artificial intelligence, computer-aided engineering design, geoseismic modeling, and are attracting increasing attention in in other branches of the physical sciences. Yet, generally speaking, they are not yet a standard component of the numerical modeler's toolboxes. In the delightful introductory essay at the beginning of his monograph on genetic programming, Koza (1992) identifies seven basic principles of good conventional optimization techniques: correctness, consistency, justifiability, certainty, orderliness, parsimony, and decisiveness. He then goes on to argue that genetic algorithms incorporate *none* of these presumably sound principles. This may go a long a way in explaining the occasional resistance encountered by genetic algorithm practitioners attempting to convince the skeptics of the power, robustness, and ultimately usefulness, of GA-based optimizers.

Watching a genetic algorithm in progress, and seeing the ease with which it locates and locks on to the optimal solution, can be a deeply fascinating, if not troubling experience. This is perhaps because conceptually, genetic algorithms embody the very mechanisms that led to our own existence. Such mystical considerations notwithstanding, the bottom line, if there is to be one, is that genetic algorithms *work*, and often frightfully well. From a purely pragmatic standpoint, this is indeed the bottom line. It is also precisely the messaged conveyed by the theory of (biological) evolution by means of natural selection.

2. INSTALLATION AND QUICKSTART

2.1 Obtaining a copy of PIKAIA

Source codes for `PIKAIA`, including the subroutine itself as well as driver programs and fitness functions for the examples discussed in chapter 5 of this guide, can be obtained from NCAR's High Altitude Observatory via anonymous ftp. First ftp to

```
hao.ucar.edu
```

and log in using the username `anonymous` and your e-mail address as the password. You have now landed on a UNIX system. Type the following command:

```
cd /pub/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 we could not (or forgot to) include in this guide. The file `userguide.ps` is a standard postscript file containing this user's guide, including encapsulated Figures, that should be printable on any postscript printer. Note that this is a large file, about 2 Megabytes. The file `userguide.txt` is an ASCII character version of the users guide, excluding Figures but including Figure captions. This should be printable on *any* printer that recognizes the standard ASCII character set. The file `pikaia.f` contains not only the genetic subroutine `PIKAIA` 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 §2.2 below). The directory `examples` contains drivers, fitness functions and synthetic datasets for the examples discussed in chapter 5 of this guide.

2.2 A test problem for installation check

The file `pikaia.f` includes a sample fitness function and driver code which can (and should) be used as a test problem for installation check. The precise nature of this test problem is presently unimportant; let us simply mention that the problem consists in finding the global maximum (x^*, y^*) of a function $f(x, y)$ that describes a 2-D "landscape" characterized by multiple regions of secondary maxima, with the absolute maximum located at $(x^*, y^*) = (0.5, 0.5)$, where $f(x^*, y^*) = 1.0$. This maximization problem will be revisited in due time below. For now simply compile/link the file `pikaia.f` (on a Unix system the command `f77 pikaia.f` should produce the executable `a.out`). Now run the executable; you are first prompted for a (positive and non-zero) integer value used as a seed for `PIKAIA`'s random number generator:

```
Random number seed (I*4)?
```

Type in the value 123456 and hit <return>; after a short delay (the duration of which being dependent on the hardware platform used) something similar to the following output should be returned to the screen:

```
status:           0
         x:   0.5002700       0.4998800
         f:   0.9999299
      ctrl: 100.00000 50.00000  5.00000  0.85000  2.00000  0.00500
              0.00050  0.25000  1.00000  1.00000  1.00000  0.00000
```

The value `status=0` indicates normal termination. The next two lines indicate that `PIKAIA` has found a "solution" $(x^*, y^*) = (0.50027, 0.49988)$ for which $f(x^*, y^*) = 0.9999299$. This actually compares favorably to the true global maximum $f(0.50000, 0.50000) = 1.00000$, even though the evolution is only carried out over 100 generations for this test problem. The next two lines simply echo some input parameter values, corresponding here to internal default values except for the generational length (more on all of these in §4.5 below).

It is important to realize that the behavior of the random number generator is dependent on the specific implementation of floating-point arithmetics, and so can be expected to be platform-dependent. Running the test problem on various platform will *not*, in general, produce a solution identical to that listed here. This is because the evolutionary "path" followed by the population is dependent on stochastic components which are governed by operators relying on the sequence of deviates returned by the random number generator (more on this in §4.6.1 below). For the same reason, different values of the `seed` for the random number generator will in general lead to different solutions. One may find that solutions having $f(x^*, y^*) \simeq 0.95$ are sometimes returned. The occasional occurrence of such a solution, for certain `seed` values, should not be a cause for alarm. However, one should become suspicious if upon using different random seeds `PIKAIA` systematically fails to converge to a solution having $(x^*, y^*) \simeq (0.50, 0.50)$. Such a behavior may indicate a true problem.

We suggest that upon successful installation, prospective users send a short e-mail to the following address:

pikaia@hao.ucar.edu

including their valid and complete e-mail address; the point is not so much to keep track of who obtained a copy of the code, but rather to keep an updated mailing list of past and present users. This would allow us to notify the user community of the timing of future releases, or distribute code patches or bug correction information, as the case may be.

2.3 User support (or more specifically, lack thereof)

Although we do *not* formally commit to providing any kind of user support, we remain interested in hearing any comment, suggestion, and/or criticism concerning the code or this user's guide. Inasmuch as our respective work commitments allow it, we *may* answer application-related questions, or at least try to provide pointers to appropriate entry points in the GA literature. Out of curiosity, we would certainly be interested in being kept informed, if only briefly, of any successful application of our little subroutine to real-life research problems. All queries or other communications should be e-mailed to:

pikaia@hao.ucar.edu

Response time should be expected to be extremely variable.

2.4 Bug reports

Although we have, at this writing, tested `PIKAIA` extensively on a number of test problems, and have used it successfully for an already lengthy list of real applications, the possibility always remains that we managed to miss something. If a user finds a bug somewhere, we would REALLY like to hear about it, so that we can notify the user community accordingly. Reports for bugs (or suspected bugs) should be e-mailed to, you guessed it:

pikaia@hao.ucar.edu

Please make it clear in your message header that the e-mail is related to a potential bug report, as we will tend to give priority to such messages, over other types of queries.

2.5 Differences with pre-release versions

A small number of pre-release versions of `PIKAIA` have been in circulation since May 1995. These versions can be identified by the comment line

```
c Version of 1995 April 13
```

in the long series of comment lines at the beginning of the subroutine `pikaia.f` itself. These various versions differ only in minor ways from version 1.0 described here. Differences are to be found in the settings of some default values, maximum array sizes, and so on. Note however that these early versions do not test for even/odd population size (more on this below), or for zero value of the random seed.

Prior to May 1995, various ancestral versions of `PIKAIA`, going under the names of `darwin`, `darwin1`, or `darwin2`, were also distributed informally to a few individuals. These codes bear only superficial resemblance to `PIKAIA 1.0`, and are markedly inferior in a number of respects (modularity, efficiency, etc.). Anybody in possession of such versions should definitely obtain a copy of `PIKAIA 1.0`, especially since some of these ancestral versions contain a rather involved bug that significantly degrades performance under some parameter settings.

2.6 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.

Note that the code listing included as an Appendix to that paper corresponds to the pre-release version of 1995 April 13 mentioned previously. The present user's guide should be referenced as

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

`PIKAIA` is a public domain piece of software, and so we impose no restrictions on further distribution for research purposes. We do encourage "second-hand" users to send a brief e-mail to the above-cited e-mail address, for the purposes already mentioned.

3. PIKAIA: A GENETIC ALGORITHM-BASED OPTIMIZATION SUBROUTINE

3.1 Overview and problem definition

`PIKAIA` is a general purpose optimization subroutine based on a genetic algorithm. The subroutine is written in FORTRAN, and except for two minor exceptions adheres strictly to the ANSI FORTRAN-77 standard. These exceptions are (1) the use of lower case alphabet, and (2) the systematic use of `implicit none` statements. Readers having strong allergic reactions to FORTRAN and/or interested in more elaborate genetic algorithm packages may wish to take a look at the Genetic Algorithm Archive Web Page (http://www.aic.nrl.navy.mil/galist) for listings and directions to various other public domain GA packages available electronically.

Internally, `PIKAIA` seeks to maximize a (user-defined) function $f(\mathbf{x})$ in a bounded `n`-dimensional space, i.e.,

$$\mathbf{x} \equiv (x_1, x_2, ...x_\mathrm{n}), \qquad x_k \in [0.0, 1.0] \quad \forall k \tag{3.1}$$

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

`PIKAIA` carries out its maximization task on a population made up of `np` individuals (trial solutions). This population size remains fixed throughout the evolutionary process. Rather than evolving the population until some preset tolerance criterion is satisfied, `PIKAIA` carries the evolution over a fixed, preset number of generations.

`PIKAIA` is meant primarily to be a learning instrument, not a production code. In most instances where conflict arose between efficiency and clarity, the former was sacrificed for the sake of the latter.

3.2 Top-level Structure

The core of PIKAIA is made up of two nested loops controlling the generational and reproductive cycles. These loops are structured as follow:

```
   do 10 ig=1,ngen                [Generational cycle          ]
      do 20 ip=1,np/2             [Reproductive cycle          ]
         call select(ip1)         [Pick dad                    ]
21       call select(ip2)         [Pick mom                    ]
         if(ip1.eq.ip2) goto 21   [No breeding with oneself!   ]
         call encode(ph1,gn1)     [build dad's chromosome      ]
         call encode(ph2,gn2)     [build mom's chromosome      ]
         call cross (gn1,gn2)     [The actual X-rated part     ]
         call mutate(gn1)         [Mutate first offspring      ]
         call mutate(gn2)         [Mutate second offspring     ]
         call decode(ph1,gn1)     [Decode first offspring      ]
         call decode(ph2,gn2)     [Decode second offspring     ]
         if(irep.eq.1)then        [Insert/store both offspring]
            call genrep(ip,ph1,ph2,newph)
         else
            call stdrep(irep,ph1,ph2,oldph,fitns,ifit,jfit)
         endif                    [insertion/storage completed]
20    continue
      if(irep.eq.1) call newpop   [Off with their heads!       ]
10 continue
```

(For the purpose of this listing, the argument lists of most subroutines have been truncated). The population size (`np`) and the number of generations through which this population is to evolve (`ngen`) are both input parameters. Note that each iteration of the inner loop (`do 20`) produces *two* offspring from two parents, so that this loop needs to be executed only `np/2` times in order to produce a new generation of `np` individuals. This implies that the population size `np` must be an even number; if the user specifies an odd value for `np`, PIKAIA truncates it to `np`−1, issues a warning message, and proceeds with the run. One iteration of the reproductive cycle entails choosing two parents (`call select`), and constructing their respective chromosomes (`gn1`, `gn2`) from their corresponding phenotypes (`ph1`, `ph2`). The crossover operator is then applied (`call cross`) to the parent chromosomes, and the mutation operator applied (`call mutate`) to each of the two resulting offspring chromosomes. The offspring chromosomes are then decoded into their corresponding phenotypes (`call decode`), and accumulated into temporary storage (`call genrep`) or inserted into the population (`call stdrep`), depending on the adopted reproduction plan (controlled by the parameter `irep`; more on this below). Under some reproduction plans, the new population is transferred from temporary storage to the main population array at the end of the generational iteration (`call newpop`), with the old population being eliminated.

`PIKAIA` stores the population in the 2-D array `oldph`, of size `n`$\times$`np`, where `n` is the number of parameters defining the function to be maximized. The column `oldph(1:n,i)` corresponds to the `i`$^{\rm th}$ individual in the population. Many genetic algorithm packages store instead the population of strings (chromosomes); the present choice was motivated by the desire to simplify I/O, and facilitate any postprocessing operation that the user may want to carry out.

3.3 Initial population

In order to avoid any bias at the beginning of the evolutionary run, each of the `n` parameters defining each of the `np` individuals in the initial population is initialized with a random number $R \in [0.0, 1.0]$. `PIKAIA` carries out this initialization in the following manner:

```
      do 1 ip=1,np                      [np individuals in population]
         do 2 k=1,n                     [n parameters per individual ]
            oldph(k,ip)=urand()         [Random initialization       ]
    2    continue
         fitns(ip) = ff(n,oldph(1,ip)) [compute fitness             ]
    1 continue
      call rnkpop(np,fitns,ifit,jfit)   [Rank population             ]
```

where the function `urand()` returns a uniformly distributed random real number $R \in [0.0, 1.0]$. The fitness is then calculated for each individual phenotype (a column `1:n` in the population matrix `oldph`), and stored in the 1-D array `fitns`. In view of this random initialization process, most individuals will typically end up with very low fitness, but the fact remains that some will be better (or perhaps more accurately, not as bad) as others. This is all that is required for natural selection to operate, since it simply favors above-average individuals, even if the average is actually really pathetic on an absolute scale. Evolution makes use of whatever material it has at its disposal. Remember, evolution is blind; evolution simply produces better-than-average individuals.

The call to subroutine `rnkpop` returns the two 1-D integer ranking arrays `ifit` and `jfit`. These are used internally by `PIKAIA` to keep track of where individuals are stored the population matrix.

3.4 Selection technique

PIKAIA uses a stochastic (as opposed to deterministic) sampling mechanism to select *both* parents in one iteration of the reproductive cycle. The selection procedure is such that the *probability* of an individual being selected for breeding is proportional to that individual's fitness.

3.4.1 The roulette wheel algorithm

Let S_i be the fitness of individual i. First compute the sum of all fitness values in the population,

$$F = \sum_{i=1}^{\mathrm{np}} S_i, \tag{3.2}$$

and define a running sum

$$T_j = \sum_{i=1}^{\mathrm{j}} S_i, \qquad j = 1, ..., \mathtt{np}; \tag{3.3}$$

clearly $T_{j+1} \geq T_j \, \forall j$, and $T_{\mathrm{np}} = F$. Now generate a random number $R \in [0.0, F]$, and locate the element T_j for which

$$T_{j-1} \leq R < T_j \tag{3.4}$$

In view of eq. (3.3), for a given value of R this condition can be satisfied for one and only one j; the corresponding individual is the one selected for breeding. Note that this procedure *requires* the fitness S (as returned by the user-supplied fitness function) to be a positive definite quantity.

While it may not be obvious at first glance, this procedure is equivalent to constructing a roulette wheel where each individual is assigned a sector of angular dimension $(2\pi)^{-1} \times S_i/F$, and the production of the random number is equivalent to spinning the wheel. The probability of the wheel stopping in sector j is clearly proportional to the angular extent of that sector, and thus to the (normalized) fitness of individual j. This procedure is accordingly known as the *Roulette Wheel Algorithm*. It is the only sampling mechanism available in PIKAIA. For a discussion of other possible sampling mechanisms see chap. 4 of Goldberg (1989) or §4.1 of Michalewicz (1994), and references therein.

3.4.2 Ranking as fitness

Directly using fitness as a measure of breeding probability suffers from a number of shortcomings (as discussed for example in Goldberg 1989, chap. 3; Davis 1991, chap. 3), but these can be overcome in a number of ways. In a nutshell, the aim is to ensure that a suitable fitness differential be maintained across the population throughout the evolutionary run. In the language of biology, *selection pressure* must be continually enforced for evolution to proceed. To achieve this effect, `PIKAIA` makes use of a strategy known as *ranking*.

Assign to each individual a rank r_j based on its fitness S_j. By convention let $r = 1$ correspond to the fittest individual, and $r = \mathtt{np}$ to the least fit. Then define a relative fitness S' in terms of this rank:

$$S'_j = \frac{\mathtt{np} - r_j + 1}{\mathtt{np}}, \tag{3.5}$$

(note that $S'_j \in [0,1]$ by construction). This relative fitness is then used as a measure of selection probability in the roulette wheel algorithm described above, in place of the true fitness S_j. In this way, a constant fitness differential is imposed across the population; this is most easily seen upon noting that the ratio of relative fitnesses for the best to second best individual in the population is always equal to [best:second] $=$ [$\mathtt{np}$:($\mathtt{np} - \mathtt{1}$)], the [best:median] ratio to [2:1], the [best:worst] ratio to [$\mathtt{np}$:1], and so on, *independently of the actual distribution of fitness values.*

3.4.3 Selection pressure

The ranking scheme described in §3.4.2 amounts to imposing a linear relationship between relative fitness and rank, with unit slope. This choice is clearly not unique. Consider instead the more general linear relationship:

$$S'(r) = b - m\left(\frac{r}{\mathtt{np}}\right), \qquad m, b \geq 0; \tag{3.6}$$

the ratio [best:worst] of normalized fitness becomes

$$\frac{S'_M}{S'_P} = \frac{b - m/\mathtt{np}}{b - m}. \tag{3.7}$$

Clearly the slope m is a measure of selection pressure. `PIKAIA` makes use of a modified representation of eq. (3.6) to define the relative fitness ($\equiv$ selection probability) as:

$$S'(r) = (\mathtt{fdif} + 1)(\mathtt{np} + 1) - 2r \times \mathtt{fdif}, \tag{3.8}$$

where

$$0.0 \leq \mathtt{fdif} \leq 1.0 \tag{3.9}$$

is equivalent to the slope m in the above linear relationship. Equations (3.6) and (3.8) are actually equivalent up to a normalization factor chosen such that

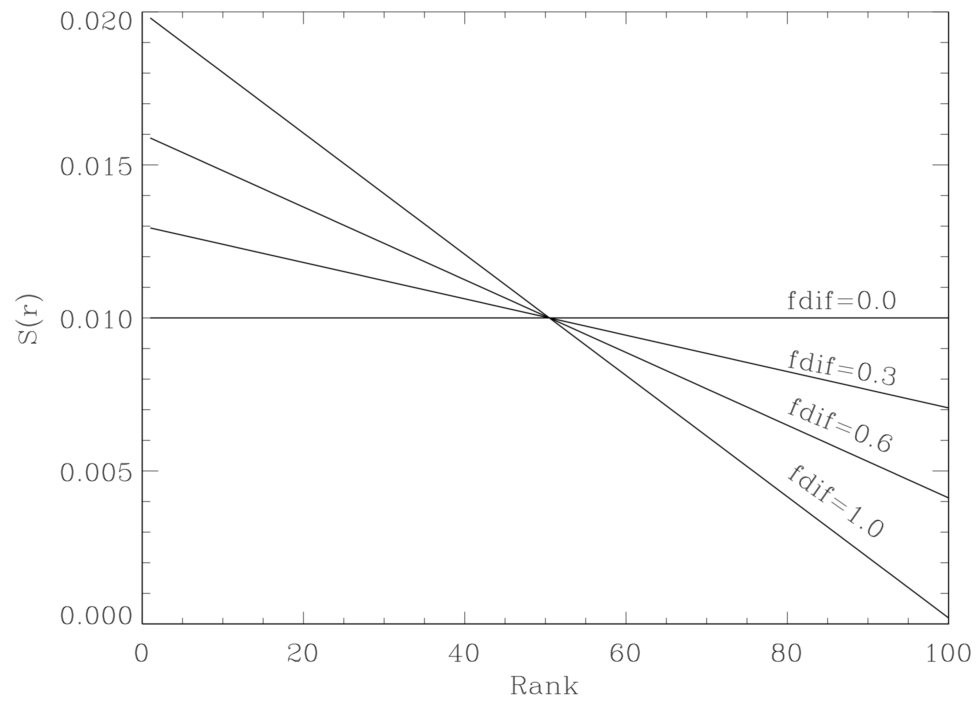

Figure 3.1: Variation of the normalized fitness S' as a function of rank r, for various values of the fitness differential parameter `fdif`. The choice `fdif=1` corresponds to standard ranking, as discussed in §3.4.2.

$F = \sum S' = 1$. Note that the limit `fdif` $\to 0$ returns the same normalized fitness $S' = 1/$`np` for every individual, independently of their rank. This corresponds to a situation where selection pressure is actually nil. The limit `fdif`$\to 1$, on the other hand, yields the standard ranking scheme described in §3.4.2. Figure 3.1 shows a few S' vs r curves computed for four different values of the fitness differential `fdif`. While strong selection pressure (i.e., `fdif`$\to 1$) is generally to be preferred, there are situations in which a weaker selection pressure can help to prevent premature convergence.

Subroutine `select` incorporates this ranking strategy with adjustable fitness differential directly within the rank-based roulette wheel algorithm:

```
subroutine select(np,jfit,fdif,idad)
np1 = np+1
dice = urand()*np*np1              [faites vos jeux...        ]
rtfit = 0.
do 1 i=1,np                        [this is the running sum   ]
   rtfit=rtfit+np1+fdif*(np1-2*jfit(i)) [compute relative
   if (rtfit.ge.dice) then          fitness on the fly        ]
```

```
            idad=i                          [a parent has been found   ]
            goto 2                          [...so no need to continue ]
         endif
    1 continue
    2 return
```

The fitness differential parameter `fdif` is an input quantity in `PIKAIA`, and is held constant for the duration of the evolutionary run.

3.5 Encoding and decoding

The encoding process produces, for each selected parent, a chromosome-like structure that will subsequently be used for breeding through the action of the various genetic operators to be discussed further below. The complementary process of decoding is the equivalent of development and growth in biology, i.e., the reconstruction of an individual from its defining genetic material.

More pragmatically, the aim of the encoding process is to produce a "chromosome" from the `n` parameters defining the function $f(\mathbf{x})$ to be maximized. Write these as

$$\mathbf{x} \equiv (x_1, x_2, ...x_{\rm n}). \tag{3.10}$$

`PIKAIA` encodes these parameters using a decimal alphabet, namely the simple 1-digit base 10 integers[1]. Schematically,

$$x_k \in [0.0, 1.0] \qquad \rightarrow \qquad X_k = (X_1, X_2, ..., X_{\rm nd})_k, \tag{3.11}$$

where the $X_j \in [0, 9]$ are positive integers. The encoding algorithm is simply:

$$X_j = \mathrm{mod}(10^{\rm nd-j+1} x_k, 10), \qquad j = 1, 2, ..., \mathtt{nd}, \tag{3.12}$$

[1] Many public-domain genetic algorithms make use of binary encoding, and some practitioners believe that a binary representation is inherently superior, while empirical evidence suggests that other encoding schemes do at least as well on many classes of problems (see for example Wright 1991; Michalewicz 1994, chap. 5). The present choice was in part motivated by the fact that manipulation of binary strings is not at all easy to implement in a platform-independent manner in standard FORTRAN-77. One may also recall that all known life forms, with the exception of some classes of viruses, are encoded in base four, the digits being the nucleotide bases Adenine, Guanine, Cytosine, and Thymine. Remember your high school biology...

where the function $\mathrm{mod}(x, y)$ returns the remainder of the division of x by y. Each of the `n` defining parameters thus becomes a sequence of `nd` 1-digit integers, so that the encoding of all `n` parameters to `nd` significant digits produces a 1-D integer array (or "chromosome") of length `n`×`nd`. Each element of this array can be thought of as a "gene" having 10 possible alleles. For each encoded parameter, the complementary decoding process is simply

$$x_k = \frac{1}{10^{\mathrm{nd}}} \sum_{j=1}^{\mathrm{nd}} X_j \times 10^j. \tag{3.13}$$

These two operations are carried out by the subroutines `encode` and `decode`:

```
  subroutine encode(n,nd,ph,gn) [ph=phenotype, gn=genotype ]
  z=10.**nd
  ii=0
  do 1 i=1,n                    [n parameters to encode    ]
     ip=int(ph(i)*z)            [convert to integer        ]
     do 2 j=nd,1,-1             [nd genes per parameter    ]
        gn(ii+j)=mod(ip,10)     [extract gene              ]
        ip=ip/10
2    continue
     ii=ii+nd                   [next block of nd genes    ]
1 continue

  subroutine decode(n,nd,gn,ph) [gn=genotype, ph=phenotype ]
  z=10.**(-nd)
  ii=0
  do 1 i=1,n                    [n parameters to decode    ]
     ip=0
     do 2 j=1,nd                [nd genes per parameter    ]
        ip=10*ip+gn(ii+j)       [sum the nd contributions  ]
2    continue
     ph(i)=ip*z                 [i th parameter now decoded]
     ii=ii+nd                   [next block of nd genes    ]
1 continue
```

Note that for efficiency reasons, these two subroutines are not direct transcriptions of eqs. (3.12) and (3.13) above, but one may easily verify that they are equivalent.

Consider the task of maximizing a function $f(x, y)$ of two variables, as in the test-problem used as installation check in §2.2. In this case an individual (or "phenotype") is a point (x, y) in 2-D parameter space. The encoding process would produce

$$(x, y) = (0.34567890, 0.23456789) \quad \rightarrow \quad \texttt{3456789023456789}$$

for `nd`=8. The chromosome `3456789023456789` is made up 16 genes, and is the full genotype of the phenotype (x, y). The number of digits retained in the encoding/decoding, `nd`, is an input quantity that remains fixed throughout the run.

3.6 The crossover operator

The crossover operator is, in essence, what distinguishes genetic algorithms from other heuristic search techniques. `PIKAIA` incorporates a single crossover operator known as *one-point crossover.* This operator acts on a *pair* of parent-chromosomes to produce a *pair* of offspring-chromosomes. Consider again two prototypical "parents" for the test-problem of §2.2:

$(x, y)_1$=(0.34567890,0.23456789)
$(x, y)_2$=(0.87654321,0.65432198)

Encoding to eight significant digits (`nd`=8) would produce the corresponding parent-chromosomes:

```
3456789023456789
8765432165432109
```

The crossover operation begins by randomly selecting a cutting point along the chromosomes, for example by generating a random integer K ∈[1,`n`×`nd`], and cutting both parent chromosomes at the corresponding locus. For example, for $K = 10$:

```
3456789023456789
8765432165432109
.........|......
```

```
345678902 | 3456789
876543216 | 5432109
```

The chromosomal fragments located right of the cutting point are then interchanged and concatenated to the fragments left of the cutting points:

```
345678902 | 5432109    →    3456789025432109
876543216 | 3456789    →    8765432163456789
```

The two strings resulting from this operation are the offspring chromosomes. These two chromosomes decode into the two offspring phenotypes:

(x_1, y_1)=(0.34567890,0.25432198)
(x_2, y_2)=(0.87654321,0.63456789)

The resulting offspring in general differ from either parent, although they do incorporate intact "chunks" of genetic material from each parent.

In practice the crossover operator is applied only if a probabilistic test yields true. Define first a crossover rate `pcross` $\in [0.0, 1.0]$, and generate a random number $R \in [0.0, 1.0]$. The crossover operator is then applied only if $R \leq$`pcross`. If $R >$`pcross`, the two offspring remain exact copies of the two parents.

The crossover operation, including the probability test, is carried out in subroutine `cross`:

```
      subroutine cross(n,nd,pcross,gn1,gn2) [gn1 are gn2 the
                                              parent chromosomes]
      if (urand().lt.pcross) then        [probability test      ]
         ispl=int(urand()*n*nd)+1        [choose cutting point  ]
         do 10 i=ispl,n*nd               [exchange genes located
            t=gn2(i)                      right of the cutting
            gn2(i)=gn1(i)                 point, directly into
            gn1(i)=t                      parent chromosomes    ]
   10    continue                        [gn1 and gn2 are now the
      endif                               offspring chromosomes ]
```

The crossover rate `pcross` is an input quantity, and remains constant throughout the evolution.

3.7 The mutation operator

`PIKAIA` incorporates a single mutation operator known as *uniform one-point mutation*, but allows the mutation rate to vary dynamically in the course of the evolutionary run.

3.7.1 Uniform mutation

The mutation operator functions as follows. For *each* gene of an offspring chromosome, a random number $R \in [0.0, 1.0]$ is generated, and mutation hits the gene only if $R \leq$ pmut, where pmut$\in [0.0, 1.0]$ is the mutation rate. The mutation itself consists in replacing the targeted gene by a random integer $K \in [0, 9]$. These operations are carried out in subroutine mutate:

```
    subroutine mutate(n,nd,pmut,gn)   [gn is a chromosome     ]
    do 10 i=1,n*nd                    [n*nd gene to test      ]
       if (urand().lt.pmut) then      [probability test       ]
          gn(i)=int(urand()*10.)      [mutation hitting gene i]
       endif
 10 continue
```

One may note that even though the mutation operator acts uniformly on the genotype, its phenotypic effects can vary by orders of magnitude, depending on the gene being affected. An important feature of the crossover and mutation operators is that they preserve the parameter bounds imposed by the encoding/decoding process. In other words, applying these two operators to chromosomes encoding parameters $\{x_k\} \in [0.0, 1.0]$ can only produce offspring chromosomes encoding parameters bounded in the same interval.

3.7.2 Dynamic adjustment of the mutation rate

The action of the mutation operator has both deleterious and beneficial effects. It can destroy a potentially superior offspring produced by the crossing of two above-average parents, yet it is required to preserve variability in the population and is often the only mechanism available to save the day if premature convergence on a secondary extremum were to occur. Experience shows that the choice of an optimal value of the mutation rate pmut is very problem-dependent, and in general cannot be made *a priori*.

PIKAIA can monitor the degree of convergence in the population, and adjust the mutation rate accordingly. Using the true fitness S (as opposed to the normalized fitness S') of the best and median individuals, define the quantity

$$\Delta S = \frac{S(r=1) - S(r=\texttt{np}/2)}{S(r=1) + S(r=\texttt{np}/2)} \tag{3.14}$$

as a measure of the degree of convergence of the population. The mutation rate is increased (lowered) whenever this quantity is smaller (larger) than a predetermined level. This is carried out in subroutine adjmut:

```
      subroutine adjmut(np,fitns,ifit,pmutmn,pmutmx,pmut)
      parameter  (rdiflo=0.05, rdifhi=0.25, delta=1.5)
      rdif=abs(fitns(ifit(np))-fitns(ifit(np/2)))[compute normalized
     +        /(fitns(ifit(np))+fitns(ifit(np/2))) spread in fitness]
      if(rdif.le.rdiflo)then                      [increase pmut    ]
         pmut=min(pmutmx,pmut*delta)
      else if(rdif.ge.rdifhi)then                 [decrease pmut    ]
         pmut=max(pmutmn,pmut/delta)
      endif
```

pmutmx and pmutmn are upper and lower bounds, respectively, to the allowed range of mutation rates. The parameters rdiflo and rdifhi are the upper and lower critical values of ΔS at which the adjustment process is activated. As long as rdiflo $\leq \Delta S \leq$ rdifhi, the mutation rate pmut remains equal to its input value. Whenever $\Delta S \leq$ rdiflo ($\Delta S \geq$ rdifhi), the mutation rate is increased (decreased) by a multiplicative factor delta.

The mutation rate (pmut) is an input quantity. If a variable mutation rate is used, then the upper (pmutmx) and lower (pmutmn) bounds on the allowed range of mutation rates must also be provided as input quantities. PIKAIA then updates the mutation rate at the end of each generational iteration, through a call to subroutine adjmut. Note that the limits rdifhi and rdiflo, as well as the increment delta, are constants set in a PARAMETER statement in subroutine adjmut.

3.8 Reproduction plans

A reproduction plan controls the ways in which newly bred individuals are to be incorporated in the population. PIKAIA operates on a fixed population size, and offers a choice of three reproduction plans:

3.8.1 Full generational replacement

This is perhaps the simplest reproduction plan. Throughout one iteration of the generation cycle, offspring are accumulated in temporary storage (by successive calls to subroutine genrep). Once np offspring have been so produced and stored, the entire parent population is wiped out and replaced by the offspring population (by a single call to subroutine newpop), after which a new generational iteration begins. Subroutine newpop also computes the fitnesses for the members of the new population, and computes their respective ranks. Under this reproduction plan, individuals have a fixed lifetime equal to a single generation.

3.8.2 Steady-state plans

Steady-state reproduction plans insert individuals as they are being bred. Criteria must be specified to decide

(1) under which conditions newly-bred offspring are to be inserted,
(2) how members of the parent population are to be deleted to make room for the new members, and
(3) if any limit is to be imposed on an individual's lifetime.

PIKAIA incorporates two steady-state plans. In both cases a newly bred offspring is inserted whenever its fitness exceeds that of the least fit member of the parent population, unless it is identical to an existing member of the population[2]. Furthermore, PIKAIA imposes no limit on the generational lifetime of a population member; a very fit individual can survive through many iterations of the generational cycle. The two plans differ in how room is made to accommodate the offspring to be inserted. Under the *steady-state-delete-worst* plan, the least fit member of the parent population is eliminated and replaced by the offspring. Under the *steady-state-delete-random* plan, a member of the old population is chosen at random and deleted, independently of its fitness.

The insertion and deletion steps are carried out in subroutine `stdrep` for both steady-state plans. Note that PIKAIA internally defines a generational iteration as the production of `np` individuals, independently of how many actually end up being inserted in the population. Under steady-state plans, a significant amount of bookkeeping is required to update the ranking arrays as new individuals are being inserted. This bookkeeping makes subroutine `stdrep` look more intricate than it really is; the alternative would have been to recompute all ranks every single time a new individual is inserted, but this appeared overly wasteful, even given our advocated preference for clarity over efficiency.

3.8.3 Select-Random-Delete-Worst plan

One interesting class of steady-state reproduction plans picks parents completely at random, with natural selection enforced in making room for new offspring: the probability of being targeted for termination is made *inversely* propor-

[2] This "no-duplicate" criterion is an important safeguard against inbreeding. It is particularly important under steady-state plans, since a very fit parent breeding without crossover or mutation —a possible occurrence if `pcross`$<$ 1.0 and `pmut`$\ll$ 1.0— would lead to the insertion of a copy of itself in the population, after which breeding with that copy —a likely occurrence if the phenotype is very fit— could produce two new identical copies *even if crossover were to occur*. It's all downhill from there on...

tional to fitness. This strategy has been found advantageous for large problems tending to exhibit premature convergence (see, e.g., Cedeño *et al.* 1994). It turns out that one extreme version of such reproduction plan is effectively already built in `PIKAIA`; a *select-random-delete-worst* reproduction plan can be produced by running `PIKAIA` with `fdif`=0 —vanishing selection pressure— under the steady-state-delete-worst reproduction plan (`irep=3`).

Clearly a number of alternate steady-state plans can be construed. The three reproduction plans available in `PIKAIA`, including the select-random-delete-worst variant described here, represents a good sampling of the range of possibilities.

The choice of a reproduction plan is controlled by the input parameter `irep`. The choice `irep`= 1 enforces full generational replacement, `irep`= 2 steady-state-delete-random, and `irep`= 3 steady-state-delete-worst. `PIKAIA` does not permit switching from one reproduction plan to another during the evolution.

3.9 Elitism

In view of the disruptive effects of the crossover and mutation operators, the possibility exists that the genotype of the fittest individual will not be passed on intact to the next generation. Only under the steady-state-delete-worst reproduction plan is the fittest guaranteed to survive. The strategy known as *elitism* alleviates this potential obstacle to efficient convergence.

How the technique operates depends on the reproduction plan being used. Under full generational replacement, the technique consists in simply saving in temporary storage the fittest individual of the parent population, and artificially reinserting it at the end of the generational iteration. This procedure is carried out in subroutine `newpop`, a listing of which is now in order:

```
      subroutine newpop
     +   (ff,ielite,ndim,n,np,oldph,newph,ifit,jfit,fitns,nnew)
      if(ielite.eq.1 .and. ff(n,newph(1,1)).lt.fitns(ifit(np)))then
         do 1 k=1,n                           [reinsert fittest of
            newph(k,1)=oldph(k,ifit(np))       previous generation  ]
    1    continue
         nnew = nnew-1
      endif
      do 2 i=1,np                             [off with their heads!]
         do 3 k=1,n
            oldph(k,i)=newph(k,i)
    3    continue
         fitns(i)=ff(n,oldph(1,i))            [compute fitness ]
```

```
 2 continue
   call rnkpop(np,fitns,ifit,jfit)    [rank new population]
```

The protected individual is (arbitrarily) inserted in the first column of the population matrix `oldph`, unless it just so happens that the generational iteration has produced a first offspring that is actually fitter than the fittest one from the previous generation; in this case insertion of the protected offspring is simply not carried out.

Under steady-state-delete-random, the use of elitism artificially protects the fittest member of the population against random deletion. Elitism is not required under steady-state-delete-worst, as this plan effectively corresponds to a generalized and rather extreme form of elitism.

The use of elitism is controlled by the input parameter `ielite`. Elitism is either on (`ielite`$= 1$) or off (`ielite`$= 0$) for the entire duration of the evolutionary run.

4. USING PIKAIA

4.1 Calling sequence

The calling sequence for `PIKAIA` is

```
call PIKAIA(funk,n,ctrl,xb,fb,status)
```

`funk` is the name of a user-supplied external function to be maximized, and `n` is the parameter space dimension, i.e., the number of adjustable parameters in `funk`. The maximum allowed size for `n` is set in a `PARAMETER` statement within `PIKAIA`, and is equal to 32 in the source code. Ideally, the user should set this value equal to the largest expected problem size, but no larger, so that `PIKAIA` does not defines its internal arrays larger than necessary (FORTRAN-77 lacks dynamic storage allocation).

4.2 Input/Output

The floating point array `ctrl` has length 12, and contains flags and parameter values that control `PIKAIA`'s evolutionary behavior; a detailed description of each element of `ctrl` is given below.

The array `xb` and scalars `fb` and `status` are the only output returned by `PIKAIA`. The floating point array `xb` has length `n`, and contains the parameters defining the best solution found by `PIKAIA` at the end of the evolutionary run. The scalar `fb` is the corresponding fitness, as would be returned by `funk` with `xb` as argument. As its name implies, the output variable `status` contains, upon return, a numerical value coding error conditions or successful termination.

4.3 Fitness function

The function `funk` to be maximized must be declared as `REAL` and `EXTERNAL` in the calling program. `funk` *must* accept as argument a single integer `n` (the dimension of parameter space) and a single floating point array of length `n` (a point in parameter space):

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

The evaluation of `funk` *must* return a positive-definite quantity that measures fitness (high/low values ≡ high/low fitnesses, goodness of fit, etc.). Recall that `PIKAIA` searches a bounded nondimensional search space spanning the range [0.0,1.0] in all `n` dimensions of parameter space. `funk` *must consequently carry out internally all appropriate scalings of dimensional variables.* Likewise, all values in the "best phenotype" array `xb` returned by `PIKAIA` upon successful termination (`status=0`) are in the range [0.0,1.0] and so must be rescaled to dimensional values in the same way.

4.4 Internal safety checks and Error/Warning messages

`PIKAIA` first calls subroutine `setctl` to perform a minimal set of run-time tests on its input parameters. If any element of the control vector `ctrl` is *negative*, then `PIKAIA` will supply its own default value(s) for the corresponding parameter(s). If any invalid but positive values are supplied, `PIKAIA` aborts and returns with a positive (non-zero) value for the `status` output variable. The actual value is equal to the index of the input vector `ctrl` containing an illegal value for the corresponding control parameter (more on `ctrl` in §4.5 below). `PIKAIA` also verifies that the population size specified on input is an even number; if not, the population size is reduced by one and a warning message issued.

One or many warning messages will also be issued if some combinations of parameter values, while formally legal, risk producing an inefficient algorithm. It is important to realize that only a very basic set of such tests is carried out, as optimal parameter settings are always to some extent a function of the problem under consideration. Consequently, it is impossible to guarantee that all potentially inefficient combinations of parameters will elicit a warning from `PIKAIA`.

Upon successful termination `PIKAIA` returns with `status`=0. Additional run-time information can be routed to standard output by choosing appropriate settings for the control flag `ivrb`, as described below.

4.5 The input control vector `ctrl`

Internally, `PIKAIA` associates the control vector `ctrl` to the following flags and parameters:

$$\mathtt{ctrl}(1:12) =$$

$$(\mathtt{np}, \mathtt{ngen}, \mathtt{nd}, \mathtt{pcross}, \mathtt{imut}, \mathtt{pmut}, \mathtt{pmutmn}, \mathtt{pmutmx}, \mathtt{fdif}, \mathtt{irep}, \mathtt{ielite}, \mathtt{ivrb})$$

These correspond to the following:

Population number [np ($\equiv$ctrl(1)); default is np=100; see §3.2]: The number of individuals in the population. Note that this remains constant throughout the run. The population size is internally restricted to np$\leq$ 512 and should be an even number.

Number of Generations [ngen ($\equiv$ctrl(2)); default is ngen=500; see §3.2]: PIKAIA evolves the population over a pre-determined number of generations set by the value of the parameter ngen, instead of trying to meet a preset tolerance criterion. The latter approach is potentially dangerous when approaching a new problem, in view of the usual convergence trends exhibited by GA-based optimizers (more on this below).

Encoding accuracy [nd ($\equiv$ctrl(3)); default is nd=5; see §3.5]: This sets the number of digits retained in encoding the phenotype into a genotype. This is internally restricted to nd$\leq$ 6 (the typical 32-bit floating point accuracy is only 6 or 7 decimal places); in most real applications, if more than 4 digits of accuracy are required it would generally be preferable to use a 4 digit-accurate genetic solution as a starting guess for a more conventional optimization method. Recall that the genotype ends up being an integer array of length n*nd, where each element ("gene") takes values in the range $[0, 9]$. The encoding scheme used in PIKAIA is clearly far from optimal in terms of efficient use of storage, but it is easy to code, understand, modify, and keep track of when something goes wrong.

Crossover rate [pcross ($\equiv$ctrl(4)); default is pcross=0.85; see §3.6]: Once two parents have been selected for breeding, a random number $R \in [0.0, 1.0]$ is generated, and the crossover operation is applied only if $R \leq$pcross.

Mutation mode [imut($\equiv$ctrl(5)); default is imut=2; see §3.7]: Integer flag controlling the behavior of the mutation operator. Setting imut=1 enforces a constant mutation rate, at a value set by pmut (see below). For imut=2, the mutation rate varies in the range [pmutmn,pmutmx] throughout the evolution, with starting value pmut. The mutation rate increases (decreases) only when the relative difference in the absolute fitnesses of the best and median member of the population falls below (exceeds) the value rdiflo (rdifhi). The mutation is varied by logarithmically constant increments delta, i.e. pmut$\leftarrow$pmut*delta (pmut/delta). The values rdiflo=0.05, rdifhi=0.25 and delta=1.5 are set in a PARAMETER statement in subroutine adjmut, inspection of which should further clarify how the variable mutation rate is implemented. Depending on the fitness contrast in parameter space, the user may wish to adjust the values of rdifhi and rdiflo.

Initial mutation rate [pmut ($\equiv$ctrl(6)); default is pmut=0.005; see §3.7]: By convention, the value of pmut corresponds to the probability ($\leq$ 1) that a given *gene* be affected by a mutation at breeding. For every gene a random number $R \in [0.0, 1.0]$ is generated, and mutation is carried out only if $R \leq$`pmut`. The mutation itself consists in generating a random integer $K \in [0, 9]$, and resetting the gene value to K; note that there is 1 in 10 chance that K will be equal to the original gene value, in which case mutation has effectively no phenotypic effect.

Minimum mutation rate [`pmutmn` (≡`ctrl(7)`); used only if `imut=2`; default is `pmutmn=0.0001`; see §3.7.2]: Minimum mutation rate attainable under variable mutation mode.

Maximum mutation rate [`pmutmx` (≡`ctrl(8)`); used only if `imut=2`; default is `pmutmx=0.1`; see §3.7.2]: Maximum mutation rate attainable under variable mutation mode. Typically, `pmut` and/or `pmutmx` must be much smaller than unity, otherwise near complete randomization is likely to occur in every offspring. Under full generational replacement (see *reproduction plans* below) and unless elitism has been turned on (by setting `ielite=1`), `PIKAIA` will issues warnings if `pmut`>0.05, or if `imut=2` and `pmutmx`>0.05.

Fitness differential [`fdif` (≡`ctrl(9)`); default is `fdif=1.0`; see §3.4]: `PIKAIA` makes use of ranking to assign fitness. Individuals are first ranked as $[1, 2, ..., \mathtt{np}]$, according to "true" fitness, where by definition the fittest individual has rank `1` and the least fit rank `np`. respectively. The breeding probability of the $i^{\rm th}$ individual in the population is then defined as

$$\frac{1}{\mathtt{np}} + \frac{\mathtt{fdif}}{\mathtt{np}} * \left(1 - \frac{2 * \mathtt{jfit(i)}}{\mathtt{np} + 1}\right),$$

where `jfit(i)` is the rank of the individual stored in the $i^{\rm th}$ column of the population array `oldph`, as described before. This defines a *linear* relationship with slope `fdif` between rank and breeding probability. Note in particular that setting `fdif=0.0` corresponds to no selection pressure (i.e., equal breeding probability for every individual), while `fdif=1.0` directly equates breeding probability to rank. Unless `irep=3` (see below and §3.8.3), `PIKAIA` will issue a warning if `fdif` is set to a value less than 1/3, which would correspond to a fitness differential too low for most practical applications. All breeding probability calculations are carried out internally in subroutine `select`.

Reproduction plans [`irep` (≡`ctrl(10)`); default is `irep=1`; see §3.8]: Integer flag controlling the choice of either one of the three available reproductive plans. Setting `irep=1` selects full generational replacement (§3.8.1), and `irep=2` or `irep=3` steady-state reproduction (§3.8.2). In these latter cases an offspring is inserted only if (1) its fitness is superior to that of the *least fit* population member, and (2) its genotype differs in at least one gene from any genotype already present in the population. The two steady-state reproduction plans differ only in how individuals from the old population are deleted to make room for new offspring fit enough for insertion; under `irep=3`, the least fit is deleted (steady-state-delete-worst plan). If on the other hand `irep=2`, an individual from the old population is chosen at random and deleted, independently of its actual fitness (steady-state-delete-random plan). When operating under a steady-state reproduction plan, a "generation" is not a well-defined concept. Internally, `PIKAIA` defines a generation as a group of `np` individuals. One should note, in particular, that if a user-supplied output subroutine is inserted within the generation loop in `PIKAIA` (see §6.2 below), this routine will be called once every time `np` individuals have been bred, independently of how many of them have actually been inserted in the population. Likewise, the mutation rate is adjusted (under `imut`=2) only once at the end of each generational iteration. As a rule of thumb, going from `irep=1` through `irep=2` to `irep=3` with `fdif` fixed represents, to some extent, a transition from enhanced *exploration* to enhanced *exploitation* (for fixed selection pressure), but neither plan seems to be clearly superior to the others in a general sense (see, e.g., Syswerda 1991). All reproduction plans operate under the assumption of a fixed-sized population, and make use of the roulette wheel algorithm for parent selection (see §3.4 herein; also chap. 1 of Davis 1991).

Elitism [`ielite` (≡`ctrl(11)`); default is `ielite=1`; see §3.9]: integer flag controlling the use of elitism. Elitism is enforced if `ielite=1`, otherwise no action is taken. Under `irep=2`, setting `ielite=1` ensures that the fittest individual cannot be selected for random deletion. This flag has no effect under `irep=3` (steady-state-delete-worst reproduction plan).

Verbose mode [`irvb` (≡`ctrl(12)`); default is `ivrb=0`]: integer flag controlling the generation of additional run-time standard output. Setting `ivrb=1` or `ivrb=2` generates a listing of input parameters, as well as information concerning the current status of the population. The latter output is generated by subroutine `report`. The first line of output generated by `report` contains (1) the generation count, (2) the number of individuals inserted in the last round of breeding activity, (3) the fitnesses of the best, second and median individuals in the current population. This is followed by `n` lines listing the phenotypes for the best, second and median individuals. The following is part of the output produced by running the installation check test problem with `ivrb=1`:

