Qubist

User's Guide

design. innovate. optimize!

Jason D. Fiege
Ph.D.

Qubist User's Guide

Optimization, Data-Modeling, and Visualization with the Qubist Global Optimization Toolbox for MATLAB®

© Jason D. Fiege, Ph.D. (2010)
nQube Technical Computing Corporation
www.nQube.ca

Qubist User's Guide: Optimization, Data-Modeling, and Visualization with the Qubist Global Optimization Toolbox for MATLAB

by Jason Fiege, Ph.D.

Printed in Canada.

Published by nQube Technical Computing Corporation, 75 Craigmohr Dr., Winnipeg, MB, Canada, R3T 6B9

This publication is available from nQube Technical Computing Corporation in either electronic or print format. Contact *info@nqube.ca* for further information, or visit our website at *www.nQube.ca*.

Printing History:
March 2010: First Edition

ISBN: 978-0-9865559-0-9 (electronic edition)
ISBN: 978-0-9865559-1-6 (print edition)

Contents

Chapter 1

Global Optimization with Qubist

nQube's goal is to remove the technical burdens of tough optimization and data-modeling problems as much as possible, so that you can focus on your particular applications.

Many problems in science and engineering can be reduced to the global optimization of parametric models or simulated systems, subject to some user-defined objective, or possibly multiple objectives. The simplest and possibly most common example is the problem of curve-fitting or data-modeling, which requires the minimization of a χ^2 function that determines the deviation from a perfect fit. Other examples common to engineering involve the optimization of a design, subject to one or several criteria. For simple problems with few parameters, the optimization procedure is simple and virtually any local optimization technique will work. However, things become more difficult when the number of parameters is large, noise is present, or multiple local optima exist in the parameter space. In such cases, simple optimization techniques will usually *fail* and one must resort to more sophisticated global optimization techniques.

Global optimization is the process of finding the 'best' feasible solution or family of solutions to a given problem that exists within the problem's parameter space, subject to one or more objectives and all applicable constraints. In general, this is a much harder problem than local optimization, which follows the topography of a function and ascends to the top of the first peak encountered, or descends to the bottom of the nearest valley (depending on whether the problem is one of maximization or minimization), without worrying about the possible existence of better solutions elsewhere in the parameter space. Global optimizers are designed to search a problem's parameter space thoroughly and provide some degree of confidence that a solution is globally optimal. Even so, it is seldom possible to guarantee mathematically that absolutely the best solution has been discovered, since this would require an exhaustive search of the entire parameter space, which is not practical unless the problem has very few parameters. There are no known global optimization procedures that are guaranteed to work on all problems, except for exhaustive search, and the greatest challenge of this field is to develop algorithms that are effective and efficient on a wide range of problems.

Qubist is third-party MATLAB toolbox for global optimization, data-modeling, and visualization that is written by myself and distributed by nQube Technical Computing Corporation. MATLAB 7.0 or higher is required, and the package works with Windows, MacIntosh (OS X and Power PC), and Linux. No additional MATLAB toolboxes are required.

The Qubist platform offers five complementary optimization algorithms

- Ferret: a parallel multi-objective genetic algorithm with sophisticated machine-learning features
- Locust: a parallel multi-objective particle swarm algorithm
- SAMOSA: a 'Simple Approach to a Multi-Objective Simplex Algorithm'
- Anvil: an innovative simulated anneal/genetic algorithm hybrid code
- SemiGloSS: a unique experimental 'Semi-Global Solution Spray' algorithm.

All Qubist optimizers are interchangeable and the package contains several other tools to assist with the analysis of runs and the visualization of multi-dimensional solution sets. Ferret and Locust are considered to be the 'front line' optimizers most suitable for very large or difficult problems. SAMOSA and Anvil are secondary optimizers that are less powerful than Ferret and Locust. SemiGloSS is an experimental optimizer that shows some promise, but is not ready for serious use. All of the Qubist optimizers, except for SemiGloSS, are discussed in this guide, as well as supporting tools for analysis and visualization.

1.0.1 Qubist's 'Front Line' Optimizers

Ferret is a powerful and flexible multi-objective genetic algorithm with numerous machine-learning enhancements, techniques borrowed from other evolutionary computing methods, and many unique features. Ferret is designed to handle extremely difficult optimization problems and is larger than all of the other Qubist optimizers combined. Locust is a multi-objective particle swarm optimizer that has undergone rapid recent development. It is a much smaller, 'snappier' code than Ferret, with fewer options and less computational overhead, but the latest versions rival Ferret on some problems. Together, Ferret and Locust are the best optimizers of the Qubist package for difficult problems. Both are designed as parallel algorithms that can take advantage of multi-CPU machines or clusters *without* requiring MATLAB's parallel computing toolbox or other third party software, and both both have sophisticated visualization capabilities to help you understand the parameter space of your problem. It is also noteworthy that both Ferret and Locust can call the secondary optimizers of the Qubist package (SAMOSA, Anvil, and SemiGloSS), as well as MATLAB's built-in optimizer fminsearch, as polishers to improve the quality of the solution set at the end of a run.