```
26     99  0.111111  0.972997  0.951311  0.545382
                        50576     40386     30042
                        49914     55441     75020
```

This was produced at the end of the 26$^{\text{th}}$generational iteration; 99 individuals have been inserted (not 100, because elitism is used), and the current mutation rate is `pmut`$= 0.111111$. The best individual is $(x, y) = (0.50576, 0.49914)$, for which $f(x, y) = 0.972997$. The second and median individual have $f(0.40386, 0.55441) = 0.951311$ and $f(0.30042, 0.75020) = 0.545382$, respectively. Under `ivrb=2` this information is printed at every generation. Under `ivrb=1` it is printed only if either (1) the mutation rate has been adjusted (under `imut=2`), or (2) the fitness of the best individual has improved since the last generation (under `irep=1`) or pseudogeneration (under `irep=2` or `irep=3`).

Whenever an element of the control vector `ctrl` is set to a negative value on input, `PIKAIA` will use its default values for the corresponding control parameter. All default values are set by a `PARAMETER` statement in subroutine `select`. The following Table summarizes the allowed and default values of the input parameters contained in the control vector `ctrl`:

Table I

Elements of input control vector

Element	Internal variable	Type	Legal values	Default values	Note
1	`np`	integer	≤ 128	100	1
2	`ngen`	integer	$< \infty$	500	1
3	`nd`	integer	≤ 6	5	1
4	`pcross`	real	$0.0 \leq$ `pcross` ≤ 1.0	0.85	2
5	`imut`	integer	1, 2	2	
6	`pmut`	real	$0.0 \leq$ `pmut` ≤ 1.0	0.005	2
7	`pmutmn`	real	$0.0 \leq$ `pmutmn` ≤ 1.0	0.0005	2
8	`pmutmx`	real	$0.0 \leq$ `pmutmx` ≤ 1.0	0.25	2
9	`fdif`	real	$0.0 \leq$ `fdif` ≤ 1.0	1.0	2
10	`irep`	integer	1, 2, 3	1	
11	`ielite`	integer	0, 1	1	
12	`ivrb`	integer	0, 1, 2	0	

Notes:
1= Maximum value set internally by a `parameter` statement within `PIKAIA`
2= Some values, while formally legal, can produce an inefficient algorithm; see main text.

4.6 Additional User-supplied functions and subroutines

4.6.1 Random number generator

`PIKAIA` requires a random number generator function that returns random or pseudo-random deviates from a uniformly distributed sequence in the interval [0.0,1.0]. This function *must* be called `urand` and *must* be declared and called without arguments, i.e., `function urand()`, `r=urand()`, and so on. Any required initialization of `urand` should also be carried out by the calling program. The use of generic system-supplied random number generators is *not* recommended.

`PIKAIA` is distributed with a simple random number generator based on the "minimal standard" Lehmer multiplicative linear congruential generator of Park and Miller (1988). This is essentially identical to the random number generator `ran0` of Press *et al.* (1992, §7.1), but Park and Miller's revised multiplier (Park, Miller, and Stockmeyer, 1993) is used. We chose the Park-Miller generator because of its simplicity and portability: for a given `seed` value, it will produce the same sequence of uniform random deviates (to within floating point precision differences) on any machine with 32-bit integer arithmetic.

One of the beauties of genetic algorithms is that in contrast to Monte Carlo methods, they are inherently robust with respect to the choice of random number generator. That is, the search space is not restricted to the finite sequence of random deviates supplied by the random number generator; the search space consists of the essentially infinite (up to the machine's floating point precision) set of all combinations and permutations of the "genes" decoded from the random deviates used to construct the initial population (cf. §3.3).

In view of this inherent robustness, we chose to make one short cut in the use of our random number generator which one would definitely not want to make with Monte Carlo methods: we use the same sequence of random deviates in all calls, independently of where in the code those calls originate. The theory which establishes the favorable characteristics of a particular random number generator applies of course only to the full sequence of deviates. Thus in order to guarantee those favorable characteristics at any one invocation of the random number generator, one must ensure that that sequence is not used (i.e., diverted) anywhere else in the same program. In principle each invocation of the random number generator should maintain its own seed variable (e.g., using Fortran's `SAVE` statement). We do not do this in `PIKAIA`. For simplicity of coding we always use calls to `urand()` with no seed; the random number generator is initialized once at the beginning of a run (with a seed supplied by the user), and all subsequent calls draw from the same single sequence of random deviates. Again, in an important production code we would probably, just for comfort's sake, use a better random number generator (such as Press *et al.*'s `ran2` subroutine) and we would use a separate sequence for each invocation. However, based on our experience to date with the use of `PIKAIA`, we strongly suspect that such improvements would make no noticeable difference in code performance whatsoever.

Listings of our random number generator `ran0`, initialization subroutine `rninit`, and interface `urand` are provided in §A.2 of the Appendix.

4.6.2 Ranking subroutine

The user must supply a subroutine that accepts as input a floating point array `arrin` of length `n`, and return an integer array `indx` of identical length, where element `indx(i)` is a key index giving the location in `arrin` of the `i`$^{\mathrm{th}}$ smallest value in `arrin` (this is equivalent to the integer array `ifit` described above). This subroutine is called only in subroutine `rnkpop`.

`PIKAIA` is distributed with a ranking subroutine which uses the Quicksort algorithm (Hoare, 1962). We use our own implementation, based on Niklaus Wirth's `quicksort1` algorithm (Wirth, 1986), which is essentially identical to the subroutine `indexx` of Press *et al.* (1992, §8.4). A listing of our `rqsort` subroutine is provided in the Appendix, §A.2.

4.7 Efficiency considerations

The global structure of a properly designed genetic algorithm, as well as the various operators and ecological strategies it incorporates, can be made essentially independent of the actual problem being tackled. The only point of contact between `PIKAIA` and a given problem is the computation of an individual's fitness, through the user-supplied fitness function `funk`. The range of applicability of genetic algorithm is consequently immense. As long as an unambiguous measure of fitness can be constructed, `PIKAIA` can be used to solve *any* numerical optimization problem without any significant alteration to the code. Which does not at all mean that it should be; `PIKAIA` can be used to invert a matrix, but this would likely yield the most inefficient matrix inversion method ever construed... Rule number one is, if you already have something that works and works well enough, don't mess with it! However, on problems where more conventional techniques fail, `PIKAIA` may well produce an acceptable solution, *given enough time*. Genetic algorithm-based optimizers do not require excessive amounts of memory (no large matrices need to be stored/inverted), but can be CPU-intensive, although in general a lot less than Monte Carlo or purely enumerative techniques.

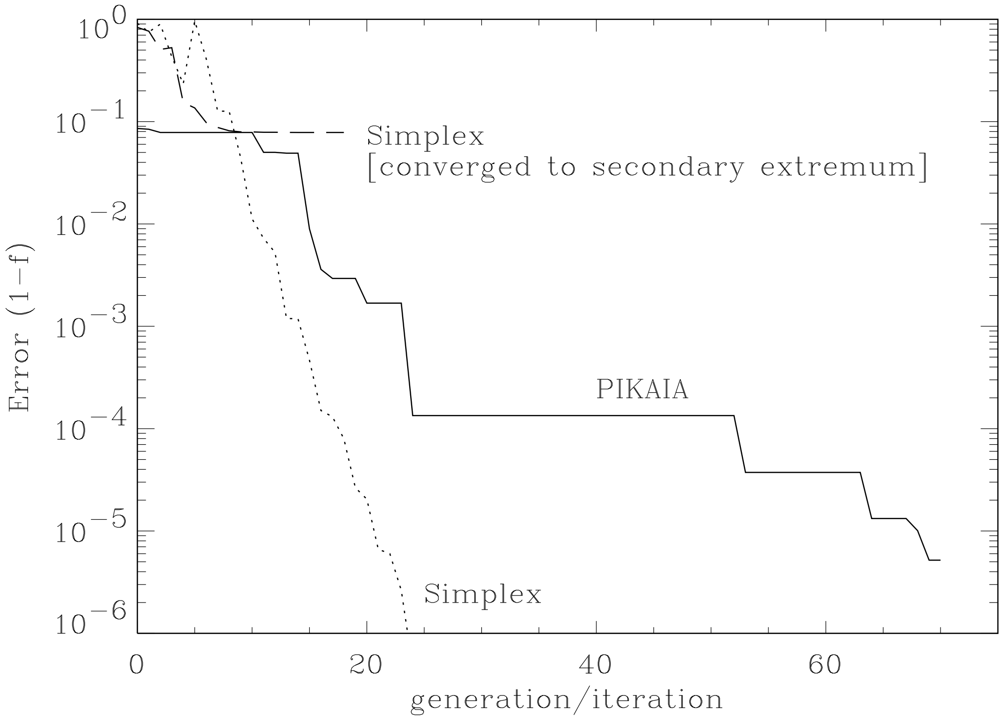

Figure 4.1: Convergence curves for the test problem of §2.2. For this maximization problem the "target" is $f(x, y) = 1.0$. The solid line is a genetic solution obtained using `PIKAIA`'s default settings. The dotted and dashed lines are two representative curves obtained using the simplex method (see text). Note the markedly different convergence behaviors.

4.7.1 Convergence rate and hybrid schemes

The solid line on Fig. 4.1 is a convergence curve for the installation test-problem of §2.2, obtained using `PIKAIA`'s default settings. The general shape of the convergence curve —relatively low initial error as a consequence of sampling by a population of trial solutions, followed by intermittent, abrupt lowering of the error separated by periods of no apparent progress— is actually quite typical of the behavior of the algorithm on more realistic problems. Moreover, this behavior is markedly different from that associated with more conventional optimization methods, such as for example conjugate gradient-based schemes; the first few iterations of such schemes often do not produce a significant (or monotonic) drop of the error, but subsequent iterations often see the error begin to decrease at a fixed rate, which can often be estimated *a priori*.

The *simplex method* (Nelder & Mead, 1965; see also Press *et al.* 1992, §10.4) is a rather robust optimization method that does not require derivative computation.

Like many other optimization methods, it is prone to getting stuck on secondary extrema. The dotted and dashed lines on Fig. 4.1 are convergence curves for two different solutions obtained using the simplex method. As advertised, the simplex method sometimes gets stuck on secondary extrema (dashed line), but when it does not (dotted line), it converges much faster than `PIKAIA`. Figure 4.1 is actually quite unfair to the simplex method, as 20 of its iterations require about as much CPU time as a single generational iteration within `PIKAIA`. However, repeated trials with the simplex method reveal that for this test problem, the true maximum is located only on 14 out of 100 trials (each trial differs in the location and size of the initial simplex), as opposed to 100/100 for `PIKAIA` running on default settings. The simplex method is (relatively) fast, but local; `PIKAIA` is slow, but global. There is no such thing as a free lunch.

The dichotomy "fast but local" versus "slow but global" can however 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 10 generations (say), and then use this individual to initialize the simplex method (or any other 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 relatively high accuracy is required on real-life problems.

4.7.2 Initialization subroutine

Returning now to more pragmatic matters, examination of `PIKAIA`'s top level structure (§3.2) shows that the user-supplied fitness function `funk` will be called at least `np`$\times$`ngen` times in the course of the evolutionary run. Experience shows that in most real-life problems, the bulk of the CPU time is actually spent in `funk`. It is consequently imperative that `funk` be coded in as efficient a manner as possible. In particular, any secondary computation that is independent of the function's arguments should be carried out once and for all before calling `PIKAIA`, and the computed quantities passed to `funk` via one or more appropriately defined `COMMON` block(s).

5. EXAMPLES

The aim of this chapter is to provide detailed examples of the use of PIKAIA for solving a five optimization problems of increasing levels of difficulty. In §5.1 we revisit once again the installation check problem of §2.2. Section 5.2 shows how to use PIKAIA to solve a straightforward linear least squares fit problem, and in §5.3 a more difficult non-linear least square fit problem is presented and various aspects of its solution with PIKAIA are discussed in some detail. Section 5.4 covers the traditionally much (much!) more difficult problem of distance regression, as exemplified by the infamous problem of fitting ellipses to a set of (x, y) data where significant measurement errors exist in both x and y coordinates. Finally, §5.5 illustrate how to use PIKAIA to perform data modeling using generalized, robust estimators that do not rely on a least squares formulation. Example driver programs, fitness functions and synthetic data for these problems are provided as part of PIKAIA's installation package, and are listed in the Appendix below (§A.4). The chapter concludes with a warning concerning the use of the population of solutions to construct error estimates (§5.6), and a few general comments regarding the applicability of GA-based optimizers to other types of modeling problems (§5.7).

5.1 Maximizing a function of two variables

The installation check problem consisted in in locating the global maximum of a multimodal function of two variable $f(x, y)$, in the sense of finding the solution (x^*, y^*) that returns the maximum evaluation of $f(x, y)$. The test function of §2.2 is defined as

$$f(x, y) = \cos^2(n\pi r)\exp(-r^2/\sigma^2), \tag{5.1a}$$

$$r^2 = (x - 0.5)^2 + (y - 0.5)^2, \qquad x, y \in [0.0, 1.0], \tag{5.1b}$$

where n and σ are known constants. Figure 5.1 is a surface plot of this function, for $n = 9$ and $\sigma^2 = 0.15$. The global maximum is at $(x^*, y^*) = (0.5, 0.5)$, with $f(x^*, y^*) = 1.0$. This global maximum is surrounded by concentric rings of secondary maxima, located at radial distances

$$r_{\max} = \{0.110192, 0.220385, 0.330582, 0.440782, 0.550986\}, \tag{5.2}$$

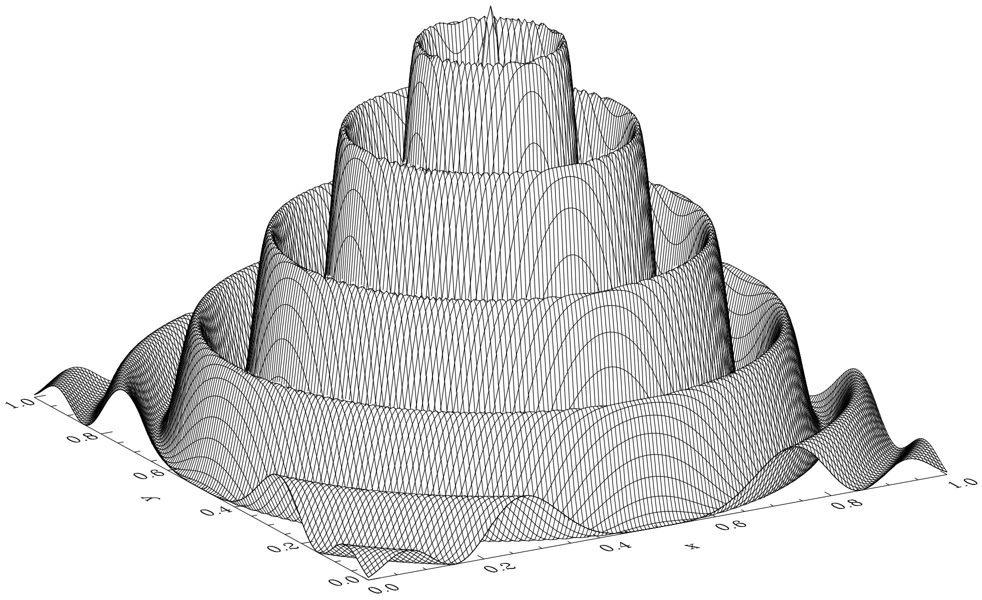

Figure 5.1: Two dimensional surface for the the installation check problem of §2.2, defined by eq. (5.1) with $n = 9$ and $\sigma^2 = 0.15$. The global maximum is at $(x^*, y^*) = (0.5, 0.5)$, with $f(x^*, y^*) = 1.0$.

from the central maximum. Between these are located another series of concentric rings corresponding to minima:

$$r_{\min} = \frac{m - 1/2}{9}, \qquad m = 1, 2, ... \tag{5.3}$$

The innermost secondary maximum ring has $f(x, y) = 0.95147$; this ring is where the installation check code sometimes remains stuck, because it only runs over 100 generations (cf. §2.2). The error associated with a given solution (x, y) can be defined as

$$\varepsilon = |1 - f(x, y)|. \tag{5.4}$$

One may 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 about 1% (this figure would decrease rapidly if eqs. (5.1) were to be generalized to higher dimensions). This seemingly simple global optimization problem would present formidable difficulties for standard techniques such as algorithms based on the conjugate gradient method, unless of course a very good guess at the solution were to be already available. Figure 4.1 already illustrated the possible pitfall associated with the use of the simplex method (a relatively robust method, if only a bit slow compared to most conjugate gradient-based methods): more often that not, the method gets "stuck" somewhere on one of the rings of secondary extrema.

The problem is easily handled by PIKAIA. The search space has dimension 2 (n=2), and all the user needs to provide is a function that accepts as input a vector of dimension two containing an (x, y) pair, and returns the function evaluation corresponding to the RHS of eq. (5.1a) above. Since the optimization task is simply to maximize $f(x, y)$, the value returned by $f(x, y)$ can be used directly as the measure of absolute fitness required to establish the ranks in the population. The population itself consists of np pairs (x, y). Note that the search space [x(1),x(2)]$\equiv [x, y]$ is already bounded in $[0, 1]$ so that no rescaling of the input parameter is required. The corresponding user-supplied function may look something like this:

```
      function  twod(n,x)
      real      x(n)
c=====================================================================
c     Compute sample fitness function (altitude in 2-d landscape)
c=====================================================================
      implicit  none
      integer   n,nn
      real      pi, sigma2, x(n), rr, twod
      parameter (pi=3.1415926536,sigma2=0.15,nn=9)
      if (x(1).gt.1..or.x(2).gt.1.) stop
      rr=sqrt( (0.5-x(1))**2+ (0.5-x(2))**2 )
      twod=cos(rr*nn*pi)**2 *exp(-rr**2/sigma2)
      return
      end
```

A call to PIKAIA, making use of default values for all input parameters, would then simply be

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

with xb defined as being a real array of length 2 in the calling program and all elements of ctrl initialized to some negative value. Recall that the function twod must be declared `EXTERNAL` by the calling program. An example of a complete driver program for this problem is provided in the Appendix (§A.3).

It is instructive at this point to examine the performance of `PIKAIA` for different choices of evolutionary strategies. Table II below lists input parameters settings that force `PIKAIA` to operate under the three different reproduction plans, with or without elitism and variable mutation rates. In all cases a population of `np`= 100 individuals evolved through `ngen`= 200 generations. Default settings were used for all other input parameters (`pcross`, `pmut`, etc.). Because of the inherently stochastic elements present in the algorithm, sequences of 100 runs were computed and averaged, so as to produce results representative of the true performance of the method. The average error ($\bar{\varepsilon}$) at the end of the 100 runs and the success rate at localizing the central maximum are also listed in Table II.

Table II

Run parameters and global performance for first example problem

Run	`ielite`	`imut`	`irep`	$\bar{\varepsilon}$	central max?	note
PK1	0	1	1	6.51×10^{-2}	22/100	
PK2	1	1	1	4.47×10^{-2}	43/100	
PK3	1	2	1	5.96×10^{-8}	100/100	1
PK4	1	2	2	7.93×10^{-4}	99/100	
PK5	1	2	3	1.59×10^{-3}	98/100	

Note: 1= These correspond to `PIKAIA`'s default settings

Figure 5.2 shows the average error $\bar{\varepsilon}$ associated with the best individual as a function of generation count, for the five versions of `PIKAIA` defined in Table II. Perhaps the most striking message conveyed by Figure 5.2 is that *variable mutation is a good thing* (with the important caveat that it be used in conjunction with elitism!). Figure 5.2 also seems to suggest that full generational replacement (version PK3) performs significantly better than either steady-state plans, everything else being equal. Some care is warranted here. Any one run remaining stuck on the first secondary maxima ring will result in an error of order 10^{-3} in the 100-run average, even if all other 99 runs are accurate to 10^{-6} or better. This is precisely what is happening here, as can be seen upon examining the fourth and fifth columns of Table II. Loosely speaking, the latter is a measure of *global* performance. Under variable mutation rate with elitism, all three reproduction

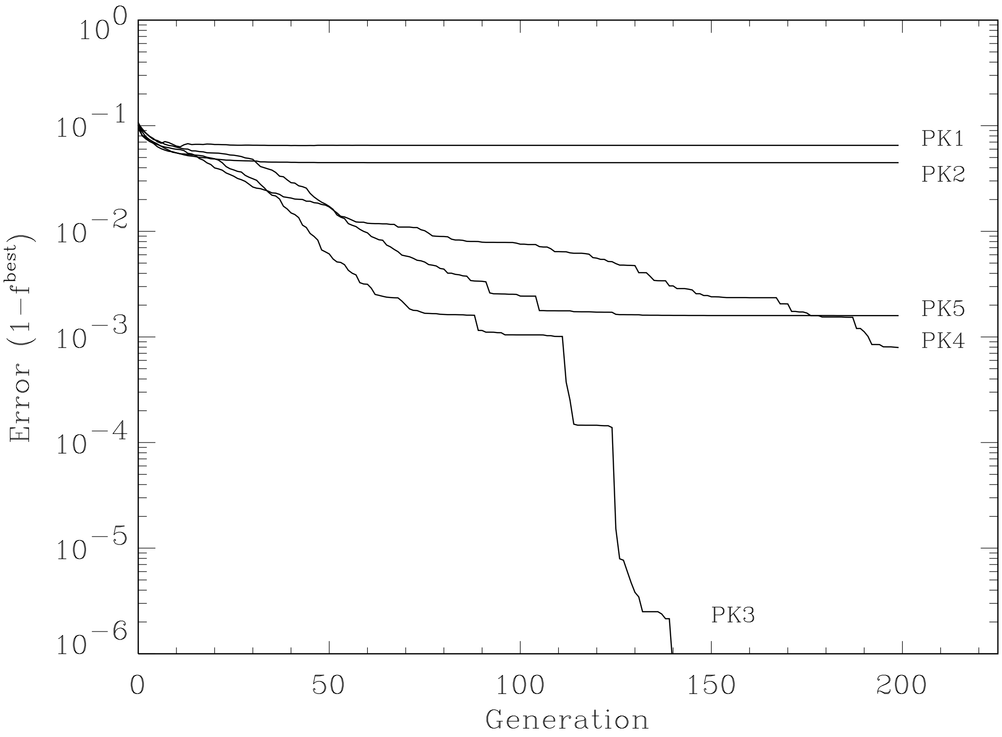

Figure 5.2: Performance of `PIKAIA` on the 2-D maximization problem of Fig. 5.1, for different choices of "ecological" strategies and mutation modes, as listed in Table II. The error is defined in eq. (5.4), and is computed for the best individual of a given generation. Each curve is actually a 100-run average, and so is fairly representative of the performance of the algorithm under the different parameter settings.

plans perform almost equally well, although one wouldn't immediately think so from a cursory glance at Fig. 5.2.

Further insight into the behavior of the method on the test problem can be gained by examining the distribution of the population as a whole in parameter space. This is carried out on Fig. 5.3, for the solution PK3 of Table II and Figure 5.2, corresponding to `PIKAIA`'s default settings. Each gray dot corresponds to one individual (a (x, y) point in 2-D parameter space), and the larger black dot is the fittest individual of a given generation. The concentric circles indicate the crests of the rings of secondary maxima (cf. Fig. 5.1). The initial, random population is distributed more or less uniformly in parameter space, but after only 5 generations (part [B]) a definite clustering on the secondary maxima is clearly apparent. By the tenth generation (part [C]) nearly all individuals are concentrated either on the central peak or on the first local extremum ring. At

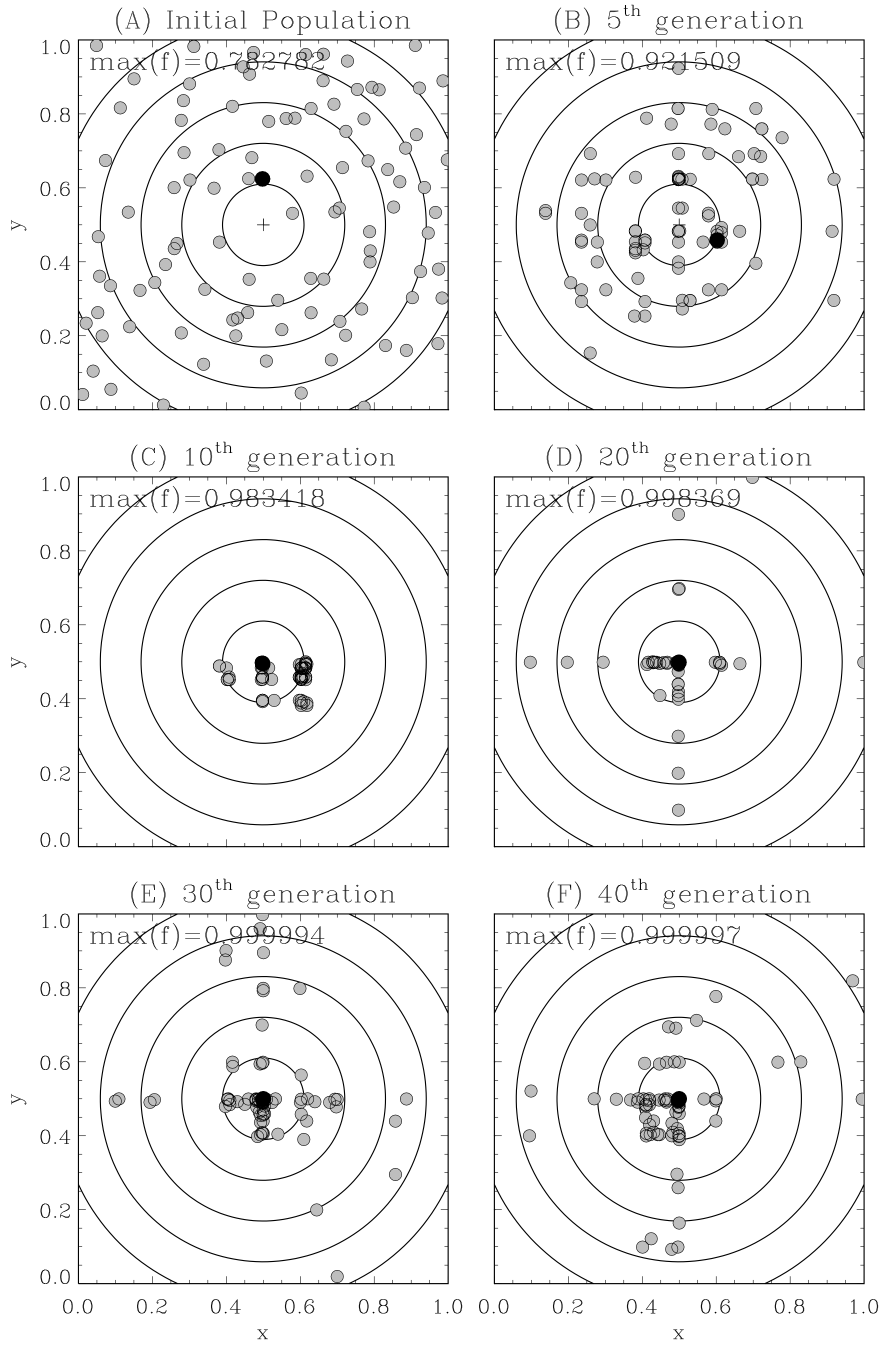

Figure 5.3: Evolution of the population during the PK3 evolutionary run (cf. Table II and Fig. 5.2). Each dot corresponds to an individual, and the black dot to the fittest individual of the given generation. The absolute maximum is located at $(x^*, y^*) = (0.5, 0.5)$, and the concentric rings indicate the crest of the secondary extrema (see Fig. 5.1). The mutation rate begins increasing at the fifteenth generation, and is responsible for the development of the crosshair pattern so particularly obvious on part (D).

that point the mutation rate begins to increase; this produces the cross-hair pattern seen on part (D). That increased mutation should produce this pattern is related to the rather direct relationship between the parameters encoded in the genotype, and their phenotypic representation; starting from a population clustered on the central peak, mutations affecting a gene decoding into the first significant digits of the x (y) parameter will produce a large horizontal (vertical) jump in solution space. As the mutation rate is further increased (parts [E] and [F]), deviations from the basic crosshair pattern become more common, as the high mutation rate makes it more likely for two mutations to affect simultaneously the first significant digits of both the x and y decoded solution parameters. The use of elitism is essential here, in order to continue improving the best solution. Notice how the "best" solution keeps improving from part (C) through (F), despites the fact that the spatial distribution of the population as a whole exhibits a greater spread about the central peaks in later phase of the evolutionary run.

5.2 A linear least squares fit

Data modeling is perhaps the most common computational task in the experimental sciences. In general terms, one is given a set of "measurements" f_j carried out at discrete set of "points" t_j:

$$f_j \equiv f(t_j), \qquad j = 1, 2, ..., J \tag{5.5}$$

with (one would hope) some estimate of the measurement error (σ_j) for the f_j's. This could describe, for example, a time series of measurements. Consider now a "model" $\hat{f}$ representing the process being measured. Such a model will of course depend on the measurement variable t, but also on a discrete set of K model parameters a_k; the modeling task consists in choosing values for these modeling parameters so as to minimize the difference between the model's predictions and the actual data:

$$\text{minimize w.r.t. } (a_1, a_2, ..., a_K) : \hat{f}(t_j; a_k) - f_j, \qquad j = 1, 2, ..., J. \tag{5.6}$$

A common strategy consists in minimizing the χ^2 merit function,

$$\chi^2(a_k) = \sum_{j=1}^{J} \left(\frac{\hat{f}(t_j; a_k) - f_j}{\sigma_j} \right)^2, \tag{5.7}$$

with respect to the a_k's. Provided that errors are normally distributed and independent of one another, this effectively corresponds to a maximum likelihood

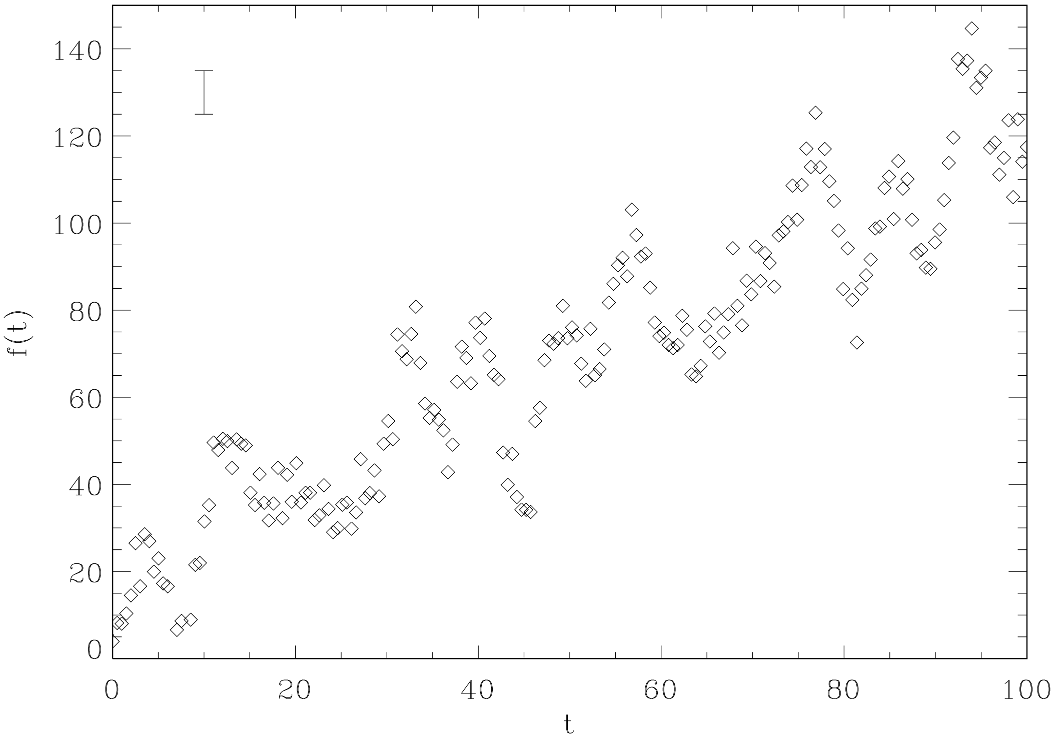

Figure 5.4: Synthetic dataset generated using eq. (5.11) below. This dataset includes normally distributed noise (zero mean) with $\sigma = 5$, corresponding to the error bar plotted in the upper left corner of the Figure. A linear trend is clearly apparent, but is that really all there is to it ?

criterion, in the sense that it selects the set of model parameters that maximizes the probability of the given dataset being realized, given those model parameters as true. Figure 5.4 shows a synthetic dataset comprising $J = 200$ measurements. How this dataset was generated is presently unimportant, other than knowing that with each point is associated the same measurement error $\sigma = \pm 5$, as indicated by the error bar plotted in the upper left corner of the Figure. Few data modelers could resists the temptation to begin by fitting a straight line through these data; the model under consideration would then take the simple form

$$\hat{f}(t; a, b) = at + b, \tag{5.8}$$

with the modeling task reducing to find the values (a, b) that minimize the RHS of eq. (5.7). Because the model depends linearly on its parameters, this is a *linear least squares* fitting problem (the dependency on the dependent variable t is actually irrelevant).

Consider now solving this fitting problem with `PIKAIA`. We have two adjustable parameters, i.e., `n`=2. Let us then associate a with `x(1)` and b with `x(2)`.

Unlike the model problem of the preceding section, it is now up to modeler to select appropriate bounds for the model parameters. A cursory glance at Fig. 5.4 would suggest that

$$0 \leq a \leq 10, \qquad 0 \leq b \leq 100$$

are probably safe choices. These ranges may appear rather generous, but to overly constrain the range of model parameters is in general a dangerous policy, even though it may greatly improve efficiency. Once the dust settles an expensive but correct solution remains infinitely better than a cheap one that is plain wrong.

The next step is to choose a measure of fitness. Many options are again available, while keeping in mind that PIKAIA is set up to *maximize* the user-supplied fitness function. In the present case perhaps the most straightforward approach is to make direct use of the χ^2, by assuming something like

$$\text{fitness} = \frac{1}{\chi^2}. \tag{5.9}$$

Recall that PIKAIA requires the fitness to be a positive definite quantity, so that writing fitness$= -\chi^2$ is definitely *not* a valid option. Because PIKAIA makes use of ranking to assign relative fitnesses in the roulette wheel algorithm, adopting instead $1/(\chi^2)^2$ or $\sqrt{1/\chi^2}$ or any other rank-preserving transformation would end up being nearly exactly equivalent to eq. (5.9) above (the "nearly" has to do with the implementation of variable mutation, and is discussed in its proper context further below). Computing the fitness associated with an individual (a, b) then involves

(1) appropriately rescaling a and b,
(2) computing the corresponding χ^2 against the data, and
(3) making use of eq. (5.9) to produce a fitness.

The user-supplied fitness function could then look as follows:

```
      function fit1a(n,x)
c===========================================================
c     Fitness function for linear least squares problem
c     (Sect. 5.2)
c===========================================================
      implicit none
      common/data/ f(200),t(200),sigma,ndata
      integer      n,i,ndata
      real         x(n),fit1a,f,t,sigma,a,b,sum
c---------- 1. rescale input variables:
```

```
      a=x(1)*10.
      b=x(2)*100.
c---------- 2. compute chi**2
      sum=0.
      do 1 i=1,ndata
         sum=sum+ ( (a*t(i)+b-f(i))/sigma )**2
    1 continue
c---------- 3. define fitness
      fit1a=1./sum
      return
      end
```

Note the presence of the common block `/data/`, which passes to the fitness function the `ndata` data points `f(1:ndata)` and `t(1:ndata)`, as well as the error `sigma` (assumed identical for all data points, and thus a scalar quantity here). The code listed here presupposes that in the program calling `PIKAIA`, the user has first called an initialization subroutine that read in these data and stored them appropriately in the common block. Note also that the fitness calculation does not require the computation of any derivatives of the χ^2 function with respect to modeling parameters. An important caveat is in order here. Performing the rescaling in the following manner:

```
x(1)=x(1)*10.
x(2)=x(2)*100.
```

besides being of dubious programming style, would have disastrous consequences; examination of `PIKAIA`'s listing will reveal that internally, the calls to `fit1` will often take a form equivalent to

```
fitns(ip)=fit1(n,oldph(1,ip))
```

at various places in the code. Because of the implicit association between `oldph` and `x` implied by the argument lists of the function, modifying `x` within `fit1` would consequently alter the content of the population matrix `oldph` within `PIKAIA`. Try it and see what happens!

Figure 5.5 shows the best fit found by `PIKAIA` after 100 generations, using default settings for other input quantities. Figure 5.6 details the evolution of some quantities of interest during the evolutionary run. The evolution of this specific solution (cf. Fig. 5.6(A)) is characterized by a gradual increase of the slope a, with concomitant decrease of the intercept b, but other solutions (i.e., starting from different initial random population, by using other `seed` values to initialize `PIKAIA`'s

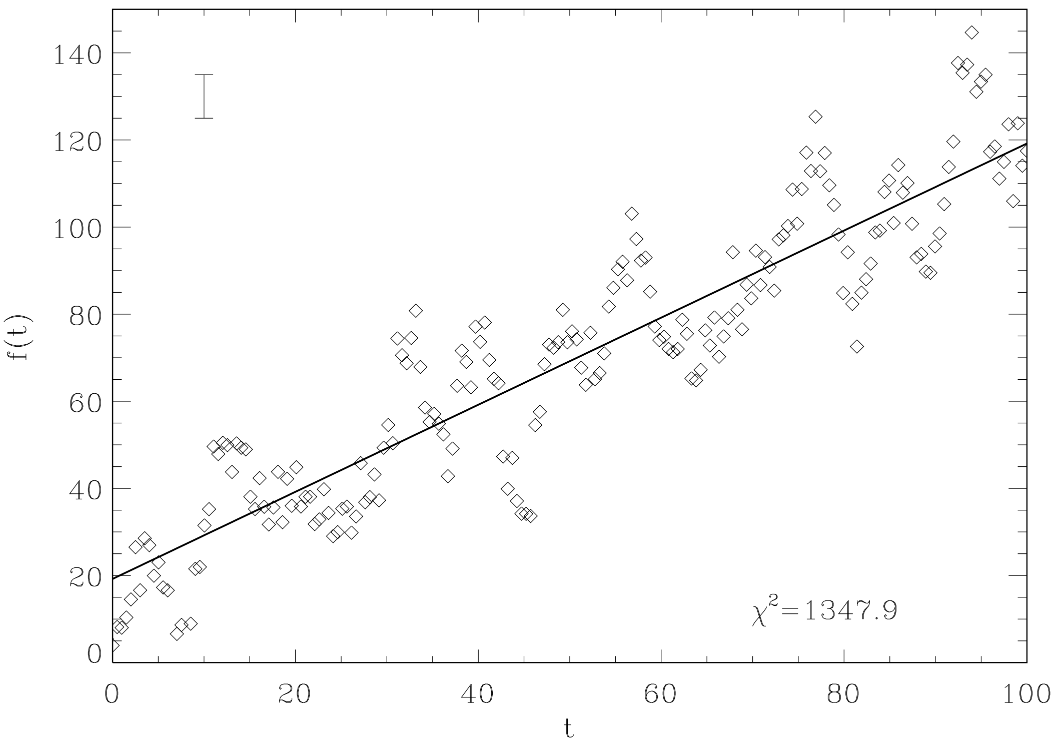

Figure 5.5: A genetic linear least squares fit to the dataset of Figure 5.4, obtained by running PIKAIA over 100 generations using default settings for the other input parameters. The final solution has $(a, b) = (0.99980, 19.216)$.

random number generator) sometimes show strongly non-monotonic evolution of the solution parameters, in particular at early evolutionary epochs.

It is instructive to examine the variations of the mutation rate in the course of the evolutionary run. One may note on Fig. 5.6(B) that during the first ten generations, the population rapidly converges, as evidenced by the χ^2 curves for the best and median individuals becoming nearly identical by the tenth generation. One may recall from §3.7.2 that this is the trigger for the onset of mutation rate increase, and this is clearly happening here as seen on Fig. 5.6(C). Note how, as the population starts to diverge again (generations 15—25) the mutation rate stabilizes, but starts increasing again once the population has again reconverged (generation 30). From that point on the population never really converges again, and the mutation rate stabilizes at a level close but inferior to the maximum allowed mutation rate `pmutmx`.

The initial drop in mutation rate in the first five generation would also suggest that for this particular application, the lower bound `rdifhi` in subroutine `adjmut` should be adjusted upwards. This is the single internal component of `PIKAIA` that

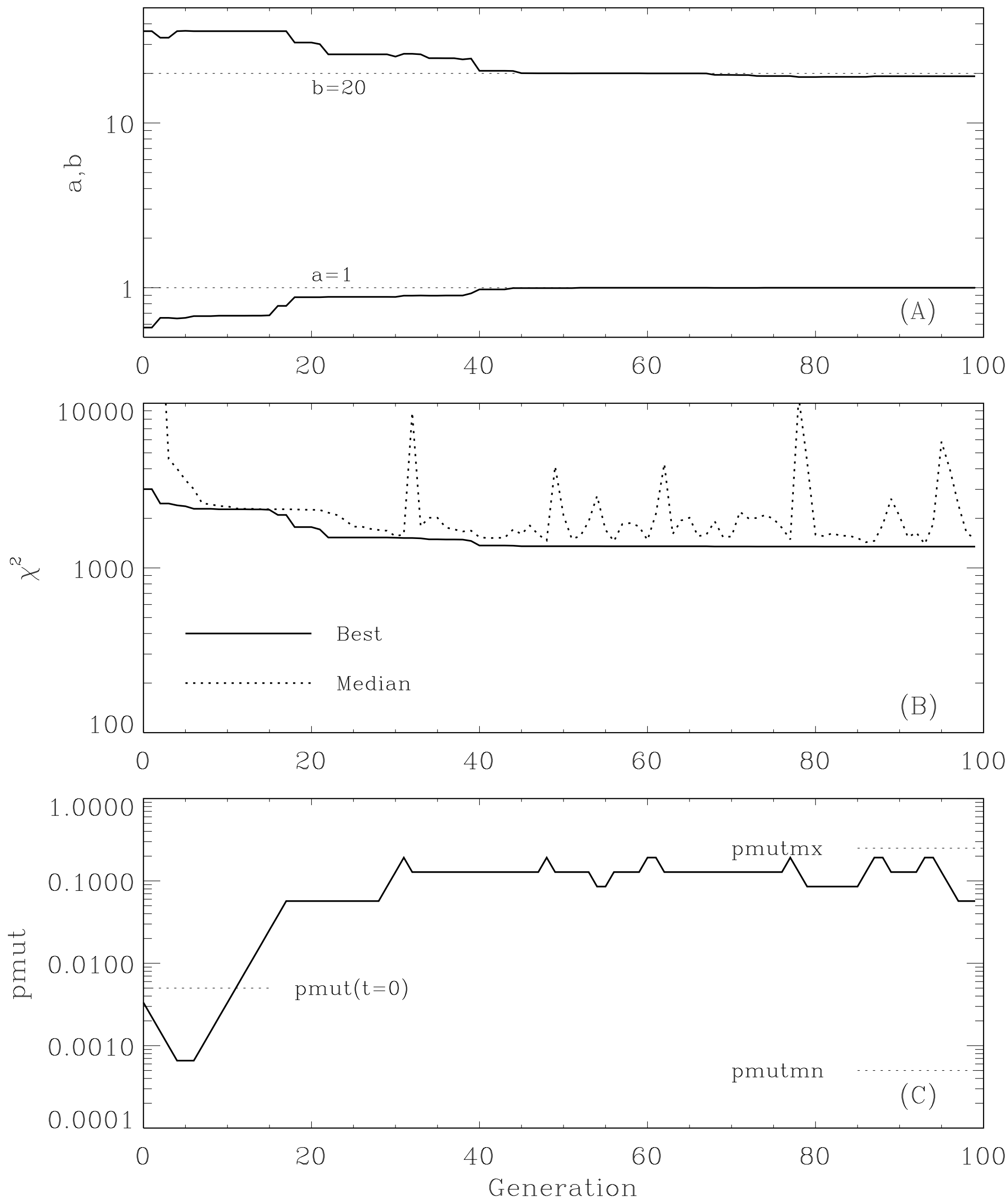

Figure 5.6: Evolution of the solution parameters throughout the evolutionary run (part [A]). Part (B) shows the evolution of the χ^2 for the best and median individuals in the population, and part (C) the evolution of the mutation rate (remember that `PIKAIA` operates under variable mutation rate by default). Note how the initial increase in mutation rate (part [C]) occurs in conjunction with convergence in the population, as evidenced by the χ^2 of the best and median becoming comparable between the fifth and tenth generations (part [B]).

is directly influenced by the specific numerical choice made for defining fitness, as in eq. (5.9) above (a way to bypass this problem is discussed in §6.3.1 below).

5.3 A non-linear least squares fit

At this point the overenthusiastic modeler may rush out of his/her office, call a press conference, and say something deeply profound (sic) about the age of the universe, or the mass of the gluon, or maybe even both. The cautious modeler, on the other hand, may notice that the χ^2 of the best fit is actually not all that good; for a model attempting to fit J measurements with a model depending on K parameters, a good fit should produce

$$\chi^2 \lesssim J - K, \tag{5.10}$$

which is not the case here, with $J - K = 198$ and $\chi^2 \simeq 1350$. The quantity $J - K$ corresponds to the number of degrees of freedom available to the fit, and embodies common sense notions such as the fact that a linear fit through two data points better return a χ^2 equal to zero. In fact, equation (5.10) is really only a zeroth-order estimate of goodness of fit, but it amply suffices for the present discussion. The excessive χ^2 value of the best fit may be due to a number of causes. Perhaps the error bars on the data have been underestimated by the experimenter ? Perhaps the errors are not normally distributed, and/or exhibit systematic trends and correlations ? Then again, perhaps the model used to fit the data is simply inadequate ? The first thing to do is to perform a few additional evolutionary runs, to ascertain the robustness of the best fit parameters. In the present case one systematically obtains $\chi^2 \geq 10^3$, while the best fit parameters ($a \simeq 1$, $b \simeq 20$) remain identical to 4 digits in most cases. This strongly suggests that the culprit is not the stochastic component of the genetic algorithm solution procedure, but perhaps instead an inadequate model.

Returning to Figure 5.4, it does not take much imagination to detect hints of periodic behavior in the data. This suspicion is spectacularly confirmed by subtracting from the data the best fit of the preceding section, and examining the power spectrum of the data residuals. So, instead of the simple linear model described by eq. (5.8), let us consider instead a more complicated model of the form

$$\hat{f}(t; a, b, \mathbf{A}, \mathbf{P}, \mathbf{\Phi}) = at + b + \sum_{m=1}^{M} A_m \sin\left[2\pi\left(\frac{t}{P_m} + \Phi_m\right)\right], \tag{5.11}$$

describing a multiperiodic signal superimposed on a linear trend. The model now depends on the a and b of the linear component, plus M amplitude/period/phase triplets $[A_m, P_m, \Phi_m]$, for a grand total of $3M + 2$ parameters. Adopting again the χ^2 merit function as a measure of goodness of fit, equations (5.7) and (5.11) now define a *non-linear least squares* fitting problem, since the model now depends nonlinearly on the P_m's[3].

The solution procedure using PIKAIA parallels to a remarkable degree the corresponding procedure for the nominally much easier linear least squares problem of the preceding section. For a given choice of M, the problem involves n$= 3M+2$ parameters. The first step is to choose the parameter ordering for the encoding process. Although there can be subtleties involved here (some to be discussed further below), for now let us simply adopt the following convention:

$$\mathbf{x} = \{a, b, A_1, P_1, \Phi_1, A_2, P_2, \Phi_2, ..., A_M, P_M, \Phi_M\} \tag{5.12}$$

The second step is to establish adequate scalings. The parameters a and b can be scaled as before. The remaining parameters can be scaled as

$$0 \leq A_m \leq 100, \qquad (2\Delta t) \leq P_m \leq 50, \qquad 0 \leq \Phi_m \leq 1, \qquad m = 1, 2, ..., M$$

Note that the lower bound on the P_m's stems from the fact that a set of equally spaced measurements, as on Figure 5.4, does not contain any useful information at frequencies above the so-called Nyquist frequency $f_N = 1/(2\Delta t)$, where Δt is the sampling interval in t.

The third step is to construct a fitness function. It is at this point that PIKAIA's versatility becomes most apparent. While each individual is now "defined" by $3M + 2$ parameters, all the fitness function needs to do is compute the χ^2 associated with the set of model parameters, which, conceptually, is no less straightforward for a nonlinear model such as eq. (5.11) than it was for the linear model defined by eq. (5.8). The fitness function for the nonlinear least squares problem could look as follow:

```
      function fit1b(n,x)
c===============================================================
c     Fitness function for non-linear least squares problem
c     (Sect. 5.3)
c===============================================================
      implicit none
```

[3] The apparently nonlinear dependence on the φ_k's can be reduced to a linear dependence by expressing each mode on the RHS of eq. (5.11) as a sum of sine and cosine.

```
      common/data/ f(200),t(200),sigma,ndata,m
      integer      n,m,mmax,ndata,i,j
      real         x(n),fit1b,f,t,sigma,amp,per,a,b,
     +             sum,sum2,nyp,pi
      parameter    (mmax=10,pi=3.1516926536)
      dimension    amp(mmax),per(mmax)
c---------- 1. rescale input variables:
      a=x(1)*10.
      b=x(2)*100.
      nyp=2.*(t(2)-t(1))
      do 10 j=1,m
         amp(j)=x(3*j)*100.
         per(j)=x(3*j+1)*(50.-nyp)+nyp
   10 continue
c---------- 2. compute chi**2
      sum=0.
      do 1 i=1,ndata
         sum2=0.
         do 2 j=1,m
            sum2=sum2+amp(j)*sin(2.*pi*(t(i)/per(j)+x(3*j+2)))
    2    continue
         sum=sum+ ( (a*t(i)+b+sum2-f(i))/sigma )**2
    1 continue
c---------- 3. define fitness
      fit1b=1./sum
      return
      end
```

Since the argument list of `fit2` must comply to what `PIKAIA` is expecting, the number (`m`) of Fourier modes used for the fit is actually passed through the common block `/data/`. It also turns out to be convenient to define two local arrays to perform the scaling of the amplitudes and periods, which in turn requires their maximum size (`mmax`) of these arrays to be hardwired in a `parameter` statement. These mild complications notwithstanding, compare this to the fitness function listed in the preceding section; the modifications are quite trivial. And once again derivatives of the merit function with respect to model parameters are simply not required.

Figures 5.7 and 5.8 show results of three solutions obtained using `PIKAIA`'s full default settings. The first solution attempts to fit a single Fourier mode ($M = 1$ in eq. (5.11)), the second solution makes use of three Fourier modes ($M = 3$) and the third of five ($M = 5$). At least in terms of the χ^2, the $M = 1$ solution is already a definite improvement over the linear fit of Fig. 5.5, although one still has $\chi^2 > J - K$. The $M = 3$ and $M = 5$ solutions, on the other hand, yield χ^2 that exceeds $J - K$ by less than a factor of two for $M = 3$, and by $\sim 10\%$ for $M = 5$. There is a trend here, yet the temptation to further increase M should be resisted. For one thing, letting M increase indefinitely for the sake of a lower and lower χ^2 could rapidly start to resemble the usually absurd notion of trying to fit J data points with a polynomial of order $J - 1$. It would also be premature to conclude, on the basis of the final χ^2 of the three solutions shown on Figs. 5.7 and 5.8, that there exists at least five Fourier components in the dataset. In this simple sequence of three solutions a clear pattern emerges in the Fourier modes selected by the algorithm, as shown in the following Table III:

Table III

Non-linear least squares solutions: parameters for best individuals

M	a	b	m	A_m	P_m	Φ_m	χ^2
1	0.9999	19.21	1	10.30	9.044	0.8346	923.5
3	0.9961	19.30	1	9.998	20.03	0.5124	285.8
			2	7.346	8.927	0.7008	
			3	6.770	7.373	0.7011	
5	0.9998	19.37	1	9.997	20.01	0.4999	207.1
			2	9.828	9.020	0.8115	
			3	6.945	7.472	0.8001	
			4	20.01	44.60	0.6462	
			5	19.76	45.10	0.1643	
original signal:							
3	1.000	20.00	1	12.00	20.00	0.5000	209.4
			2	10.00	9.000	0.8000	
			3	8.000	7.500	0.8000	

The mode identified in the $M = 1$ solution is also present in both $M = 3$ and $M = 5$ solutions, which is certainly encouraging. Likewise, the two additional modes found in the $M = 3$ solution also show up in the $M = 5$ solution. Great. Examination of the two remaining modes ($m = 4, 5$) of the $M = 5$ solution should sound a warning bell: nearly equal amplitudes and periods, $\Phi_4 - \Phi_5 \simeq 0.5$, corresponding to a phase difference of almost exactly π rad (cf. eq. (5.11)). The fourth and fifth Fourier component cancel one another out, and do not contribute significantly to the net signal. This degeneracy is a strong hint that using $M = 5$ is overkill. Related behaviors are (1) modes for which $A_m \rightarrow 0$, and (2) modes that become fragmented, i.e., identical periods and phases so that the two amplitudes simply add up arithmetically to form what is effectively a single Fourier component. The fact remains that the $M = 5$ solution has produced a better final fit than $M = 3$; this can be traced to the added flexibility provided by the availability of extra Fourier modes, which can be of great help in moving away from secondary extrema; as a rule of thumb here, if one is specifically looking for M modes (say), it would be a good idea to try to have `PIKAIA` search for $\sim 3M/2$.

Figure 5.8 also illustrates an important, general feature of GA-based optimizers: notice how, in the first 100 generations, the convergence rate of the solutions deteriorates with increasing M (and thus `n`), even though the final accuracy in this case increases with increasing M. This of course reflects the fact that the size of the parameter space to be explored increases geometrically with `n` while the population size is held fixed, but it remains true in general that the performance of GA-based optimizers can sometimes deteriorate dramatically as the number of adjustable parameters becomes large, although the magnitude of this deterioration tends to be very problem-specific. A final comment related to Figure 5.8; while the χ^2 of the $M = 1$ and $M = 5$ solutions do not decrease much in the second half of the evolutionary run, the χ^2 of the $M = 3$ solution shows a significant drop around generation 480. This is due to the appearance and subsequent spread of a favorable mutation in the population. Such an event opens up new roads in parameter space, the full exploration of which being likely to require some tens of generations. Faced with a convergence curve looking like the dotted line on Fig. 5.8, a careful genetic modeler would be well advised to extend the run by a hundred more generations.

It is now time to confess that the synthetic dataset of Fig. 5.4 was produced using eq. (5.11), with $M = 3$ and using the parameter values listed at the bottom of Table III. It is encouraging to note that the $M = 5$ solution actually returns a χ^2 slightly inferior to that produced by the original signal. This may appear odd, but is simply related to the specific noise realization used in producing the synthetic dataset[4]. Examination of Table III also shows that all non-degenerate and/or non-canceling Fourier modes identified by `PIKAIA` map rather closely onto modes existing in the original signal. An interesting exception is the amplitude

[4] Actually, if a $M > 3$ solution were to consistently yield a χ^2 significantly inferior to that associated with the original signal, one would have to seriously wonder about the quality of the random number generator used to produced the synthetic dataset.

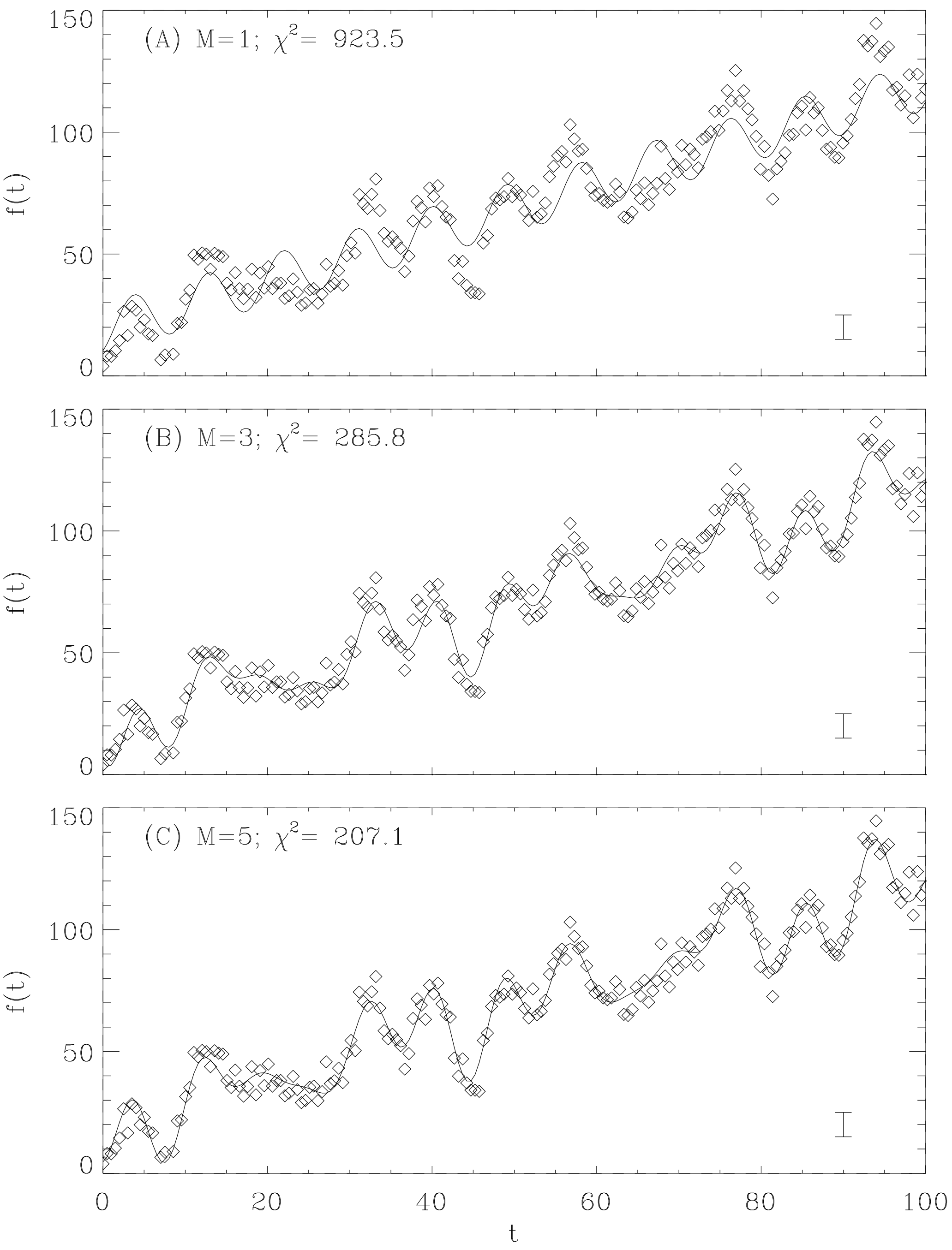

Figure 5.7: Non-linear least squares solutions obtained with `PIKAIA` using an increasing number (M) of Fourier modes (cf. eq. (5.11)) to fit the dataset of Figure 5.4. All solutions were computed using `PIKAIA`'s default settings, and so evolved over 500 generations.

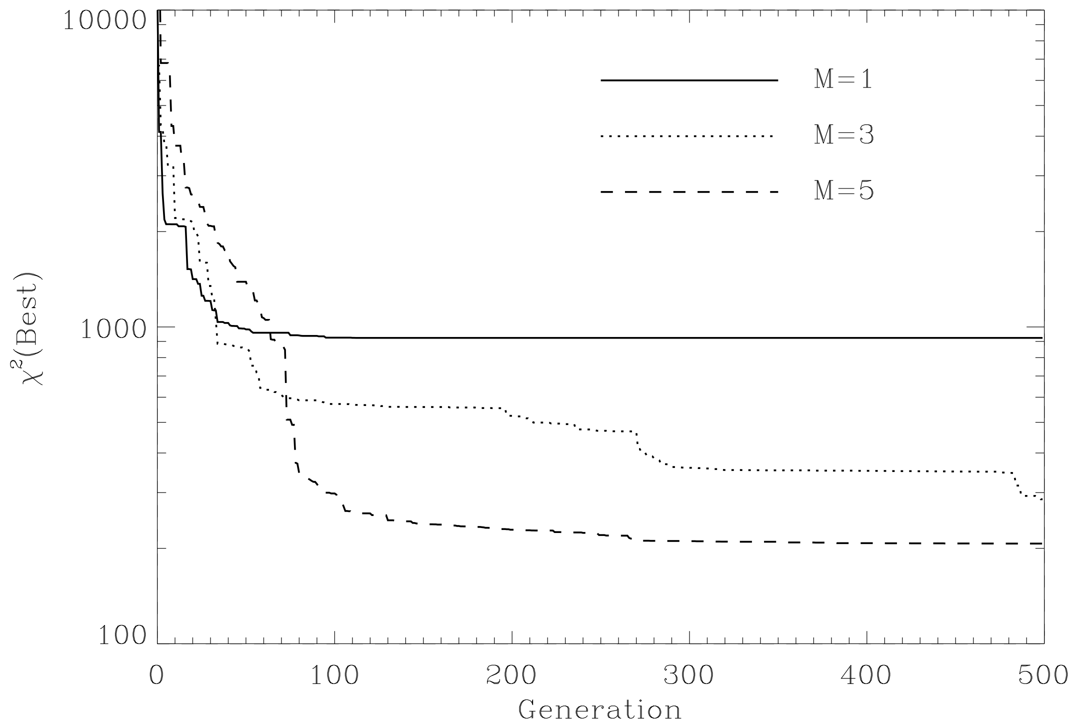

Figure 5.8: Evolution of the χ^2 associated with the best individual of a given generation, for the three genetic solutions depicted on Fig. 5.7. Note how, in the early stages of the evolution, the convergence is fastest for the solutions involving the least number of parameters.

of the $P_1 = 20$ mode; the original amplitude is $A_1^* = 12$ (superscripts "$*$" are hereafter used to denote parameter values of the original signal), yet both the $M = 3$ and $M = 5$ solutions ended up with $A_1 = 9.99...$.

We are encountering here a problem related to the rather subtle interplay that exists between the encoding protocol (§3.5) and the action of the simple one-point mutation operator (§3.7) incorporated in `PIKAIA`. Detailed examination of either the $M = 3$ or $M = 5$ solution reveals that early in the evolutionary run, a Fourier mode with $P_1 \sim P_1^*$ and $\Phi_1 \sim \Phi_1^*$ but $A_1 < 10$ shows up. Any individual encoding these three parameters early in the run ends up significantly fitter than its competitors, with the consequence that most individuals of subsequent generations incorporate those parameter values in their genotypes. As evolution proceeds and other Fourier modes become fixed in the population, natural selection favors an increase towards $A_1 \to A_1^* = 12$. Such incremental increases occur primarily (if not exclusively, in later evolutionary phases) through the action of the mutation operator. Consider the following portion of an individual's chromosome (with `nd`=5):

$$\texttt{......09999......} \qquad (5.13a)$$

corresponding to the chromosome segment where the amplitude $A_1 = 9.999$ is encoded. The "target" chromosome segment for A_1^* would look like

$$\texttt{......12000......} \qquad (5.13b)$$

for $A_1 = 12.000$. Clearly, transforming the first chromosome segment into the second requires a very coordinated piece of work on the part of the mutation operator. In this case only the first two genes are really important; the first gene must mutate from `0` to `1` *and* the second from `9` to `0` or `1`. Any of these mutations occurring in isolation, or mutating to other digits, will produce an inferior individual. One may easily verify that the probability of the required mutations simultaneously occurring in the same individual is

$$p = \frac{\texttt{pmut}}{10} \times \frac{\texttt{pmut}}{5},$$

even with `pmut` floored at `pmutmx`=0.25, this remains a very small number, i.e., $p \simeq 10^{-3}$. Furthermore, this event must *not* be accompanied by any mutation at genes mapping onto the first few significant digits of the other parameters. The probability of this *not* occurring would be of order $(1 - \texttt{pmut})^{n-1}$ if we consider only the first significant digit of each parameters, so that the net probability of going from (5.13a) to (5.13b) is

$$p \simeq \frac{\texttt{pmut}^2}{50} \times (1 - \texttt{pmut})^{n-1}, \qquad (5.14)$$

with `pmutmx`=0.25 and $n = 11$ (for $M = 3$), this new net probability drops to $p \simeq 7 \times 10^{-5}$, down to 4×10^{-6} if two significant digits are considered essential. Notice that increasing the mutation rate does not help! The above probability vanishes in both limits `pmut`$\to 0$ and `pmut`$\to 1$. The bottom line is that going from (5.13a) to (5.13b) above is very unlikely indeed. The encoding scheme produces a "wall" (often referred to in the GA literature as a "Hamming wall") that the standard one-point mutation operator is highly unlikely to cross. This could be interpreted as a rather severe limitation of the decimal encoding scheme adopted herein. Indeed, the use of a binary scheme based on the so-called Gray coding (e.g. Press *et al.* 1992, §20.2) does not suffer from this shortcoming. Should we then abandon decimal encoding altogether? Not necessarily; another possibility is to specifically modify the standard one-point mutation operator so that it can cross Hamming Walls. This may involve (for example) incrementing by ± 1 the gene located left of another gene mutating from a `9` to a `0`, or from a `0` to a `9`. This "carry-over-the-one" should bring back nostalgic memories of grade school days... and is left as an exercise! It does involve additional operations and tests within the mutation operator, but experience shows that in real-life applications the amount of CPU time spent running the mutation operator is utterly insignificant as compared to fitness evaluation, the latter being typically where most of the CPU time ends up being spent.

A final comment concerns the convention adopted above to order the solution parameters prior to encoding. We adopted above the ordering

$$\mathbf{x} = \{a, b, A_1, P_1, \Phi_1, A_2, P_2, \Phi_2, ..., A_M, P_M, \Phi_M\}, \qquad (5.15a)$$

but one may think that the following would have presumably worked just as well:

$$\mathbf{x} = \{a, b, A_1, A_2, ..., A_M, P_1, P_2, ..., P_M, \Phi_1, \Phi_2, ..., \Phi_M\}. \qquad (5.15b)$$

It turns out that the convention (5.15b) yields markedly slower convergence that (5.15a). This time we are dealing with a (crucial) subtlety associated with the operation of the crossover operator (see §3.6). Consider again the appearance, in early evolutionary phase of an individual having locked onto the $P = 20$ Fourier mode with reasonable accuracy, e.g.,

$$(A, P, \Phi) = (9.987, 20.666, 0.48888);$$

ignoring parameter rescaling for the sake of simplicity, under the convention (5.15a) the corresponding amplitude/period/phase triplet would end up encoded on the chromosome as

..........`099872066648888`............................. (5.16*a*)

say, while under the convention (5.15b) the chromosome would look something like

..........`09987`..........`20666`..........`48888`.......... (5.16*b*)

Now, admittedly an individual incorporating those parameters in its genotype will end up fitter than average, independently of the parameter ordering. However, upon breeding, an individual encoding the parameters according to convention (5.15a) is far more likely to pass on the corresponding amplitude/period/phase triplet *intact* to one of its offspring. This is of course directly related to the cut-swap-splice mode of operation of the crossover operator. For $M = 5$ and `nd`$= 5$

one may verify that the corresponding probabilities for the crossover operator *not* to disturb the above sets of advantageous genes are $p = 0.82$ for convention (5.15a), but only $p = 0.35$ under convention (5.15b). This possibility of passing on and combining through successive generations discrete chunk of "advantageous" chromosomal segments, with the parallel elimination of "detrimental" segments, is largely what confers to genetic algorithms their power for efficient exploration of parameter space. This behavior is embodied in the *schema theorem* (see e.g. Holland 1975, 1992; Goldberg 1989, chap. 2, and references therein), which forms the basis of most theoretical analysis of genetic algorithm behavior and performance. Such considerations have no counterpart in most conventional optimization methods. To dwell further into these matters would take us to far outside the purview of a user's guide, but the reading list provided in §6.4 should be consulted for directions to comprehensive discussions of the crucial concept of schemata processing by genetic algorithms.

5.4 Generalized least squares fitting by distance regression

Consider the dataset shown in Fig. 5.9, and the associated modeling task of finding the ellipse that yields the "best" representation of these data. There are five fitting parameters involved: the geometric center (x_0, y_0) of the ellipse, its semi-major and -minor axes a and b, and the inclination θ_0 of the major axis with respect to the x-axis (say). The sought after ellipse is described in polar coordinates by the curve

$$r^2(\theta) = \frac{a^2 b^2}{a^2 \cos^2(\theta - \theta_0) + b^2 \sin^2(\theta - \theta_0)}. \tag{5.17}$$

The J data points (x_j, y_j) are then compared to

$$X_j = x_0 + r(\theta_j)\cos(\theta_j), \tag{5.18a}$$

$$Y_j = y_0 + r(\theta_j)\sin(\theta_j), \tag{5.18b}$$

where $\theta_j = \mathrm{atan}(y_j/x_j) \in [0, 2\pi]$ is the angle subtented by the segment joining (x_0, y_0) to (x_j, y_j). The procedure known as *distance regression* (hereafter DR) functions as follow: given the ellipse center (x_0, y_0), compute the distance d_j between that center and the j^{th} data point; compute then the predicted radius $r(\theta_i)$ associated with the ellipse having parameters $(x_0, y_0, a, b, \theta_0)$. The DR merit function can then be defined as

$$F(x_0, y_0, a, b, \theta_0) = \sum_{j=1}^{J} (r(\theta_j) - d_j)^2 \tag{5.19}$$

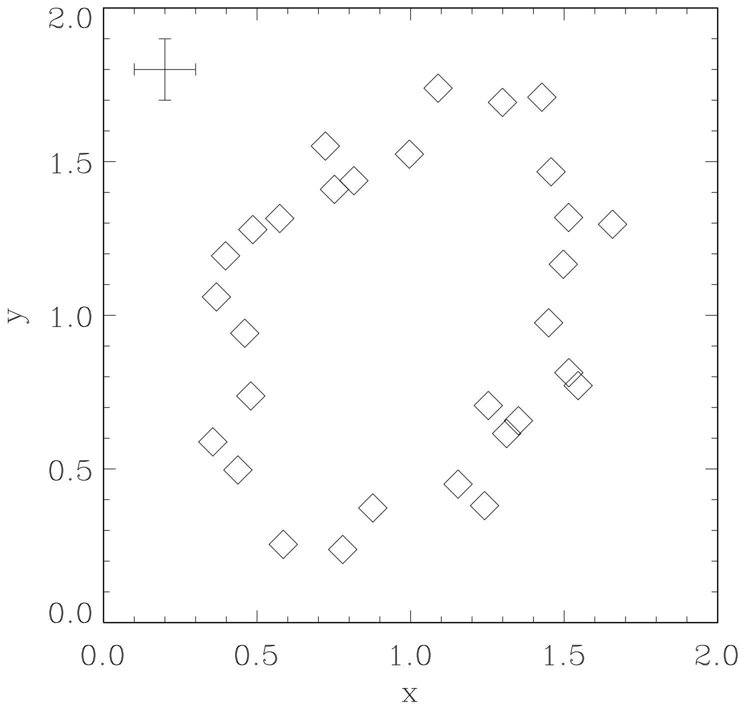

Figure 5.9: Synthetic data for the generalized nonlinear least squares example of §5.4. Measurement errors of relatively large amplitude exist for both x and y coordinates, as indicated by the error cross in the upper left. Fitting an ellipse to these data is a rather difficult and involved modeling problem by any standard.

(The above expression explicitly assumes that the measurements errors on x and y are identical for all measurements; otherwise appropriate weights must be introduced in eq. (5.19); see, e.g., Boggs *et al.* 1987)[5]. The optimization task is then

---

[5] The related, more general procedure known as *orthogonal distance regression* (ODR) described in Boggs *et al.* (1987) minimizes the sum of the distances between data and parametric curve; however the distance is computed not in relation to a common "center of gravity", but along a line segment perpendicular to the parametric curve being used to model the data. The ODR procedure then minimizes the sum of "nearest approach" distances between the data and the parametric curve. This is of course identical to the scheme described here in the case of circle fitting, and effectively equivalent for fitting ellipses of low eccentricities. The software package ODRPACK (Boggs *et al.* 1989) is a good example of the current state of the art for ODR. An alternate solution procedure can be also devised based on Singular Value Decomposition; see Golub & Van Loan 1989, §12.3.

to find the parameter set

$$\mathbf{x}^* = (x_0^*, y_0^*, a^*, b^*, \theta_0^*)$$

that minimizes eq. (5.19). Clearly, the relationship between the merit function and the model parameters is wickedly non-linear. The mathematics involved in applying conventional techniques to this problem are rather gruesome (try to work out the partial derivatives $\partial F/\partial a$ for yourself and see how you like it), and the resulting software packages are of substantial size and complexity. This problem is *far* from trivial.

Consider now a solution computed using PIKAIA. The procedure is basically as straightforward as for the much simpler least squares problems of §§5.2 and 5.3. We are dealing with a 5-parameter search space, so n=5. From Fig. 5.9 one may adopt the scalings:

$$0 \leq x_0, y_0, a, b \leq 2, \qquad 0 \leq \theta_0 \leq \pi.$$

Using the DR merit function (eq. (5.19)) as a measure of fitness, the required fitness function could look something like this:

```
      function fit3(n,x)
c=============================================================
c     Fitness function for circle fitting problem using
c     a robust estimator based on the Hough transform
c     (Sect. 5.5)
c=============================================================
      implicit none
      common/data/ xdat(200),ydat(200),sigma,ndata
      integer      n,ndata,i
      real         x(n),fit3,xdat,ydat,sigma,x0,y0,distdat,
     +             r1,r2,sum
c---------- 1. rescale input variables:
c           0 <= x(1,2) <= 3
      x0    = x(1)*3.
      y0    = x(2)*3.
c---------- 2. compute merit function
      r1=1.-sigma/2.
      r2=1.+sigma/2.
      sum=0.
      do 1 i=1,ndata
```

```
c     (a) compute distance d_j from center to data point
         distdat=sqrt((x0-xdat(i))**2+(y0-ydat(i))**2)
c     (b) increment Hough merit function
         if(distdat.ge.r1.and.distdat.le.r2) sum=sum+1.
    1 continue
c---------- 3. equate fitness to Hough merit function
      fit3=sum
      return
      end
```

The code is direct transliteration of the DR procedure outlined above. The user-supplied function `fatan` must return the arctangent $\in [0, 2\pi]$. The inverse of the DR merit function is used as a measure of fitness, because `PIKAIA` is set up to *maximize* the user-supplied fitness function.

The solid line in Figure 5.10 shows a solution obtained by letting a population of 50 individuals evolve over 200 generations, with `PIKAIA` operating under the Select-random-delete-worst variation of the Steady-state-delete-worst reproduction plan (by setting `fdif`=0.0 and `irep`=3). Defaults settings are used for all other parameters. The dotted is the ellipse from which the synthetic dataset of Fig. 5.9 was actually generated; this original ellipse had

$$(x_0, y_0, a, b, \theta_0) = (1.0, 1.0, 0.75, 0.50, \pi/3),$$

while the genetic solution returns:

$$(x_0, y_0, a, b, \theta_0) = (0.97076, 0.97996, 0.77182, 0.50538, 0.32119\pi).$$

Of course different noise realizations would lead to different "best fit" ellipses, and there is no reason to expect that in the presence of noise, the best fit should coincide exactly with the original ellipse. The point here is that despite the rather noisy data, which translates in a very multimodal merit function hypersurface in parameter space, `PIKAIA` has no difficulty recovering the original ellipse to impressive accuracy.

5.5 Data modeling using robust estimators

The distance regression procedure used to solve the ellipse fitting problem of the preceding section remains a form of generalized least squares minimization. Such statistical estimators perform well if errors are normally (or at least compactly and symmetrically) distributed about the "true" values, but can give spurious results

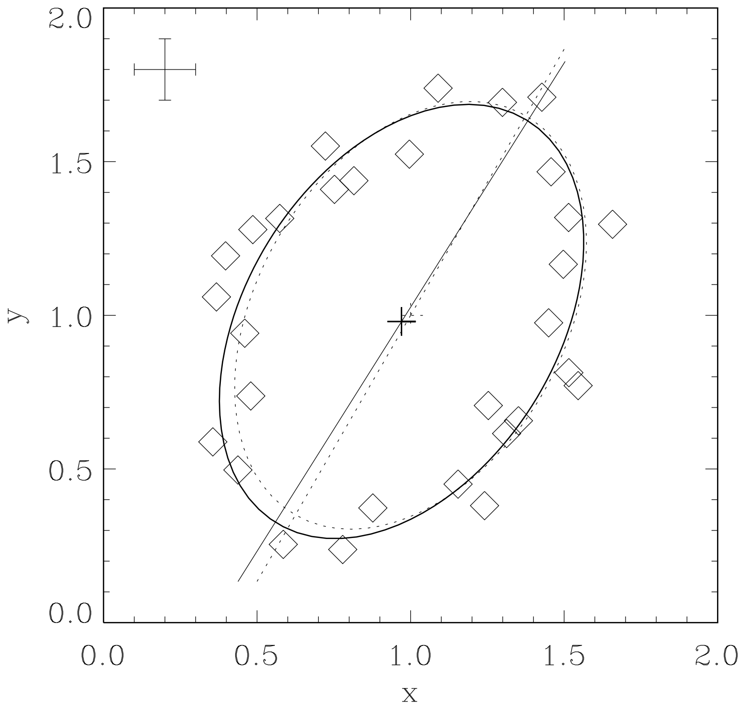

Figure 5.10: A 200-generation genetic solution to the distance regression problem defined by the dataset of Fig. 5.9, using `PIKAIA`'s default setting. Both the best ellipse and the corresponding major axis are indicated (solid lines). The dotted lines are the original ellipse (and corresponding semi-major axis) from which the synthetic dataset was generated. Note that for this noise realization, the genetic solution yields a smaller value of the DR merit function than the original ellipse.

if errors exhibit strong asymmetric trends and/or are polluted by outliers. Using a χ^2 (or other related least squares estimators, such as in §5.4) will give great weight, in computing the merit function, to data points that are *way* off in outer space. This is perhaps most easily seen by computing a linear least squares fit for 10 data point lying *exactly* on a straight line, but where one endpoint datum is artificially displaced by a large factor above above the original $y = mx + b$ relationship; the best fit will in general be a poor fit (see the discussion in Press *et al.* 1992, §15.7). This characteristic of least squares estimators has given rise to the all too common practice of deleting data points that are "obviously" way off, which is even more objectionable than "chi-by-eye". Clearly a more robust statistical estimator is required in such cases.

Consider a simplified version of the problem treated in the preceding section, namely fitting a circle of known radius to a dataset with errors in both coordinates. While presumably simpler than the ellipse fitting problem of the preceding section, this is *still* a hard problem (see, e.g., Cox & Jones 1989; Moura & Kitney 1991). Introduce now the following complication (see Figure 5.11): unlike the data of Fig. 5.9, there are now two components to the "error"; a low amplitude symmetrically distributed component, as well as a high amplitude, highly asymmetric component: someone took a bite out of the circle... Consider now the task of locating the true center of a "damaged" circle such as this, given the a priori information that the sought after circle has unit radius, and that we *may* be dealing with an incomplete circle (this apparently odd kind of task is actually commonly encountered in the field of computer vision and automated pattern recognition). The DR estimator of the preceding section can certainly be used. The problem reduces to two parameters, namely the (x_0, y_0) coordinate of the circle center. The dashed line on Figure 5.11 is the resulting best fit. That circle is displaced to the right with respect to the original circle, in view of the strong "pull" exerted by the deviant data points of the bite mark; these deviant points are outliers, and they receive overly large weights from DR (or ODR) estimators, exemplifying the need for a more robust estimator.

Consider the following, alternate procedure. With the target circle known to have a radius $R = 1$ and knowing the error estimate σ, one constructs an annulus of inner radius $R - \sigma$ and outer radius $R + \sigma$ centered on a trial center (x_0, y_0), counts the number n of data points lying within the annulus, and assigns that value to the estimator:

$$H(x_0, y_0) = \sum_{j=1}^{J} \begin{cases} 1, & R - \sigma \leq d_j \leq R + \sigma \\ 0, & \text{otherwise} \end{cases} \tag{5.20}$$

where, as before, $d_j = \sqrt{(x_j - x_0)^2 + (y_j - y_0)^2}$ is the distance between the j^{th} data point and the trial center (x_0, y_0). The function $H(x_0, y_0)$ is known as a *Hough transform* of the data, and is used extensively for pattern recognition in computer vision (see, e.g., Ballard & Brown 1982). It should be emphasized that the estimator based on the Hough transform differs in a crucial way from the ODR estimator: once a data point is out of the annulus, it simply does not matter how far out is stands; the merit function is simply *not* incremented. Contrast this to the ODR-type estimators, where extreme outliers can introduce strong biases, the more so the farther away they lie.

This important difference notwithstanding, are we not simply back again to the problem of §5.1, namely maximizing a function of two variables ? Depending on the technique used to solve the problem, the answer may well be no; the Hough transform assumes only integer values $\in [0, J]$, so that all local derivatives formally vanish throughout the domain. Furthermore the transform most often

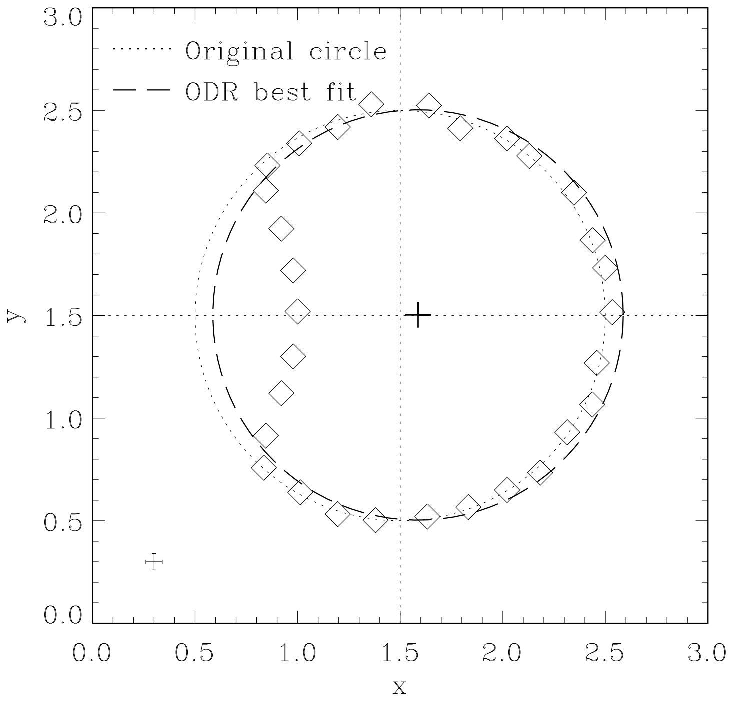

Figure 5.11: Synthetic dataset for a "damaged" circle. A substantial "bite" was first taken out of the left side of the original circle (dotted line). The data points were then generated by sampling the resulting geometric figure at constant angular increment, and adding a random error of relatively low amplitude (see error cross on lower left) to the x and y coordinates of the point. While this latter error component is symmetrically distributed, the original bite corresponds to errors with a strong systematic trend that in now way resembles a normal distribution. The dashed circle is a fit to these data with a circle of unit radius using the DR estimator of the preceding section, exactly equivalent to an ODR estimator in the case of circle fitting.

ends up being multimodal on a variety of spatial scales. This is certainly the case for the Hough transform for the dataset of Fig. 5.11, as shown on Figure 5.12. Optimization schemes relying on derivative information, or even local nearest-neighbour sampling, are clearly useless on this problem!

Once again using `PIKAIA` in conjunction with the Hough estimator (as opposed to a least squares-type estimator) is absolutely straightforward. One can directly adopt eq. (5.20) as a measure of fitness—and that is all there is to it. We are indeed back to the maximizing problem of §5.1. So let's hit it as before, `n=2` and `PIKAIA`'s default settings. Figure 5.13 shows the resulting best fit produced by

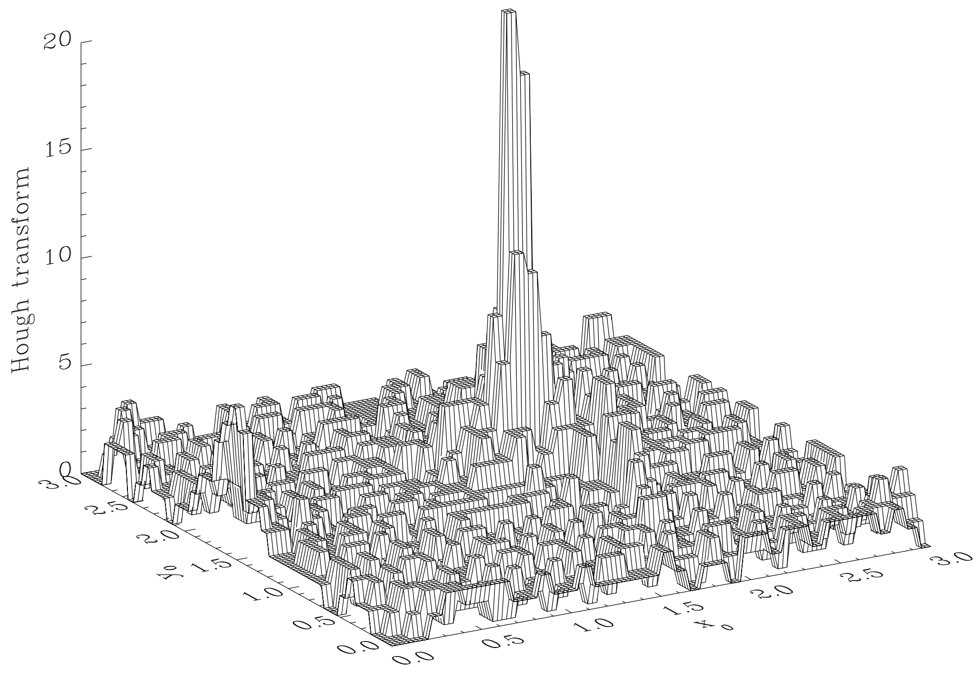

Figure 5.12: Hough transform for the dataset of Fig. 5.11. The Hough transform assumes integer values, here in the range $[0, 30]$.

`PIKAIA`. The original circle is recovered with much better accuracy, with the best fit yielding

$$(x_0^*, y_0^*) = (1.5068, 1.4953).$$

The point here is not so much that incorporating robust estimators in a genetic code produces an inherently better algorithm, in the sense of accuracy and/or efficiency; what is noteworthy here is the *ease* with which robust estimators can be incorporated in a suitably designed general purpose GA-based optimizer. In general, this is definitely *not* the case when robust estimators are used in conjunction with more conventional optimization methods relying on derivative information, even when such derivatives can be computed. We refer the interested reader to §15.7 of Press *et al.* (1992) for a brief but accessible introduction to these issues. Consider in contrast how robust estimators are accommodated within `PIKAIA`; the fitness function used to compute the solution shown on Fig. 5.13 looks like:

```
      function fit2(n,x)
c==========================================================
c     Fitness function for ellipse fitting problem (Sect. 5.4)
```

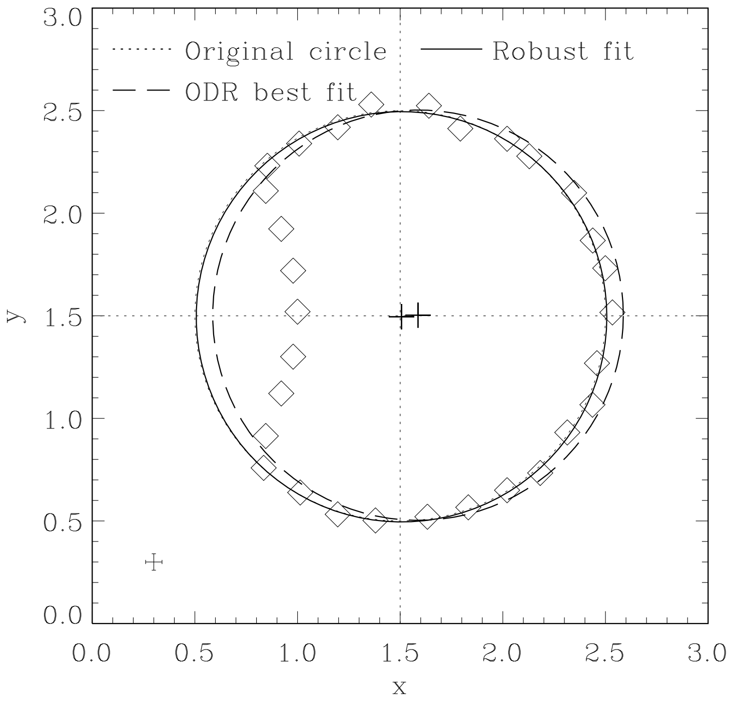

Figure 5.13: Robust fit to the dataset of Figure 5.11 using the Hough transform (solid line). The original circle is recovered with much better accuracy than when using the ODR estimator.

```
c
c     Ellipse parameters are:
c     x(1)=x_0, x(2)=y_0, x(3)=a, x(4)=b, x(5)=theta_0
c==============================================================
      implicit none
      common/data/ xdat(200),ydat(200),ndata
      integer      n,ndata,i
      real         x(n),fit2,xdat,ydat,fit2,fatan,x0,y0,a2,b2,
     +             theta0,distdat,angdat,rthj,sum,pi
      parameter    (pi=3.1415926536)
c---------- 1. rescale input variables:
c           0 <= x(1...4) <= 2, 0 <= x(5) <= pi
      x0    = x(1)*2.
      y0    = x(2)*2.
      a2    =(x(3)*2.)**2
      b2    =(x(4)*2.)**2
```