Up until this year, Ferret was *by far* the most powerful and sophisticated global optimizer in the Qubist package, and the clear choice for most problems. There were two reasons for this disparity of power. First of all, Ferret has been under constant development for over seven years at the time of this writing, while the other optimizers are only one to three years old. Secondly, and more importantly, the paradigm of biological evolution offers a very rich framework to design an optimizer. The dynamics of evolution is complex and offers more flexibility than the relatively simple dynamics of a particle swarm, the statistical process of annealing, or the simple rules of a simplex algorithm. The gap between Ferret and Locust has narrowed very recently - in fact during the writing of this user's guide. As I wrote, I cleaned out some questionable features of Locust that were muddying my understanding of the code, and the result was a new insight that improved the code dramatically. The new Locust-2 particle swarm optimizer is a very powerful optimizer - still perhaps not as powerful as Ferret on truly difficult problems, but the difference is much smaller than it was even six months ago. I expect that future development may narrow this margin even further. Locust has developed much more rapidly than Ferret, largely because in some way it *is* Ferret. The swarm equations are unique to Locust, but many other components have been recycled from

its older brother, including parts of the user interface, built-in visualization and analysis code, and even a large fraction of the internal logic. Nevertheless, Ferret remains my choice for tough, 'mission-critical' problems, partly due to its much longer development and testing history, as well as my own personal bias towards evolutionary codes.

1.0.2 Qubist's Secondary Optimizers and Polishers

Qubist's secondary optimizers can be used as as standalone optimizers for easier problems, or 'polishers' for tough problems to improve the accuracy of the final solution set. They are also useful for difficult problems when the emphasis is on obtaining good solution quickly, and not necessarily solutions that are truly optimal in a global sense. SAMOSA is an acronym for 'Simple Approach to a Multi-Objective Simplex Algorithm'. It is Qubist's default polisher for multi-objective optimization problems (see Section 2.1), and quite a useful tool for small stand-alone problems. As a simplex algorithm, SAMOSA has much in common with MATLAB's built-in fminsearch optimizer, except that it is designed for multi-objective problems like all Qubist optimizers, and has a nice graphical interface that you can optionally use for visualization of your run. SAMOSA is sufficiently lightweight that it can even be called from *inside* of a Ferret or Locust fitness function, for sophisticated problems that require an internal optimization step within an outer optimization handled by Ferret or Locust. Ferret also optionally uses SAMOSA as a local optimizer while as it runs to provide a local optimization step to speed up the convergence of multi-objective problems.[1]

Anvil is an innovative multi-objective simulated annealing/genetic algorithm hybrid code, whose genetic algorithm components borrow heavily from Ferret. Anvil is quite a powerful global optimizer in its own right, although not as powerful or as 'global' as Ferret and Locust for most problems. It has less computational overhead than either of the front line optimizers though, and might find a niche for medium-difficulty problems where speed is important. Like SAMOSA, Anvil can be used by both Ferret and Locust as a very thorough and robust solution polisher at the end of a run. However, it cannot be used during Ferret's local optimization step, where a more lightweight optimizer (SAMOSA or fminsearch) is required.

SemiGloSS is the 'Semi-Global Solution Spray' optimizer, which is an experimental multi-objective optimizer that is very fast, but not a true global optimizer. SemiGloSS is not ready ready for research-level problems, but is included for comparison. It may occasionally be useful as an alternative multi-objective solution polisher, but it probably should not be used as a standalone optimizer for problems of any importance. I have left SemiGloSS out of this user's guide because its future is highly uncertain at the time of this writing. Try it if you wish, but its features will remain undocumented, and subject to major changes, until the code is much further along in development.

1.0.3 The Big Picture

Qubist's design includes seamless integration between its component optimizers, simplicity of use, extreme flexibility, built-in parallelization, visualization tools, and 'knowledge discovery' capabilities that go beyond traditional optimizers. The Qubist package is best regarded as the 'front line' optimizers Ferret and Locust, plus a collection of visualization and analysis tools, and lesser optimizers that are suitable or polishing solutions and problems of lesser difficulty. Because of Ferret's dominant role the history of the Qubist project and the package, Qubist as a whole always carries the same version number as Ferret.

[1] Note that the local optimization step is distinct from polishing, since it happens *during* the run and takes relatively few steps. Polishing is optionally done at the end of the run, and is usually more thorough.

Locust has developed rapidly in recent history and I expect that this trend will continue for the near future, but Ferret still remains the gold standard of the current release.