```
      theta0= x(5)*pi
c---------- 2. compute merit function
      sum=0.
      do 1 i=1,ndata
c     (a) compute distance d_j from center to data point
         distdat=sqrt((x0-xdat(i))**2+(y0-ydat(i))**2)
c     (b) compute angle theta_j of segment center---data point
         angdat=fatan((ydat(i)-y0),(xdat(i)-x0))-x(5)*pi
c     (c) compute radius r(\theta_j) of ellipse at that angle
         rthj=sqrt(a2*b2/(a2*sin(angdat)**2+b2*cos(angdat)**2))
c     (d) increment DR merit function
         sum=sum+(distdat-rthj)**2
    1 continue
c---------- 3. equate fitness to inverse of DR merit function
      fit2=1./sum
      return
      end
```

Rescaling and circle vs ellipse issues notwithstanding, this differs in only *two* source code lines from the source code for the DR estimator of the preceding section. Very few methods could be so easily modified (an important one that could is the Simplex method, already encountered in §4.7).

5.6 A warning concerning error estimates

WARNING: Users should resist the temptation to use the population distribution in parameter space, at the end of the evolutionary run, to calculate variances, and from those infer error bars; the final distribution of the population in parameter space represents an *extremely* biased sample, and will be found to depend significantly on the types of genetic operators and ecological strategies used throughout the evolutionary process. END OF WARNING.

Broadly speaking, there are two avenues open to obtain error estimates on the "best fit" parameters. The first involves the hybrid approach advocated in §4.7.1 above, in the sense of using the best genetic solution to initiate a conjugate-gradient type method (say), and make use of the covariance machinery that often accompanies such routines (e.g. routine `mrqmin` of Press *et al.* 1992) to extract error estimates. There are two potential shortcomings here; the first is that derivative information is usually required, which, as we saw, can rapidly become messy for strongly nonlinear models. The second is that the resulting error estimates are local, in the sense that they are essentially constructed using information relating to local curvature in parameter space about the optimal solution. Such estimates usually have nothing to say about the ever lurking possibility that the "best fit" is a local, rather than global, extremum. This type of error is of course notoriously difficult to pin down. Yet there is a good chance that if you were using a GA-based optimizer in the first place, it is because you are dealing with a complex, ill-behaved and most likely multimodal search space. Remember, no free lunch!

The second option, both safest and costliest, involves Monte Carlo simulation. In the case of the various modeling problems treated in this chapter, the mapping of χ^2 boundaries in n-dimensional parameter space is usually the procedure used to established *confidence limits* on model parameters. This is a vast topic, a detailed discussion of which is not appropriate here. We point the interested reader to §15.6 of Press *et al.* 1992, and to Bevington & Robinson (1992, chap. 11). Let us simply mention that the procedure involves producing a set of perturbed solutions $\mathbf{a}^i$ around the "best fit" $\mathbf{a}^*$

$$\mathbf{a}^i = \mathbf{a}^* + \Delta \mathbf{a}^i, \qquad i = 1, 2, ..., I,$$

and computing the χ^2 associated with each perturbed solution; for large enough I, it becomes possible to estimate the probability that the "true" solution parameters lie within a confidence region $\chi^2(\mathbf{a}^*) + \Delta\chi^2$ in n-dimensional parameter space. The point here is that if a genetic solution has already been obtained, then all the machinery required to compute the required

$$\chi^2(\mathbf{a}^* + \Delta \mathbf{a}^i), \qquad i = 1, 2, ..., I$$

is already in place: it is simply the user-supplied fitness function! While the construction of confidence limits may turn out to add a significant CPU-time requirement, in terms of coding or other (real-time) work on the part of the modeler, the associated coding overhead is much smaller than one may originally imagine.

5.7 Other applications

The ease with which `PIKAIA` moves across problem domains is due to the fact that, fundamentally, GA-based optimizers repeatedly solve the *forward problem* associated with the modeling task, as opposed to the original *inverse problem* posed by the data itself. Craig & Brown (1986) give an accessible introduction to some of the sneakiness lurking in inverse problems. The point is that forward problems (given a and b, compute a simulated dataset $y_j = ax_j + b$ and its associated χ^2) tend to be much easier and better posed than the corresponding inverse problem (given data, solve directly for the values of a and b that minimize χ^2). In that respect GA-based optimizers indeed resemble Monte Carlo methods. Not surprisingly genetic techniques have attracted the interest of workers involved in complex inverse modeling tasks. The power of GA-based methods has already been amply demonstrated in the field of geoseismic acoustic inversion (see, e.g., Wilson & Vasudevan, 1991; Sen & Stoffa, 1992; Sambridge & Drijkoningen, 1992), where their efficiency has been shown to exceed that of Monte Carlo methods by orders of magnitude. Preliminary applications to helioseismic inversion (Tomczyk *et al.* 1995) also look extremely promising. In particular, the ease with which constraints such as positivity and/or monotonicity can be effectively hardwired at the encoding level makes the usually formidable task of *constrained inversion* easily manageable for wide classes of constraints.

6. WHERE TO GO FROM HERE: HACKING PIKAIA

As mentioned already a few times in the preceding pages, `PIKAIA` is primarily a learning instrument. It has nonetheless been successfully used, as of this writing, to solve a number of real life problems encountered by the authors (and some friends and colleagues) in their respective research endeavours. Especially in the development phases, putting `PIKAIA` to work on a real problem may in some cases be facilitated by performing some modifications to the code itself. The aim of this chapter is to provide the user with some basic information that may be useful toward that end.

6.1 Overall coding structure and subroutine dependencies

Figure 6.1 illustrates schematically the subroutine dependencies within `PIKAIA`, as called from a typical driver code, such as for the linear and non-linear least squares fit problems discussed in the preceding chapter. Note that appropriate random number generator and ranking subroutines are provided with `PIKAIA`'s installation package, even though they are labeled "user-supplied" on Figure 6.1.

In constructing the code we have strived for maximal modularity, and but for a single exception have entirely avoided communication across subroutine through `COMMON` blocks[6]. While this has produced a source code lengthier than it could have been, the motivation for doing so is to facilitate the task facing a user wishing to incorporate additional subroutines *within* the `PIKAIA` subroutine itself.

6.2 Tailoring or expanding the output

`PIKAIA` is currently set up to produce minimal output, but this is most likely an area where the user may wish to tailor the code to his/her specific needs. In addition to the `status` variable, the only output returned as arguments by `PIKAIA` is (1) a real array `xb` of length `n`, containing the `n` parameter values associated with the best phenotype in the final population, and (2) a floating point scalar `fb` corresponding to the evaluation of `funk` for that phenotype. In many cases

[6] The exception concerns the use of a single labeled `COMMON` block to allow the random number routine `urand` to keep track of its `seed`.

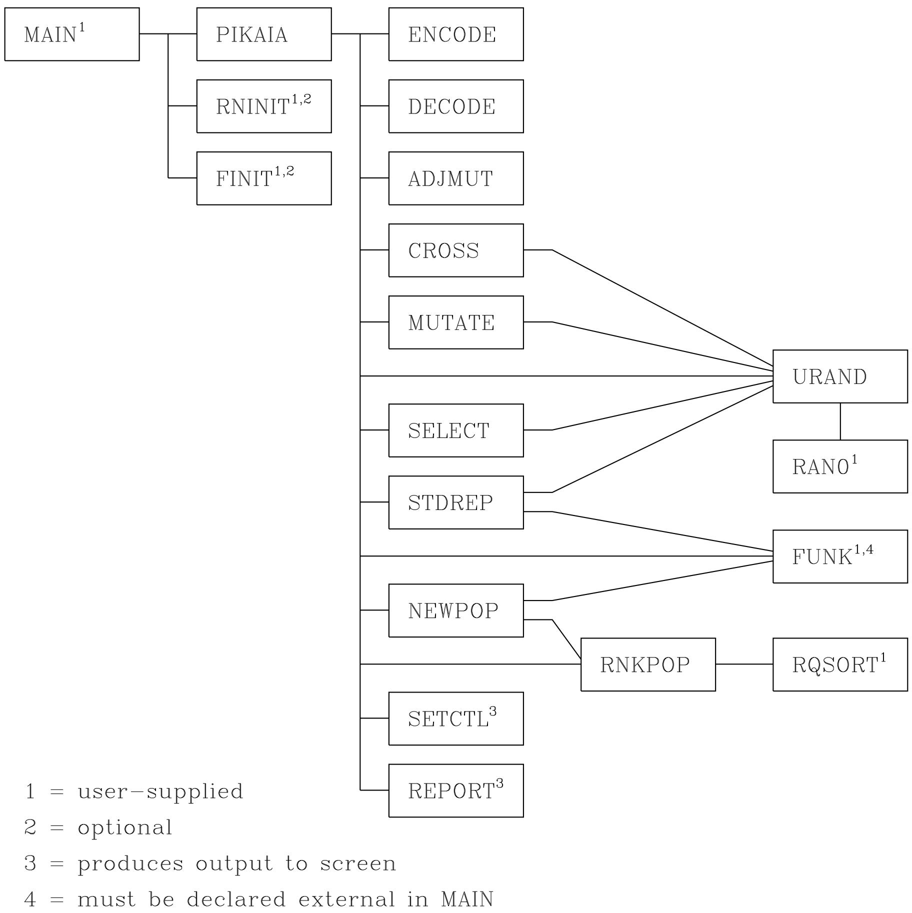

Figure 6.1: Dependency chart for `PIKAIA`, as called from a typical driver program, such as those used for the example problems of §§5.2 through 5.5, and listed in the Appendix.

it can be useful to output more information to file, for example the best and median phenotypes at each generation together with their associated fitness values (cf. Fig. 5.8), or maybe even the complete phenotype population (cf. Fig. 5.3). An appropriate place to output such information is at the end of the main generation loop, as indicated in the `PIKAIA` listing provided in the Appendix below.

Producing such additional output requires the user to be aware of where and how the population is stored. Internally, the population is stored in the 2-D array `oldph` (the 2-D array `newph` is only used as temporary storage when operating under full generational replacement). The ranking array `ifit` allows access to the population in terms of rank; the fittest phenotype is stored in `oldph(1:n,ifit(np))`, the second fittest in `oldph(1:n,ifit(np-1))`, and so on all the way to `oldph(1:n,ifit(1))` for the worst. The ordering of parameters for a given individual (a column in the population matrix) is the same as that associated with the input vector `x(1:n)` fed to the user-supplied fitness function. Once again, never forget that the parameters stored in the population matrix are normalized to the interval $[0.0, 1.0]$. The complementary array `jfit` contains the actual ranks: `jfit(i)` is the rank of the individual stored in `oldph(1:n,i)`. The array `fitns` contains the true fitness (i.e., `funk` evaluation) for each population member, with again the evaluation for the fittest being stored in `fitns(ifit(np))`, etc., as for `oldph`. Table IV below summarizes size and type information for these important internal arrays.

Table IV

Internal arrays

Array	type	true size	active size	description
`oldph`	real	(32,128)	(n,np)	main population
`newph`	real	(32,128)	(n,np)	new population
`fitns`	real	(128)	(np)	fitness values
`jfit`	integer	(128)	(np)	fitness-based rank
`ifit`	integer	(128)	(np)	rank key index

6.3 Incorporating additional strategies and operators

The modular design of `PIKAIA` was adopted in part to facilitate the incorporation of additional genetic operators and/or ecological strategies by the user. This section merely outlines some possible modifications/additions, as an initial bit of inspiration for users eager to experiment. In engaging in such experiments, the user should not be overly worried about degrading the code's performance by introducing additional computations and/or tests at the genetic or ecological levels; in most real applications, the bulk of the CPU time will be spent in the fitness function, which is consequently where efficiency considerations are most critical.

6.3.1 Variable selection pressure

`PIKAIA` is set up to function with constant selection pressure, as controlled by the fitness differential parameter `fdif`. It should be quite easy to let `fdif` (≡`ctrl(9)`) vary in the course of the evolutionary run, for example by following the procedure used to adjust the mutation rate `pmut` in subroutine `adjmut`.

In that context, one may also recall that eq. (3.14) is not the only possible way to monitor the degree of convergence in the population; eq. (3.14) uses the fitness, as defined by the user-supplied fitness function, to construct a dimensionless measure of convergence. In many applications it may be advantageous to use instead the Euclidian distance in parameter space between the fittest and median individuals. Let x_k^1 and x_k^m denote the parameter sets defining the best and median individual in `PIKAIA`'s `n`-dimensional bounded search space. The Euclidian distance between best and median is simply

$$\Delta S = \left(\sum_{k=1}^{n} (x_k^1 - x_k^m)^2 \right)^{1/2} \tag{6.1}$$

Note that this definition of ΔS is completely independent of the way in which fitness has been defined. Equation (6.1) can be used to control either the mutation rate or selection pressure, as the case may be. For "fitness landscapes" having broad and flat global maxima, this distance-based criterion can be expected to yield faster convergence than the fitness-based criterion defined in §3.7.

6.3.2 Creep mutation

The Hamming Wall problem discussed at the end of §5.3 is a serious one, and efforts should be made to correct it. It should be reasonably easy to modify the existing mutation operator (subroutine `mutate`) by forcing the operator to "carry over the one", as outlined in §5.3. Alternately, the user may wish to construct a *creep mutation* operator (see Davis 1991, chap. 5) designed to operate within the decimal encoding scheme adopted herein, and call *either* the creep mutation *or* `PIKAIA`'s standard mutation operator after the crossover step.

Another potentially interesting and straightforward modification consists in letting the mutation rate be a function of locus along the chromosome, genes mapping onto least significant digits being subjected to higher mutation rates. Such operators will typically require the setting of some additional ajustable parameters, such as the mutation rate gradient along the chromosome, etc., but can speed up convergence if high accuracy is absolutely required[7] (see, e.g., Michalewicz 1994, §6.2).

6.3.3 Floating-point-based genetic operators

One may also reflect upon the fact that the encoding/decoding process used in `PIKAIA` (§3.5) is somewhat contorted, and in some way even superfluous. The

[7] although in such a situation we would tend to advocate the hybrid approach outlined in §4.7.

choice of this encoding strategy was in part motivated by the (pedagogical) aim to produce a code operationally similar to more powerful, existing GA-based optimizers operating under binary encoding. Another motivation was to retain in `PIKAIA` the possibility of easy modification for handling mixed optimization problems, i.e., problems where some parameters can assume continuous values, and others discrete values[8]. Nonetheless, it is perfectly possible to construct a genetic algorithm that operates directly on the problem parameters as floating-point quantities. A "floating-point chromosome" can simply be the floating-point array containing the parameter values defining the corresponding phenotype. What would genetic operators for floating-point chromosomes look like? Note first that in the one-point crossover operation (§3.6), the value of only one of the `n` floating point parameters x_k defining the phenotype ends up being perturbed by the crossover operator, with the others being left unrouched or interchanged intact across chromosomes. Starting with the two parent phenotypes

$$x_k^1,\ x_k^2, \qquad k = 1, 2, ..., \mathtt{n}, \tag{6.2}$$

"floating-point crossover" can be defined by first generating a random integer $K \in [1, \mathtt{n}]$, and interchanging the elements $k > K$ of the parent phenotypes:

$$x_k^1 \rightleftharpoons x_k^2, \qquad k = K + 1, ..., \mathtt{n} \tag{6.3}$$

and leave elements 1 through $K - 1$ untouched. For element K, write something like

$$x_K^1 \leftarrow R \times x_K^1 + (1 - R) \times x_K^2, \tag{6.4a}$$

$$x_K^2 \leftarrow (1 - R) \times x_K^1 + R \times x_K^2, \tag{6.4b}$$

where, as before, $R \in [0, 1]$ is a random number drawn from a uniform distribution. Alternately, to preserve the logarithmic distribution of phenotypic effects characteristic of the standard one-point crossover operator of §3.6, one could write instead

$$x_K^1 \leftarrow (1 - 10^{-R}) \times x_K^1 + 10^{-R} \times x_K^2, \tag{6.5a}$$

$$x_K^2 \leftarrow 10^{-R} \times x_K^1 + (1 - 10^{-R}) \times x_K^2, \tag{6.5b}$$

[8] Consider for example the design of an optical system where certain quantities such as focal lengths, distance between components, etc., can assume continuous (but likely bounded) values, while refractive indices of existing optical glasses form a finite, discrete set. The current `PIKAIA` is much easier to modify to tackle such a problem than a purely floating-point based version.

where this time $R \in [0, \mathtt{nd}]$. Likewise, a floating-point mutation operator can be defined by running a probability test for each parameter x_k, and when the test yields true carry out the operation:

$$x_k \leftarrow x_k \pm 10^{-R}, \tag{6.6}$$

where again $R \in [0, \mathtt{nd}]$ is a random number. Note, in particular, that this approach to the mutation operator has the important advantage of cleanly avoiding the problem of Hamming Walls discussed above. One must however test to ensure that the post-mutation x_k still satisfies the parameter bounds.

6.3.4 Niches and multimodal optimization

In all examples treated in the preceding chapter it was implicitly assumed that the truly global best solution (the absolute greatest height in the multidimensional fitness "landscape") is the only one of interest. There are a number of applications where it may also be useful to be "informed" by the algorithm of other regions in parameter space having *nearly* the same fitness. A practical example encountered in data modeling is trying to locate all regions of parameter space having a χ^2 below the value deemed acceptable, as opposed to simply locate the absolute minimum in χ^2. Because it uses ranking and sustained selection pressure (via the constant fitness differential parameter `fdif`), `PIKAIA` can only home in on the absolute extremum; if run long enough, `PIKAIA` will *always* decimate individuals living on near-extremum peaks, in favor of the extremum peak.

This problem can be alleviated in a number of ways. By far the most powerful is the definition of *ecological niches*, so that individuals compete only with a local subset of the population. This is a vast topic, currently the subject of intense research effort on the part of the GA research community. The basic idea and simple implementations are discussed in Goldberg (1989, chap. 5), which would be a good starting point. Among more recent work on the topic, the user may note the papers by Beasley *et al.* (1993), Horn *et al.* (1994), and Cedeño *et al.* (1994).

6.4 Suggested further reading

The user planning to utilize `PIKAIA` in the context of some significant piece of research/development should seriously consider looking further into the theory of genetic algorithms, and become acquainted with some of the strategies and techniques not included in `PIKAIA`. Recommended starting points are the following:

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), chaps. 1 to 8.

Goldberg's book is essentially a textbook, complete with exercises and so on, while the first part of Davis's book takes the form of a tutorial. Actually, the first ancestral version of `PIKAIA` was developped on the fly as one of us (P.C.) made his way through the Davis book in the course of a dozen or so dark and stormy winter nights (really).

`PIKAIA` is a GA-based function optimizer, but genetic algorithms were originally developped 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 (and still extremely relevant) bible in the field, namely

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

The following are two more recent monographs also well worth looking into:

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

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

Be also on the lookout for an introductory text by Melanie Mitchell, to be published by MIT Press in early 1996. To get a feel for the wide (and ever-growing) range of application of genetic algorithm-based optimization, the interested reader may wish to take a look at chapters 9 to 22 of Davis's book, and at

Belew, R.K., & Booker, L.B. (eds.) 1991, *Proceedings of the fourth international Conference on Genetic Algorithms* (San Mateo: Morgan Kaufmann), sections VI and VII.

In the area of genetic algorithm theory, the following series of conference proceedings pretty much covers the current state of the art:

Rawlins, G.J.E. 1991 (ed.), *Foundations of Genetic Algorithms* (San Mateo: Morgan Kaufmann).

Whitley, L.D. 1993 (ed.), *Foundations of Genetic Algorithms 2* (San Mateo: Morgan Kaufmann).

Rawlins, G.J.E. 1995 (ed.), *Foundations of Genetic Algorithms 3* (San Mateo: Morgan Kaufmann).

Note finally that the journal *Evolutionary Computation*, edited by K. De Jong and published quarterly since 1993 by The MIT Press, is dedicated to the publication of research papers dealing with both theory and applications of genetic algorithms.

APPENDIX: SOURCE CODE

A.1 Source code for PIKAIA

The following is a reduced listing of the subroutine PIKAIA; It is “reduced” in the sense that the complete subroutine PIKAIA contains many more explanatory comments lines than the listing given here; all operational lines are included as they appear in the source code. Note also that in this reduced listing all REAL, INTEGER, etc, statements have been regrouped independently of their local/input/output status.

```
      subroutine pikaia(ff,n,ctrl,x,f,status)
      implicit   none
      real       ctrl(12),x(n),f,ff
      integer    n,status
      external   ff
c=========================================================================
c     Optimization (maximization) of user-supplied function ff over
c     n-dimensional parameter space  x  using a GA-based optimizer.
c
c     Paul Charbonneau & Barry Knapp
c     High Altitude Observatory
c     National Center for Atmospheric Research
c     Boulder CO 80307-3000
c     <paulchar@hao.ucar.edu>
c     <knapp@hao.ucar.edu>
c
c     Version 1.0   [ 1995 December 01 ]
c
c     Genetic algorithms (GA) are heuristic search techniques that
c     incorporate in a computational setting, the biological notion
c     of evolution by means of natural selection.  This subroutine
c     is a general purpose GA-based optimizer, incorporating the basic
c     GA operators of one-point crossover and mutation, as well as a
c     few additional robust reproductive and ecological strategies
c     known to improve the performance of GA in the context of
c     function optimization. Parameters defining the function to be
c     optimized (maximized) are encoded as string of simple decimal
c     integers [0,9].
c
c     References:
c
c        Goldberg, David E.  Genetic Algorithms in Search, Optimization,
c           & Machine Learning.  Addison-Wesley, 1989.
c
c        Davis, Lawrence, ed.  Handbook of Genetic Algorithms.
c           Van Nostrand Reinhold, 1991.
c=======================================================================
c     USES: setctl, ff, urand, rnkpop, select, encode, decode, cross,
c           mutate, genrep, stdrep, newpop, adjmut, report
      integer    NMAX, PMAX, DMAX
      parameter  (NMAX = 32, PMAX = 128, DMAX = 6)
      integer    np, nd, ngen, imut, irep, ielite, ivrb, k, ip, ig,
     +           ip1, ip2, new, newtot, gn1(NMAX*DMAX), gn2(NMAX*DMAX),
     +           ifit(PMAX), jfit(PMAX)
      real       pcross, pmut, pmutmn, pmutmx, fdif, fitns(PMAX),
     +           ph(NMAX,2), oldph(NMAX,PMAX), newph(NMAX,PMAX), urand
```

```
      external   urand
c     Set control variables from input and defaults
      call setctl
     +   (ctrl,n,np,ngen,nd,pcross,pmutmn,pmutmx,pmut,imut,
     +    fdif,irep,ielite,ivrb,status)
      if (status .ne. 0) then
         write(*,*) ' Control vector (ctrl) argument(s) invalid'
         return
      endif
c     Make sure locally-dimensioned arrays are big enough
      if (n.gt.NMAX .or. np.gt.PMAX .or. nd.gt.DMAX) then
         write(*,*)
     +      ' Number of parameters, population, or genes too large'
         status = -1
         return
      endif
c     Compute initial (random but bounded) phenotype population
      do 1 ip=1,np
         do 2 k=1,n
            oldph(k,ip)=urand()
    2    continue
         fitns(ip) = ff(n,oldph(1,ip))
    1 continue
c     Rank initial population by fitness order
      call rnkpop(np,fitns,ifit,jfit)
c     Main Generation Loop
      do 10 ig=1,ngen
c        Main Population Loop
         newtot=0
         do 20 ip=1,np/2
c           1. pick two parents
            call select(np,jfit,fdif,ip1)
   21       call select(np,jfit,fdif,ip2)
            if (ip1.eq.ip2) goto 21
c           2. encode parent phenotypes
            call encode(n,nd,oldph(1,ip1),gn1)
            call encode(n,nd,oldph(1,ip2),gn2)
c           3. breed
            call cross(n,nd,pcross,gn1,gn2)
            call mutate(n,nd,pmut,gn1)
            call mutate(n,nd,pmut,gn2)
c           4. decode offspring genotypes
            call decode(n,nd,gn1,ph(1,1))
            call decode(n,nd,gn2,ph(1,2))
c           5. insert into population
            if (irep.eq.1) then
```

```
            call genrep(NMAX,n,np,ip,ph,newph)
         else
            call stdrep(ff,NMAX,n,np,irep,ielite,
     +                  ph,oldph,fitns,ifit,jfit,new)
            newtot = newtot+new
         endif
c        End of Main Population Loop
   20    continue
c        if running full generational replacement: swap populations
         if (irep.eq.1)
     +      call newpop(ff,ielite,NMAX,n,np,oldph,newph,
     +         ifit,jfit,fitns,newtot)
c        adjust mutation rate?
         if (imut.eq.2) call adjmut(np,fitns,ifit,pmutmn,pmutmx,pmut)
c        print generation report to standard output?
         if (ivrb.gt.0) call report
     +      (ivrb,NMAX,n,np,nd,oldph,fitns,ifit,pmut,ig,newtot)
c**** User-supplied (optional) output routine could go here
c     End of Main Generation Loop
   10 continue
c     Return best phenotype and its fitness
      do 30 k=1,n
         x(k) = oldph(k,ifit(np))
   30 continue
      f = fitns(ifit(np))
      end
c*********************************************************************
      subroutine setctl
     +   (ctrl,n,np,ngen,nd,pcross,pmutmn,pmutmx,pmut,imut,
     +    fdif,irep,ielite,ivrb,status)
      implicit none
      integer    n, np, ngen, nd, imut, irep, ielite, ivrb, status
      real       pcross, pmutmn, pmutmx, pmut, fdif, ctrl(12)
c=====================================================================
c     Set control variables and flags from input and defaults
c=====================================================================
      integer    i
      real       DFAULT(12)
      save       DFAULT
      data       DFAULT /100,500,5,.85,2,.005,.0005,.25,1,1,1,0/
      do 1 i=1,12
         if (ctrl(i).lt.0.) ctrl(i)=DFAULT(i)
    1 continue
      np = ctrl(1)
      ngen = ctrl(2)
      nd = ctrl(3)
```

```
      pcross = ctrl(4)
      imut = ctrl(5)
      pmut = ctrl(6)
      pmutmn = ctrl(7)
      pmutmx = ctrl(8)
      fdif = ctrl(9)
      irep = ctrl(10)
      ielite = ctrl(11)
      ivrb = ctrl(12)
      status = 0
c     Print a header
      if (ivrb.gt.0) then
         write(*,2) ngen,np,n,nd,pcross,pmut,pmutmn,pmutmx,fdif
    2    format(/1x,60('*'),/,
     +      ' *',13x,'PIKAIA Genetic Algorithm Report ',13x,'*',/,
     +           1x,60('*'),//,
     +      '   Number of Generations evolving: ',i4,/,
     +      '       Individuals per generation: ',i4,/,
     +      '    Number of Chromosome segments: ',i4,/,
     +      '    Length of Chromosome segments: ',i4,/,
     +      '            Crossover probability: ',f9.4,/,
     +      '            Initial mutation rate: ',f9.4,/,
     +      '            Minimum mutation rate: ',f9.4,/,
     +      '            Maximum mutation rate: ',f9.4,/,
     +      '    Relative fitness differential: ',f9.4)
         if (imut.eq.1) write(*,3) 'Constant'
         if (imut.eq.2) write(*,3) 'Variable'
    3    format(
     +      '                    Mutation Mode: ',A)
         if (irep.eq.1) write(*,4) 'Full generational replacement'
         if (irep.eq.2) write(*,4) 'Steady-state-replace-random'
         if (irep.eq.3) write(*,4) 'Steady-state-replace-worst'
    4    format(
     +      '                Reproduction Plan: ',A)
      endif
c     Check some control values
      if (imut.ne.1 .and. imut.ne.2) then
         write(*,10)
         status = 5
      endif
   10 format(' ERROR: illegal value for imut (ctrl(5))')
      if (fdif.gt.1.) then
         write(*,11)
         status = 9
      endif
   11 format(' ERROR: illegal value for fdif (ctrl(9))')
```

```
      if (irep.ne.1 .and. irep.ne.2 .and. irep.ne.3) then
         write(*,12)
         status = 10
      endif
   12 format(' ERROR: illegal value for irep (ctrl(10))')
      if (pcross.gt.1.0 .or. pcross.lt.0.) then
         write(*,13)
         status = 4
      endif
   13 format(' ERROR: illegal value for pcross (ctrl(4))')
      if (ielite.ne.0 .and. ielite.ne.1) then
         write(*,14)
         status = 11
      endif
   14 format(' ERROR: illegal value for ielite (ctrl(11))')
      if (irep.eq.1 .and. imut.eq.1 .and. pmut.gt.0.5 .and.
     +    ielite.eq.0) then
         write(*,15)
      endif
   15 format(' WARNING: dangerously high value for pmut (ctrl(6));',
     +      /' (Should enforce elitism with ctrl(11)=1.)')
      if (irep.eq.1 .and. imut.eq.2 .and. pmutmx.gt.0.5 .and.
     +    ielite.eq.0) then
         write(*,16)
      endif
   16 format(' WARNING: dangerously high value for pmutmx (ctrl(8));',
     +      /' (Should enforce elitism with ctrl(11)=1.)')
      if (fdif.lt.0.33) then
         write(*,17)
      endif
   16 format(' WARNING: dangerously low value of fdif (ctrl(9))')
      if (mod(np,2).gt.0) then
         np=np-1
         write(*,18) np
      endif
   18 format(' WARNING: decreasing population size (ctrl(1)) to np=',i4)
      return
      end
c**********************************************************************
      subroutine report
     +   (ivrb,ndim,n,np,nd,oldph,fitns,ifit,pmut,ig,nnew)
      implicit none
      integer    ifit(np),ivrb,ndim,n,np,nd,ig,nnew
      real       oldph(ndim,np),fitns(np),pmut
c======================================================================
c     Write generation report to standard output
```

```
c=====================================================================
      real       bestft,pmutpv
      save       bestft,pmutpv
      integer    ndpwr,k
      logical    rpt
      data       bestft,pmutpv /0,0/
      rpt=.false.
      if (pmut.ne.pmutpv) then
         pmutpv=pmut
         rpt=.true.
      endif
      if (fitns(ifit(np)).ne.bestft) then
         bestft=fitns(ifit(np))
         rpt=.true.
      endif
      if (rpt .or. ivrb.ge.2) then
c        Power of 10 to make integer genotypes for display
         ndpwr = nint(10.**nd)
         write(*,'(/i6,i6,f10.6,4f10.6)') ig,nnew,pmut,
     +      fitns(ifit(np)), fitns(ifit(np-1)), fitns(ifit(np/2))
         do 15 k=1,n
            write(*,'(22x,3i10)')
     +         nint(ndpwr*oldph(k,ifit(np  ))),
     +         nint(ndpwr*oldph(k,ifit(np-1))),
     +         nint(ndpwr*oldph(k,ifit(np/2)))
   15    continue
      endif
      end
c*********************************************************************
      subroutine encode(n,nd,ph,gn)
      implicit none
      integer    n, nd, gn(n*nd)
      real       ph(n)
c=====================================================================
c     encode phenotype parameters into integer genotype
c     ph(k) are x,y coordinates [ 0 < x,y < 1 ]
c=====================================================================
      integer    ip, i, j, ii
      real       z
      z=10.**nd
      ii=0
      do 1 i=1,n
         ip=int(ph(i)*z)
         do 2 j=nd,1,-1
            gn(ii+j)=mod(ip,10)
            ip=ip/10
```

```
    2   continue
        ii=ii+nd
    1 continue
      return
      end
c**********************************************************************
      subroutine decode(n,nd,gn,ph)
      implicit none
      integer    n, nd, gn(n*nd)
      real       ph(n)
c======================================================================
c     decode genotype into phenotype parameters
c     ph(k) are x,y coordinates [ 0 < x,y < 1 ]
c======================================================================
      integer    ip, i, j, ii
      real       z
      z=10.**(-nd)
      ii=0
      do 1 i=1,n
         ip=0
         do 2 j=1,nd
            ip=10*ip+gn(ii+j)
    2    continue
         ph(i)=ip*z
         ii=ii+nd
    1 continue
      return
      end
c**********************************************************************
      subroutine cross(n,nd,pcross,gn1,gn2)
      implicit none
      integer    n, nd, gn1(n*nd), gn2(n*nd)
      real       pcross
c======================================================================
c     breeds two parent chromosomes into two offspring chromosomes
c     breeding occurs through crossover starting at position ispl
c======================================================================
c     USES: urand
      integer    i, ispl, t
      real       urand
      external   urand
c     Use crossover probability to decide whether a crossover occurs
      if (urand().lt.pcross) then
c        Compute crossover point
         ispl=int(urand()*n*nd)+1
c        Swap genes at ispl and above
```