The combination of Ferret, Locust, SAMOSA, and Anvil effectively covers the spectrum of optimization requirements for most users, and together represent a formidable and very complete set of optimization tools. An important strength of the Qubist package is that its components are completely integrated and consistent in their usage. The optimizers are 'swappable' in the sense that they all use the same formats for their fitness functions and understand Ferret's setup files. Once a project runs with Ferret, it will work automatically with any of the Qubist optimizers without even a single modification.

nQube's goal, and my goal personally, is to remove the technical burdens of tough optimization and data-modeling problems, as much as possible, so that you can focus on your particular applications. Qubist works toward this goal by providing advanced tools for global optimization and modeling of real-world problems, which are embedded in a simple and intuitive user interface designed to simplify the tasks of visualization, parameter space exploration, analysis, and parallel computing. The features that make this possible are introduced in Chapter 2 and discussed in more detail throughout the remainder of this user's guide.

1.0.4 About This User's Guide

As a final note in this chapter, I would like to point out that this user's guide is absolutely not intended as an academic treatise on GAs, evolutionary computing, or techniques for global optimization. My goal is perhaps less ambitious - I intend to document the features that I built into Qubist, discuss why things are the way that they are, and offer advice on how to configure and use this particular software effectively. In the process, I will show you how to configure Qubist to solve some very difficult optimization and modeling problems, while avoiding the pitfalls that new users commonly run into. I have written this guide in a very informal style with the hope that it might capture a wider audience in technical computing circles outside of academia. Certainly, there are some very advanced techniques 'under the hood' of Qubist, but fortunately you don't have to understand these techniques in detail to use the software effectively. With optimization problems, the proof is truly in the pudding - your final result is more important than how you found it, and furthermore, you can always verify whether your solution is better than one you obtained by other more conventional means. This guide is also deliberately light on references, which might disappoint some academics, but this would not be consistent with the conversational style of the text or my stated goals.

For those readers who would like a more formal introduction to genetic algorithms, I heartily recommend the two books by David Goldberg: 'Genetic Algorithms in Search, Optimization, and Machine Learning' [Goldberg, 1989] and 'The Design of Innovation: Lessons From and for Competent Genetic Algorithms.' [Goldberg, 2002]. His newer book was particularly influential to me as I developed Ferret's linkage-learning system, which is similar in spirit to the techniques described there, but quite different in operation. There are also plenty of academic papers on genetic algorithms, particle swarms, simulated annealing, and other global optimization techniques that can be readily found by searching electronic journals on the internet. I would encourage you to do so if you are at all interested in optimization as a field of study. However, if you just want to get down to business and tackle your own optimization and modeling problems, then this guide will give you sufficient tools and background information to do so.

Chapter 2

A Brief Tour of Qubist

'The computer programmer is a creator of universes for which he alone is the lawgiver. Universes of virtually unlimited complexity can be created in the form of computer programs.'

-Joseph Weizenbaum, *Computer Power and Human Reason*

In this chapter, we take a brief tour of the five optimizers in Qubist, as well as several other analysis and visualization tools that are included with the package. Ferret is Qubist's oldest optimizer, its most sophisticated, and really the origin of the entire Qubist package. It is therefore fitting that we should begin this chapter by discussing Qubist's flagship optimizer in some detail before moving on to the other components.

2.1 Multi-Objective Optimization

Multi-objective optimization can be a tricky concept the first time you encounter it, especially if you are used to working with single-objective optimizers. There are many real optimization problems in science, engineering, finance and other fields where the design goals are characterized by multiple objectives, but I like to use a very simple (albeit somewhat contrived) example to illustrate the key point. Imagine a factory manufacturing widgets, where the design of a widget is controlled by some set of parameters, and the goal is to find a widget design that simultaneously maximizes quality and minimizes production cost. Intuitively, higher quality widgets are usually more costly to make than lower quality widgets. What then is the optimal widget design that the factory should ultimately produce?

The correct answer is that there is no single optimal widget design. Rather, there is an optimal family of widgets defined by the property that you cannot improve the quality of any widget in the optimal set without simultaneously increasing its cost. Likewise, it is not possible to make any optimal widget cheaper without also degrading its quality. If you search harder and discover a new design that is simultaneously better and cheaper than some of the other designs in the current optimal set, then this new solution would be truly superior, and it would effectively remove all inferior solutions from the family of optimal solutions. Eventually you will discover a curve containing feasible solutions that you can't improve upon, which

represents the trade-offs between the design objectives. Trade-off surfaces are discussed commonly in engineering, where the goal is often to optimize a design subject to multiple criteria and constraints.

The trade-off surface is also referred to as the Pareto-optimal set, and is easily generalized to an arbitrary number of objectives. The Pareto-optimal set can be defined as the set of solutions to a multi-objective problem such that no objective can be improved without degrading at least one other. This definition can be made more mathematical, but from a practical standpoint, little is gained by doing so.

A multi-objective optimizer has a very difficult task, since it must search for solutions on the trade-off surface and then map the surface as completely as possible by populating it with points. Note that this is much more complicated than the goal of a standard optimizer, which usually seeks only a single optimal solution. All of the Qubist optimizers are multi-objective, but Ferret is best at mapping trade-off surfaces. Unfortunately, adding multi-objective capabilities to a genetic algorithm increases the complexity of its implementation substantially, although not necessarily its use. However, a good multi-objective GA can solve problems that you simply can't do with a single-objective optimizer. Furthermore, since a trade-off surface represents the necessary compromises between equally good feasible solutions that are all optimal, the user of such a code is presented with many alternatives solutions. Inevitably, the user is bound to prefer some of these solutions over others, even though all are equally good from a mathematical point of view, and this may in turn help the user to refine the problem's objectives.

The core feature of a multi-objective optimizer is its ability to spread solutions out over an optimal region of parameter space, for problems that do not admit a single optimal solution. This is absolutely required for multi-objective problems, but in practice, it is also often useful for a common type of data-modeling problem, where the data contain significant noise or uncertainty. Let me illustrate with the common problem of modeling data with noise, where you typically want to minimize the reduced chi-squared statistic over the space of n model parameters: $\chi^2_{reduced}(P_1, P_2, \ldots, P_n)$. In such problems, all models within about 1-sigma of the minimum value ($\chi^2_{reduced} < \chi^2_{min,reduced} + 1$) are virtually indistinguishable because their differences are buried in the noise. In a sense, this amounts to a 'fuzzy' optimization problem, where there is usually a single solution that is truly optimal, but also a finite-sized region that is nearly indistinguishable within the errors of the data. Mapping this region provides an intuitive and automatic method to determine all parameter values, their uncertainties, and the sensitivity of the objective function to each parameter.

In simple cases, the optimal region of a fuzzy single-objective problem forms a multi-dimensional ball surrounding the 'best' solution, but the Qubist optimizers sometimes discover more complex geometric structures in real-world problems. When this happens, it can tell you something important about the degeneracies of your model. If such a fuzzy optimal set contains models that range widely in their parameters, but 'look' similar to the fitness function in that they all produce essentially the same fitness value, then this is the symptom of mathematical degeneracy - two or more parameters playing off against each other in a way that makes their independent values *unknowable* from the existing data alone. Chapter 5 shows an example of how Ferret can be used to discover degeneracy in a model and infer its mathematical structure. This type of problem is where visualization of the optimal set becomes critical because mathematical degeneracy *looks* like a long curve of solutions snaking through the parameter space, a surface populated with points, or some higher-dimensional hypersurface. The point is that the signature of model degeneracy is usually quite obvious *when you see it*. All of the Qubist optimizers can be used for this sort of analysis, although Ferret and Locust do it most thoroughly. They are both very good at mapping these structures, and visualization tools are directly accessible from the Ferret and Locust interfaces to help you to see them.

All of the core optimizers in the Qubist package are designed for multi-objective optimization and

parameter space mapping. Ferret has the longest development history of these optimizers, and its multi-objective capabilities have been a core feature of the algorithm from the start. These multi-objective capabilities have been well-tested on a variety of test problems and difficult problems in research, and are regarded as robust and extremely reliable. Locust is one of few multi-objective particle swarm optimizers in existence, and its multi-objective feature is based almost entirely on robust code borrowed from Ferret. The same is true of Anvil, whose multi-objective capabilities also borrow heavily from Ferret. SAMOSA has a different goal. Rather than fully mapping the optimal set, SAMOSA aims to *either* map the trade-off surface sparsely with a few high quality solutions, *or* zero in on the nearest part of the trade-off surface to an initial guess, depending on how it is configured. This second configuration makes it extremely useful as a solution polisher for Ferret and Locust, but it is of limited use on problems that require a thoroughly mapped optimal set of solutions. All Qubist optimizers share a common interface to user-defined initialization, fitness, and custom graphics functions, and all can understand Ferret's setup file format. This makes it trivial to swap optimizers for comparison purposes.

One of Qubist's overall goals is to go beyond the limitations of a typical optimization package by offering nearly autonomous knowledge discovery tools, which can illuminate interesting features of a problem's parameter space and help direct the human user towards an improved understanding of the problem. Qubist's multi-objective and parameter space-mapping capabilities, in conjunction with good visualization tools, are the first and simplest features that illustrate this discovery capability. Ferret also contains other advanced features designed to elucidate the mathematical structure of a problem, which will be outlined in Sections 2.2.6 to 2.2.8. Some of these capabilities carry over to the other Qubist optimizers, which will be discussed in subsequent chapters.

2.2 The Ferret Genetic Algorithm

> *'Owing to this struggle for life, any variation, however slight and from whatever cause proceeding, if it be in any degree profitable to an individual of any species, in its infinitely complex relationship to other organic beings and to external nature, will tend to the preservation of that individual, and will generally be inherited by its offspring.'*
>
> -Charles Darwin, *The Origin of Species*

Genetic algorithms (GAs), like Ferret, are an important class of algorithms for global optimization that work in analogy to biological evolution. Evolution is biology's optimization strategy of choice, in which organisms evolve and continually improve their own designs as they struggle to survive. It is a common misconception that evolution is a random process that is dominated by mutation. It is more correct to regard evolution as a directed stochastic search, where random processes like mutation work in concert with non-random processes like genetic crossover (mating) and natural selection to optimize the genome of a species. GAs use the principles of biological evolution as the basis for a global optimization scheme.