```
        do 10 i=ispl,n*nd
           t=gn2(i)
           gn2(i)=gn1(i)
           gn1(i)=t
   10   continue
      endif
      return
      end
c***********************************************************************
      subroutine mutate(n,nd,pmut,gn)
      implicit none
      integer    n, nd, gn(n*nd)
      real       pmut
c=======================================================================
c     Mutations occur at rate pmut at all gene loci
c=======================================================================
c     USES: urand
      integer    i
      real       urand
      external   urand
      do 10 i=1,n*nd
         if (urand().lt.pmut) then
            gn(i)=int(urand()*10.)
         endif
   10 continue
      return
      end
c***********************************************************************
      subroutine adjmut(np,fitns,ifit,pmutmn,pmutmx,pmut)
      implicit none
      integer    np, ifit(np)
      real       fitns(np), pmutmn, pmutmx, pmut
c=======================================================================
c     dynamical adjustment of mutation rate; criterion is relative
c     difference in absolute fitnesses of best and median individuals
c=======================================================================
      real       rdif, rdiflo, rdifhi, delta
      parameter  (rdiflo=0.05, rdifhi=0.25, delta=1.5)
      rdif=abs(fitns(ifit(np))-fitns(ifit(np/2)))/
     +        (fitns(ifit(np))+fitns(ifit(np/2)))
      if(rdif.le.rdiflo)then
         pmut=min(pmutmx,pmut*delta)
      else if(rdif.ge.rdifhi)then
         pmut=max(pmutmn,pmut/delta)
      endif
      return
```

```
      end
c*****************************************************************
      subroutine select(np,jfit,fdif,idad)
      implicit none
      integer    np, jfit(np), idad
      real       fdif
c=================================================================
c     Selects a parent from the population, using roulette wheel
c     algorithm with the relative fitnesses of the phenotypes as
c     the "hit" probabilities [see Davis 1991, chap. 1].
c=================================================================
c     USES: urand
      integer    np1, i
      real       dice, rtfit, urand
      np1 = np+1
      dice = urand()*np*np1
      rtfit = 0.
      do 1 i=1,np
         rtfit = rtfit+np1+fdif*(np1-2*jfit(i))
         if (rtfit.ge.dice) then
            idad=i
            goto 2
         endif
    1 continue
    2 return
      end
c*****************************************************************
      subroutine rnkpop(n,arrin,indx,rank)
      implicit none
      integer    n, indx(n),rank(n)
      real       arrin(n)
c=================================================================
c     Calls external sort routine to produce key index and rank order
c     of input array arrin (which is not altered).
c=================================================================
c     USES: rqsort
      integer    n, indx(n),rank(n)
      real       arrin(n)
      integer    i
      external   rqsort
c**** User-supplied routine: Compute the key index
      call rqsort(n,arrin,indx)
c     ...and the rank order
      do 1 i=1,n
         rank(indx(i)) = n-i+1
    1 continue
```

```
      return
      end
c*********************************************************************
      subroutine genrep(ndim,n,np,ip,ph,newph)
      implicit none
      integer    ndim, n, np, ip
      real       ph(ndim,2), newph(ndim,np)
c=====================================================================
c     full generational replacement: accumulate offspring into new
c     population array
c=====================================================================
      integer    i1, i2, k
      i1=2*ip-1
      i2=i1+1
      do 1 k=1,n
         newph(k,i1)=ph(k,1)
         newph(k,i2)=ph(k,2)
    1 continue
      return
      end
c********************************************************************
      subroutine stdrep
     +   (ff,ndim,n,np,irep,ielite,ph,oldph,fitns,ifit,jfit,nnew)
      implicit none
      integer    ndim, n, np, irep, ielite, ifit(np), jfit(np), nnew
      real       ff, ph(ndim,2), oldph(ndim,np), fitns(np)
      external   ff
c====================================================================
c     steady-state reproduction: insert offspring pair into population
c     only if they are fit enough (replace-random if irep=2 or
c     replace-worst if irep=3).
c====================================================================
c     USES: ff, urand
      integer    i, j, k, i1, if1
      real       fit, urand
      external   urand
      nnew = 0
      do 1 j=1,2
c        1. compute offspring fitness (with caller's fitness function)
         fit=ff(n,ph(1,j))
c        2. if fit enough, insert in population
         do 20 i=np,1,-1
            if (fit.gt.fitns(ifit(i))) then
c              make sure the phenotype is not already in the population
               if (i.lt.np) then
                  do 5 k=1,n
```

```
                 if (oldph(k,ifit(i+1)).ne.ph(k,j)) goto 6
    5          continue
               goto 1
    6          continue
            endif
c           offspring is fit enough for insertion, and is unique
c           (i) insert phenotype at appropriate place in population
            if (irep.eq.3) then
               i1=1
            else if (ielite.eq.0 .or. i.eq.np) then
               i1=int(urand()*np)+1
            else
               i1=int(urand()*(np-1))+1
            endif
            if1 = ifit(i1)
            fitns(if1)=fit
            do 21 k=1,n
               oldph(k,if1)=ph(k,j)
   21       continue
c           (ii) shift and update ranking arrays
            if (i.lt.i1) then
c              shift up
               jfit(if1)=np-i
               do 22 k=i1-1,i+1,-1
                  jfit(ifit(k))=jfit(ifit(k))-1
                  ifit(k+1)=ifit(k)
   22          continue
               ifit(i+1)=if1
            else
c              shift down
               jfit(if1)=np-i+1
               do 23 k=i1+1,i
                  jfit(ifit(k))=jfit(ifit(k))+1
                  ifit(k-1)=ifit(k)
   23          continue
               ifit(i)=if1
            endif
            nnew = nnew+1
            goto 1
         endif
   20    continue
    1 continue
      return
      end
c***********************************************************************
      subroutine newpop
```

```
     +   (ff,ielite,ndim,n,np,oldph,newph,ifit,jfit,fitns,nnew)
      implicit none
      integer    ndim, np, n, ielite, ifit(np), jfit(np), nnew
      real       ff, fitns(np), oldph(ndim,np), newph(ndim,np)
      external   ff
c=======================================================================
c     replaces old population by new; recomputes fitnesses & ranks
c=======================================================================
c     USES: ff, rnkpop
      integer    i, k
      nnew = np
c     if using elitism, introduce in new population fittest of old
c     population (if greater than fitness of the individual it is
c     to replace)
      if (ielite.eq.1 .and. ff(n,newph(1,1)).lt.fitns(ifit(np))) then
         do 1 k=1,n
            newph(k,1)=oldph(k,ifit(np))
    1    continue
         nnew = nnew-1
      endif
c     replace population
      do 2 i=1,np
         do 3 k=1,n
            oldph(k,i)=newph(k,i)
    3    continue
c        get fitness using caller's fitness function
         fitns(i)=ff(n,oldph(1,i))
    2 continue
c     compute new population fitness rank order
      call rnkpop(np,fitns,ifit,jfit)
      return
      end
```

A.2 Random number generator and ranking subroutine

The following are source code listings for the random number generator and ranking subroutines provided with `PIKAIA`'s installation package. See §4.6 for details.

```
      function urand()
      implicit none
c=====================================================================
c     Return the next pseudo-random deviate from a sequence which is
c     uniformly distributed in the interval [0,1]
c
c     Uses the function ran0, the "minimal standard" random number
c     generator of Park and Miller (Comm. ACM 31, 1192-1201, Oct 1988;
c     Comm. ACM 36 No. 7, 105-110, July 1993).
c=====================================================================
      real     urand, ran0
      integer  iseed
      external ran0
c
c     Common block to make iseed visible to rninit (and to save
c     it between calls)
      common /rnseed/ iseed
c
      urand = ran0( iseed )
      return
      end
c*********************************************************************
      subroutine rninit( seed )
      implicit none
      integer    seed
c=====================================================================
c     Initialize random number generator urand with given seed
c=====================================================================
      integer iseed
c     Common block to communicate with urand
      common /rnseed/ iseed
c     Set the seed value
      iseed = seed
      if(iseed.le.0) iseed=123456
      return
      end
c*********************************************************************
      function ran0( seed )
c=====================================================================
c     "Minimal standard" pseudo-random number generator of Park and
c     Miller.  Returns a uniform random deviate r s.t. 0 < r < 1.0.
c     Set seed to any non-zero integer value to initialize a sequence,
c     then do not change seed between calls for successive deviates
c     in the sequence.
c
c     References:
```

```
c        Park, S. and Miller, K., "Random Number Generators: Good Ones
c           are Hard to Find", Comm. ACM 31, 1192-1201 (Oct. 1988)
c        Park, S. and Miller, K., in "Remarks on Choosing and Imple-
c           menting Random Number Generators", Comm. ACM 36 No. 7,
c           105-110 (July 1993)
c======================================================================
c *** Declaration section ***
      implicit none
c     Input/Output:
      integer seed
c     Output:
      real ran0
c     Constants:
      integer A,M,Q,R
      parameter (A=48271,M=2147483647,Q=44488,R=3399)
      real SCALE
      parameter (SCALE=1./M)
c     Local:
      integer j
c
c *** Executable section ***
c
      j = seed/Q
      seed = A*(seed-j*Q)-R*j
      if (seed .lt. 0) seed = seed+M
      ran0 = seed*SCALE
      return
      end
```

```
      subroutine rqsort(n,a,p)
      implicit none
      integer   n
      real      a(n), p(n)
c======================================================================
c     Return integer array p which indexes array a in increasing order.
c     Array a is not disturbed.  The Quicksort algorithm is used.
c
c     B. G. Knapp, 86/12/23
c
c     Reference: N. Wirth, Algorithms and Data Structures,
c     Prentice-Hall, 1986
c======================================================================
c     Constants
      integer   LGN, Q
      parameter (LGN=32, Q=11)
c        (LGN = log base 2 of maximum n;
c         Q = smallest subfile to use quicksort on)
c     Local:
      real      x
      integer   stackl(LGN),stackr(LGN),s,t,l,m,r,i,j
c     Initialize the stack
      stackl(1)=1
      stackr(1)=n
      s=1
c     Initialize the pointer array
      do 1 i=1,n
         p(i)=i
    1 continue
    2 if (s.gt.0) then
         l=stackl(s)
         r=stackr(s)
         s=s-1
    3    if ((r-l).lt.Q) then
c           Use straight insertion
            do 6 i=l+1,r
               t = p(i)
               x = a(t)
               do 4 j=i-1,l,-1
                  if (a(p(j)).le.x) goto 5
                  p(j+1) = p(j)
    4          continue
               j=l-1
    5          p(j+1) = t
    6       continue
         else
```

```
c           Use quicksort, with pivot as median of a(l), a(m), a(r)
            m=(l+r)/2
            t=p(m)
            if (a(t).lt.a(p(l))) then
               p(m)=p(l)
               p(l)=t
               t=p(m)
            endif
            if (a(t).gt.a(p(r))) then
               p(m)=p(r)
               p(r)=t
               t=p(m)
               if (a(t).lt.a(p(l))) then
                  p(m)=p(l)
                  p(l)=t
                  t=p(m)
               endif
            endif
c           Partition
            x=a(t)
            i=l+1
            j=r-1
    7       if (i.le.j) then
    8          if (a(p(i)).lt.x) then
                  i=i+1
                  goto 8
               endif
    9          if (x.lt.a(p(j))) then
                  j=j-1
                  goto 9
               endif
               if (i.le.j) then
                  t=p(i)
                  p(i)=p(j)
                  p(j)=t
                  i=i+1
                  j=j-1
               endif
               goto 7
            endif
c           Stack the larger subfile
            s=s+1
            if ((j-l).gt.(r-i)) then
               stackl(s)=l
               stackr(s)=j
               l=i
```

```
      else
         stackl(s)=i
         stackr(s)=r
         r=j
      endif
      goto 3
   endif
   goto 2
endif
return
end
```

A.3 Driver and fitness function for installation check

The following is a listing of the driver and fitness function for the installation check test problem. The driver `xpkaia` seeks to maximize a 2-D function `twod`, a listing of the latter being also provided. This function corresponds to the 2-D multimodal landscape shown on Fig. 5.1.

```
      program xpkaia
c=====================================================================
c     Driver program for the installation check of Section 2.3
c
c     This program performs repeated maximization of a 2-D function
c     named two_d, prompting the user for a random seed
c=====================================================================
      implicit none
      integer n, seed, i, status
      parameter (n=2)
      real ctrl(12), x(n), f, twod
      external twod
c**** First, initialize the random-number generator
c     (no other initialization required for this application)
    1 write(*,'(/A$)') ' Random number seed (I*4)? '
      read(*,*) seed
      call rninit(seed)
c     Set control variables (use defaults)
      do 10 i=1,12
         ctrl(i) = -1
   10 continue
c     Now call pikaia
      call pikaia(twod,n,ctrl,x,f,status)
c     Print the results
      write(*,*) ' status: ',status
      write(*,*) '      x: ',x
      write(*,*) '      f: ',f
      write(*,20) ctrl
 20   format(   '    ctrl: ',6f11.6/10x,6f11.6)
      goto 1
      end
      function  twod(n,x)
      real      x(n)
c====================================================================
c     Compute sample fitness function (altitude in 2-d landscape)
c====================================================================
      implicit  none
      integer   n,nn
      real      pi, sigma2, x(n), rr, twod
      parameter (pi=3.1415926536,sigma2=0.15,nn=9)
      if (x(1).gt.1..or.x(2).gt.1.) stop
      rr=sqrt( (0.5-x(1))**2+ (0.5-x(2))**2 )
      twod=cos(rr*nn*pi)**2 *exp(-rr**2/sigma2)
      return
      end
```

A.4 Drivers for the example problems of chapter 5

The following are driver source codes and, when appropriate, initialization subroutine(s), for the example problems of §§5.2 (`xpk1a`), 5.3 (`xpk1b`), 5.4 (`xpk2`), and 5.5 (`xpk3`). The corresponding fitness functions are listed in the main text.

```
      program xpk1a
c=================================================================
c     Driver program for linear least-squares problem
c     (Sect. 5.2)
c=================================================================
      implicit  none
      integer   n, seed, i, status
      parameter (n=2)
      real      ctrl(12), x(n), f, fit1a
      external  fit1a
c
c     First, initialize the random-number generator
c
      seed = 123456
      call rninit(seed)
c
c     Initializations
c
c     open(1,file='fake.i3e',form='unformatted')
      call finit
c
c     Set control variables (evolve 50 individuals over 100
c     generations, use defaults values for other input parameters)
c
      do 10 i=1,12
         ctrl(i) = -1
   10 continue
      ctrl(1)=50
      ctrl(2)=100
c
c     Now call pikaia
c
      call pikaia(fit1a,n,ctrl,x,f,status)
c
c     Print the results
      write(*,*) ' status: ',status
      write(*,*) '      x: ',x
      write(*,*) '      f: ',f
      write(*,20) ctrl
 20   format(   '    ctrl: ',6f11.6/10x,6f11.6)
      end
c*****************************************************************
      subroutine finit
c
c     Reads in synthetic dataset (see Figure 5.4)
c
```

```
implicit none
common/data/ f(200),t(200),sigma,ndata
dimension    f0(200),vdum(11)
real         f,t,sigma,f0,vdum
integer      ndata,i
open(1,file='syndat1.i3e',form='unformatted')
read(1) ndata
read(1) (vdum(i),i=1,11)
read(1) (t(i),i=1,ndata)
read(1) (f0(i),i=1,ndata)
read(1) (f(i),i=1,ndata)
sigma=5.
return
end
```

```
      program xpk1b
c=================================================================
c     Driver program for non-linear least-squares problem
c     (Sect. 5.3)
c=================================================================
      implicit  none
      integer   n, seed, i, status
      parameter (n=17)
      real      ctrl(12), x(n), f, fit1b
      external  fit1b
c
c     First, initialize the random-number generator
c
      seed=13579
      call rninit(seed)
c
c     Initializations
c
      call finit
c
c     Set control variables (use defaults except for population size)
c
      do 10 i=1,12
         ctrl(i) = -1
   10 continue
      ctrl(1)=50
c     Now call pikaia
      call pikaia(fit1b,n,ctrl,x,f,status)
c
c     Print the results
      write(*,*) ' status: ',status
      write(*,*) '      x: ',x
      write(*,*) '      f: ',f
      write(*,20) ctrl
 20   format(   '    ctrl: ',6f11.6/10x,6f11.6)
c
      end
c**************************************************************
      subroutine finit
c==============================================================
c     Reads in synthetic dataset (see Figure 5.4)
c==============================================================
      implicit     none
      common/data/ f(200),t(200),sigma,ndata,m
      dimension    f0(200),vdum(11)
      real         f,t,f0,vdum,sigma,delt
```

```
      integer      ndata,m,i
      open(1,file='syndat1.i3e',form='unformatted')
      read(1) ndata
      read(1) (vdum(i),i=1,11)
      read(1) (t(i),i=1,ndata)
      read(1) (f0(i),i=1,ndata)
      read(1) (f(i),i=1,ndata)
c     Use 5 Fourier modes for the fit
      m=5
c     same error bar for all point
      sigma=5.
      delt=t(3)-t(1)
      return
      end
```

```
      program xpk2
c=====================================================================
c     Driver program for ellipse fitting problem (Sect. 5.4)
c=====================================================================
      implicit  none
      integer   n, seed, i, status
      parameter (n=5)
      real      ctrl(12), x(n), f, fit2
      external  fit2
c
c     First, initialize the random-number generator
c
      seed=654321
      call rninit(seed)
c
c     Initializations
c
      call finit
c
c     Set control variables (50 individuals for 200 generations
c     under Select-random-delete-worst reproduction plan)
c
      do 10 i=1,12
         ctrl(i) = -1
   10 continue
      ctrl(1)=50
      ctrl(2)=200
      ctrl(9)=0.0
      ctrl(10)=3
c     Now call pikaia
      call pikaia(fit2,n,ctrl,x,f,status)
c
c     Print the results
      write(*,*) ' status: ',status
      write(*,*) '      x: ',x
      write(*,*) '      f: ',f
      write(*,20) ctrl
 20   format(   '    ctrl: ',6f11.6/10x,6f11.6)
      end
c******************************************************************
      subroutine finit
c==================================================================
c     Reads in synthetic dataset (see Figure 5.9)
c==================================================================
      implicit     none
      common/data/ xdat(200),ydat(200),ndata
```

```
      real          xdat,ydat
      integer       ndata,i
      open(1,file='syndat2.i3e',form='unformatted')
      read(1) ndata
      read(1) (xdat(i),i=1,ndata)
      read(1) (ydat(i),i=1,ndata)
      return
      end
c************************************************************
      function fatan(yy,xx)
c============================================================
c     Returns arctangent in full circle
c============================================================
      real yy,xx,fatan,a1,pi
      data pi/3.1415926536/
      a1=atan(yy/xx)
      if(xx.lt.0.) a1=a1+pi
      if(a1.lt.0.) a1=2.*pi+a1
      fatan=a1
      return
      end
```

```
      program xpk3
c===========================================================
c     Driver program for circle fitting problem using
c     a robust estimator based on the Hough transform
c     (Sect. 5.5)
c===========================================================
      implicit  none
      integer   n, seed, i, status
      parameter (n=2)
      real      ctrl(12), x(n), f, fit3
      external  fit3
c
c     First, initialize the random-number generator
c
      seed = 123456
      call rninit(seed)
c
c     Read in synthetic data
c
      call finit
c
c     Set control variables (use
c
      do 10 i=1,12
         ctrl(i) = -1
   10 continue
c     Now call pikaia
      call pikaia(fit3,n,ctrl,x,f,status)
c
c     Print the results
      write(*,*) ' status: ',status
      write(*,*) '      x: ',x
      write(*,*) '      f: ',f
      write(*,20) ctrl
 20   format(   '    ctrl: ',6f11.6/10x,6f11.6)
      end
c**************************************************************
      subroutine finit
c==============================================================
c     Reads in synthetic dataset
c==============================================================
      implicit     none
      common/data/ xdat(200),ydat(200),sigma,ndata
      real         xdat,ydat,sigma
      integer      ndata,i
      open(1,file='syndat3.i3e',form='unformatted')
```

```
      read(1) ndata
      read(1) (xdat(i),i=1,ndata)
      read(1) (ydat(i),i=1,ndata)
c     Same error estimate in x and y for all data points
      sigma=0.05
      return
      end
```

BIBLIOGRAPHY

The following are references for work quoted specifically throughout this guide. An annotated bibliography providing general entry points into the genetic algorithm literature is provided in §6.4.

Ballard, D.H., & Brown, C.M. 1982, *Computer Vision* (Englewood Cliffs: Prentice-Hall), chap. 4

Beasley, D., Bull, D.R., & Martin, R.R. 1994, Evolutionary Comp., 1, 101

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

Boggs, P.T., Byrd, R.H., & Schnabel, R.B. 1987, SIAM J. Sci. Stat. Comput., 8, 1052

Boggs, P.T., Donaldson, J.R., Byrd, R.H., & Schnabel, R.B. 1989, ACM Trans. Math. Software, 15, 348

Bowler, P.J. 1983, *Evolution* (Berkeley: University of California Press; revised edition 1989)

Cedeño, W., Vemuri, V.R., & Slezak, T. 1994, Evolutionary Comp., 2, 321

Cox, M.G., & Jones, H.M. 1989, IMA J. Numer. Anal., 9, 285

Craig, I.J.D., & Brown, J.C. 1986, *Inverse Problems in Astronomy*

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)

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

De Jong, K.A. 1993, in *Foundations of Genetic Algorithms 2*, ed. L.D. Whitley (San Mateo: Morgan Kaufmann), 5

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

Golub, G.H., & van Loan, C.F. 1989, *Matrix Computations*, (Baltimore: The Johns Hopkins University Press; second edition)

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

Hoare, C.A.R. 1962, Comp. J., 5 (No. 1), 10.

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

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

Horn, J., Goldberg, D.E., & Deb, K. 1994, Evolutionary Comp., 2, 37

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

Moura, L., & Kitney, R. 1991, Comp. Phys. Comm., 64, 57

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

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

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

Park, S. K., & Miller, K. W. 1988, Comm. of the ACM, 31, 1192

Park, S. K., Miller, K. W., & Stockmeyer, P. K. 1993, Comm. of the ACM, 36 (No. 7), 108

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

Sambridge, M., & Drijkonigen, G. 1992, Geophys. J. Int. 109, 323

Sen, M.K., & Stoffa, P.L. 1992, Geophys. J. Int. 108, 281

Syswerda, G. 1991, in *Foundations of Genetic Algorithms*, ed. G.J.E. Rawlins (San Mateo: Morgan Kaufmann), 94

Tomczyk, S., Charbonneau, P., Schou, J., and Thompson, M.J. 1995, in *4th SOHO Workshop: Helioseismology*, ESA Publication SP-375, (T. Hoeksema, ed.), 271

Wilson, W.G., & Vasudevan, K. 1991, Geophys. Res. Lett. 18, 2181

Wirth, N. 1986, *Algorithms and Data Structures* (Englewood Cliffs, NJ: Prentice-Hall)

Wright, A.H. 1991, in *Foundations of Genetic Algorithms*, ed. G.J.E. Rawlins (San Mateo: Morgan Kaufmann), 205

POSTFACE: PIKAIA AND *pikaia*

As this user's guide draws to a close one question may well remain on the mind of the reader: what the &%#@ is a "pikaia" anyway ? The PIKAIA described in these pages is a rather modest GA-based optimizer, as compared to a number of other genetic algorithm packages available commercially or in the public domain. Likewise, *Pikaia gracilens*, a little flattened worm-like beast some five centimeters long, crawling in the mud of a long gone seafloor 530 million years ago at the dawn of the Cambrian era, must have looked pretty insignificant compared to some of its contemporaries. Yet it is now believed by some paleontologists that *Pikaia* may well be a (if not "the") founder of the phylum *Chordata*, which met with a rather remarkable degree of success in the subsequent half billion years (e.g., Gould 1989). It would hardly be reasonable to hope for such a spectacular fate here. Nevertheless, if PIKAIA can help getting some real science done, or ends up serving as *Bauplan* for more serious undertakings, then the efforts and occasional frustrations having led to its existence will find themselves largely rewarded. Have fun.