2.2.1 History of the Ferret Project

Sometimes I find that a historical perspective is useful to put a project in context, and I will briefly recollect Ferret's seven-year history in this short section. Readers who prefer to skip the history and move on to the technical details should skip to Section 2.2.2.

The Ferret project began in 2002 while I was a Plaskett Research Fellow with the Canadian National Research Council in Victoria, British Columbia. At this time, I was grappling with a particularly difficult data-modeling problem, and I was getting nowhere using local optimizers, grid searches and random search techniques. My frustration was exacerbated by my attempts to visualize the results from my grid searches, and I recall producing some of the most confusing and least useful plots that I have ever generated. I distinctly remember an evening when I realized - in all seriousness - that every major problem that I had worked on since I was a Master's student had amounted to essentially the same problem: that of testing complicated theoretical models in astrophysics by applying them to even more complicated and noisy astronomical data sets. Furthermore, I realized that all of these problems were just difficult parameter search and optimization problems, and that a sufficiently powerful code for performing these tasks would go a long way in terms of solving all of my data-modeling problems at once. I turned to GAs because more standard techniques had failed miserably on my problem, and I built my first rather naive GA in about an hour. Nevertheless, I was delighted to discover that this rather primitive code was outperforming everything that I had tried up to that point. I was hooked, and I have spent much of my free time building evermore powerful GAs since that day.

I have never really considered Ferret (or later Qubist) to be part of my work as an astrophysicist, but rather a separate exploration to satisfy my curiosity, which was fortuitously well-aligned with my scientific goals. So I had an interesting new hobby, and Ferret really took on a life of its own about six months later. Ferret-1 was working quite nicely as a multi-objective code towards the end of 2003, and I had also built versions for Scilab (http://www.scilab.org) and Octave (http://www.gnu.org/software/octave), which are essentially free ports of the MATLAB language, with minor differences. I also built a parallel Java-based GA called 'Finch' (after Darwin's finches), which was somewhat different from Ferret, but about as powerful as Ferret was at the time. Eventually, this period of exploration ended and I decided that it was better to focus on a single software package rather than two. I chose Ferret over Finch because I realized that I could develop Ferret more rapidly than Finch due to my experience with MATLAB, and because I felt that Ferret had the greater potential to find users in the scientific computing community, where MATLAB and MATLAB-like languages are used extensively. I abandoned Finch, although some of the ideas and algorithms from this code influenced Ferret significantly due to their concurrent development.

I almost completely re-built Ferret in 2004-2005, after moving to Winnipeg, Manitoba to start my current position at the University of Manitoba. I had a lot of new responsibilities - most notably teaching - and I found it necessary to abandon the Scilab port of Ferret, as well as the Octave version, which was already lagging behind by the time I moved. I chose the MATLAB version because I had my sights set on applications in both academia *and* industry, where MATLAB is the gold standard. This was an innovative time in Ferret's development, and the time period when I developed Ferret's linkage-learning features. I actually developed two independent and successively more powerful linkage-learning systems during this period, first for Ferret-2 and then for Ferret-3, which each required quite a dramatic re-structuring of the code. By the end of 2005, Ferret had grown to several times its previous size, and there was very little left of the original code.

Around 2006, I began to put greater emphasis on developing an intuitive user interface and visualization system. This was motivated by the fact that an increasing number of people were using the code for problems of increasing complexity. Most of these early users were fellow physics and astronomy faculty members, or students at the University of Manitoba. I quickly discovered that this group is extremely good breaking any feature that is not absolutely robust, and Ferret went through an intense period of enhancements, crash-proofing, and overall hardening. I owe this group of people a debt of gratitude for their numerous bug reports and occasional feature requests. Moreover, the astronomy group was extremely interested in graphics and visualization of Ferret's parameter sets because of the very large astrophysical

data modeling projects that we were working on. As a result, my own interest in graphics and visualization increased, and I developed the visualization system that is integrated with the present version of the code. Finally, I built the current version of the code - Ferret-4 - in 2008. This new version was highlighted by many improvements to the graphics and visualization system, the addition of a built-in (and very easy to use) parallel computing system, the introduction of Ferret's strategy parameter auto-adaptation features, and user-modifiable 'skins' for all of the major operating systems. The transition to Ferret-4 was not as fundamental of a revision as Ferret-2 or Ferret-3 had been, but it included many cumulative improvements that together were at least as important, and therefore an increment in the version number was warranted. Incidentally, I began development of all of Qubist's other optimizers and supporting tools during the period from early 2007 - 2009, when Ferret-3 and 4 were under active development.

2.2.2 Ferret's Features

Ferret contains many enhancements that go far beyond a typical genetic algorithm and existing optimization packages. These enhancements include the following:

- Multi-objective search and parameter space mapping: Ferret is designed as a powerful multi-objective optimizer, which very effectively maps out and displays trade-off surfaces between multiple objective functions. This ability allows you to understand the compromises that must be made between several conflicting objectives, and can sometimes help to refine the true objectives of your problem if you discover that you prefer one part of the trade-off surface over another. The same powerful parameter space mapping machinery used for multi-objective problems can also be unleashed on problems with only a single objective. It is often useful to allow some tolerance on the objective function (or fitness function), so that Ferret returns the true optimum, plus a scattering of points within the tolerance. Why is this useful? In any data-modeling problem, one is always concerned with both the best-fit and the errors on the fit. If one is minimizing a reduced χ^2 function, for example, it is easy to tell Ferret to minimize the function, but also map out the entire region within $\Delta\chi^2_{reduced} = 1$ (say) of the minimum to map the 1-sigma confidence region. In effect, this provides built-in error analysis.
- Built-in parallelization, which does *not* require the user to purchase MATLAB's parallel computing toolbox or use any other software not included with Qubist.
- Simple handling of discrete and cyclic parameters.
- Automatic zooming: Ferret can be configured to automatically zoom in on the global solution as the population converges toward the optimal set. This allows the code to find the optimal set efficiently and to high accuracy.
- Critical Parameter Detection: Ferret contains a unique system for *Critical Parameter Detection* (CPD), which allows the code to dynamically monitor the set of parameters during a run to determine which ones are actually important to the optimization. If a parameter is determined to be unimportant, it is explicitly ignored, thus reducing the size of the parameter space.
- Linkage-Learning: A novel *linkage-learning* algorithm that is designed to reduce a complex, multi-parameter problem to a natural set of smaller sub-problems whenever such a reduction is possible. These simpler sub-problems are discovered experimentally by Ferret during the process of optimization, and strongly partitioned sub-problems evolve almost independently during a run. This process is dynamic. Parameters that appear linked at the start of a run may not appear linked at the end, when most trial solutions may be nearly optimal. Conversely, new links can also arise as the

code explores previously uncharted regions of parameter space. The ability to partition a complicated problem into *natural* sub-problems is crucial to the successful optimization of large problems. A difficult 100-parameter problem with many local minima is probably unsolvable on its own, but it becomes quite tractable *if* it can be partitioned into (say) ten sub-problems (or building blocks) with ten parameters each. An interesting feature of Ferret's linkage-learning system is that the linkages discovered are entirely insensitive to scale. Two sub-problems that are orders of magnitude different in importance are discovered at the same rate, so that Ferret can solve all of the sub-problems correctly and simultaneously, rather than one at a time in order of significance. This ability allows Ferret to discover the true, globally optimal solution or solution set, even when applied to problems with very poorly scaled building blocks.

- Advanced Lethal Suppression (ALS): 'Lethals' are poor quality solutions that are sometimes obtained from the crossover of two good parent solutions. Lethals degrade the performance of a GA, since they represent wasted solutions that are likely to be be discarded during the next round of tournament selection. When enabled, Ferret's ALS algorithm allows the code to learn what parts of the parameter space are likely to contain lethals, and avoid these regions during crossovers.

- Strategy Auto-Adaptation: Ferret contains a powerful algorithm that monitors its progress and uses this information to automatically adapt several of its most important control parameters, including the mutation scale, size scale of crossover events, and several others. If these parameters are set poorly by the user, Ferret will quickly and dynamically adapt them to improve the search.

- Pausing, stopping, and resuming runs: Ferret produces 'History' files every few generations, so that no information is lost if the user chooses to suspend a run, and only minimal information is lost if MATLAB or the computer system happens to crash.

- Analysis and integration of other Qubist optimizers: You can analyze a Ferret run while it is in progress or after it is complete to compute the optimal set of solutions. These solutions can be fine-tuned by your choice of three different polishers with a single button click.

- Integrated visualization: You can visualize the progress of a Ferret run while it is in progress through an intuitive and informative interface. A more sophisticated interface is available after the run is analyzed to find the optimal set. The analysis window contains various graphics options to tease out interesting features from the optimal set. These features include two and three-dimensional scatter plots, image plots, contour plots, and user-defined graphics. You can also 'paint' interesting regions of the parameter space and easily select different two and three-dimensional projections to explore and visualize where the painted solutions reside in a high-dimensional space. Of course, you can turn off Ferret's graphical user interface if desired, and Qubist's visualization tools can still be applied to the final solution set at the end of the run.

In conclusion, it should be clear that Ferret bears little resemblance to traditional GAs like the original version presented by John Holland in 1975 [Holland, 1975]. Ferret is an integrated package for parameter search, global optimization, data-modeling, and visualization, based on an evolutionary algorithm that is much more sophisticated than Holland's original GA or subsequent GAs like those discussed by David Goldberg is his authoritative 1989 monograph on the subject [Goldberg, 1989]. Ferret works with real-valued parameters and self-adaptive mutation scales, much like an Evolution Strategies (ES) code, but Ferret remains closer to a GA than an ES code due to its emphasis on crossovers. As a multi-objective GA, Ferret has some features in common with other multi-objective GAs that have appeared in the literature (e.g. Fonseca & Fleming [1993], Horn et al. [1994]). However, Ferret blends these multi-objective capabilities with powerful linkage learning techniques that are in the same spirit (but quite different in

their internal workings) as those discussed by Goldberg [2002]. Visualization is central to the package because Ferret is designed to help you to discover the meaning of your results, and seeing the structure of the multi-dimensional optimal set is the first step in this process. Ferret's built-in parallelization system takes advantage of modern multi-CPU machines and clusters. It does not require any knowledge of parallel computing - literally, you just select the number of 'worker nodes' that you want from a drop-down menu, click a start button, and the rest is automatic.

I believe that Ferret is the only GA in existence (at the time of this writing) that contains the powerful combination of features discussed above. Ferret is designed for real-world research problems, and as an active scientific researcher, I have been the code's harshest critic and most demanding user. I have spent much of the past seven years finding problems, fixing them, developing new techniques, and pushing Ferret's limits to handle massive-scale optimization and data-modeling problems on fairly modest computing hardware.

Most of this manual will be devoted to discussing Ferret and how to configure its advanced features to suit specific types of problems. The chapters on SAMOSA, Anvil, and Locust are much shorter because these optimizers are much simpler than Ferret, but they share a great deal with Ferret in terms of their configuration options and usage. All of the other optimizers understand fitness functions and setup files made for Ferret. Therefore, it is a fair statement that once you learn to use Ferret, you will have an excellent working knowledge of the entire Qubist package.

2.2.3 Multi-Objective Optimization with Ferret

Of all of the Qubist optimizers, Ferret has the most advanced and thoroughly tested features for multi-objective optimization. Multi-objective genetic algorithms are wonderful tools for mapping out optimal sets and multi-objective trade-off surfaces, but basic genetic algorithms are not well suited for this purpose.

Consider so-called 'scalarization' techniques, which attempt to solve a multi-objective problem with N objectives, by converting it into an equivalent scalar-valued single objective problem:

$$f(\mathbf{x}) = \sum_{i=1}^{N} w_i F_i = \mathbf{w} \cdot \mathbf{F}, \tag{2.1}$$

where $\mathbf{w}$ is a vector of weights such that $|\mathbf{w}|=1$. The geometric interpretation is clear: optimization is performed in the direction of $\mathbf{w}$, and the Pareto front can be mapped by varying $\mathbf{w}$ over all possible values, *provided that the trade-off surface is geometrically convex*. However, non-convex regions of the Pareto surface will be missed entirely by this approach. Ferret does not use scalarization and performs very well on non-convex problems. This means that Ferret can be used to explore more general fitness functions, without worrying about the issue of convexivity.

Let us examine the following problem as a trivial example of a non-convex multi-objective minimization problem on the interval $\theta \in [0, \pi/2]$:

$$R(\theta) = \frac{1 + k\cos^2(2\theta)}{1+k}; \quad F_1(\theta) = R\cos\theta; \quad F_2(\theta) = R\sin\theta, \tag{2.2}$$

where $F_1(\theta)$ and $F_2(\theta)$ are functions to be minimized, and k is a constant. Figure 2.1 shows the Pareto surface of equation 2.2 for constant $k = 0$ (left), $k = 1$ (middle), and $k = 2$ (right). Solutions from Ferret are shown as blue dots that join together to form a continuous curve for $k = 0$ and $k = 1$, but the curve is

correctly broken into three isolated 'islands' of solutions when $k = 2$. A scalarization method using equation 2.1 only finds the convex portions of the curve populated with red dots or the endpoints shown as red stars. We note that scalarization missed much of the Pareto surface for all values of k shown.

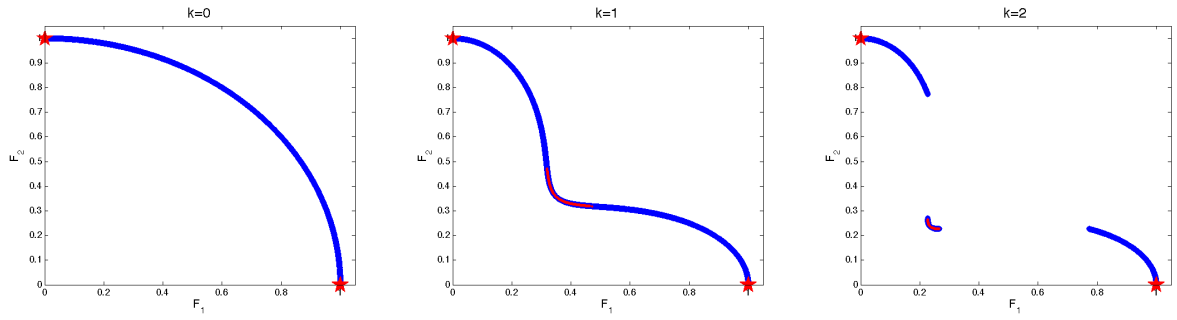

Figure 2.1: Comparison of Ferret to a scalarization method for a simple non-convex problem with two objectives (equation 2.2). Ferret maps a continuous trade-off surface shown in blue when $k = 0$ and $k = 1$, but the Pareto surface is correctly broken into three isolated 'islands' when $k = 2$. The red dots and red stars are solutions found using a scalarization approach. Ferret maps the entire Pareto surface for all three cases, but the scalarization method always misses non-convex parts of the Pareto surface.

A single-objective GA (plus scalarization) is not ideal for mapping trade-off surfaces even if your problem is known to be convex, and it is also not ideal for mapping multi-dimensional confidence intervals in data-modeling problems (see Section 2.2.2) for the same underlying reason. Genetic algorithms naturally suffer from a well-known property called genetic drift, which tends to cause a population to converge to some arbitrary point within the optimal set rather than mapping it out. The basic problem is that genetic algorithms require a 'crossover' step, in which two solutions are combined to generate a new solution, and a selection step, which destroys the weaker solution in favour of the stronger. Both of these operations tend to destroy the 'diversity' of the solution set - selection directly destroys a solution, but crossover is more subtle. It destroys diversity because it is essentially an averaging operation that produces a 'child' solution with properties that are in some way intermediate between the parents. When this is done repeatedly, the solutions tend toward uniformity - i.e. a single point in parameter space. In some ways, I find it strange that a class of algorithms with this serious defect could become the basis for a powerful multi-objective code, but there are well-known techniques that correct this problem. Multi-objective genetic algorithms like Ferret require sophisticated schemes, usually based on a 'niching' algorithm (see Section 8.9), to 'prop up' the optimal set against over-convergence due to genetic drift. A good niching scheme can be quite tricky to design, and this was one of the major successes of Ferret-1 - the first generation of the Ferret code.

2.2.4 Cyclic and Discrete Parameters:

By default, Ferret is configured as a bounded optimizer, which searches a parameter space with well-defined minimum and maximum boundaries in each parameter.[1] A bounded parameter space can be represented geometrically as an n-dimensional box, where n is the number of parameters. However, it is

[1] This can be relaxed however, by specifying some parameter boundaries as 'soft', which allows Ferret to wander somewhat outside of the pre-defined boundaries if suspects that there are better solutions outside. See parameters `par.general.softMin` and `par.general.softMax`, discussed in Section 8.3.

possible to indicate that some parameters are cyclic, which tells the code that the two opposite extremes of these parameter are really the same. Such parameters often represent angles, where 0 and 2π radians normally represent the same angle. In such cases, Ferret joins the corresponding opposite edges of the search box, so that it becomes geometrically equivalent to an n-dimensional torus. While this may be difficult to visualize for more than two parameters, the details are handled internally and transparently to the user once a parameter had been designated as cyclic.

Ferret can also handle parameters that are only allowed to take on only certain discrete values with some uniform spacing. This is intended for real-valued parameters that represent the ordered state of a system, but have underlying constraints that force them into discrete values. However, it is not meant for combinatorial optimization problems, where a parameter represents an arbitrary state of the system, without any natural ordering of the states. For example, an angle that is constrained to take on discrete values in increments of ten degrees is fine, because the angles have a natural ordering. On the other hand, a parameter that labels the cities on a map in a traveling salesman problem is not an ordered discrete parameter and will not be handled as well by Ferret. This is not to say that it is impossible to represent such a parameter in Ferret, or that the code won't make any progress on such a problem. Usually it will make reasonable progress anyway, but combinatorial optimization techniques are better suited for these types of problems, and Ferret is not designed as a combinatorial optimizer.

2.2.5 Automatic Zooming:

Ferret can be configured to dynamically zoom in on a problem's high performance region - the region of parameter space containing the currently best solution or solutions - as a run progresses. This works by explicitly changing the size of the search box, and allows the code to automatically restrict the range of the parameters to the part of the space where the best solutions are known to reside. Zooming improves both the rate of convergence and the accuracy of the final solutions. Ferret can also automatically zoom back out if solutions begin clustering near the boundary of the search box after zooming has occurred, since this may indicate that the optimal solution, or at least some part of the optimal set, resides outside of the search box.

2.2.6 Critical Parameter Detection

It is not uncommon in large, non-linear parameter search and optimization problems for the user to initially have an incomplete understanding of the role that each parameter plays in minimizing the problem's fitness function or functions. For example, some parameters may be much more important than others, or some parameters may even be nearly irrelevant. Imagine an extreme example of optimizing the settings of an industrial machine that is controlled by 100 dials, but unbeknownst to the operator, only ten of the dials actually have a significant effect on the process being optimized. Obviously, the key to solving such a problem is to discover which dials are important, and forget about the rest, since this would transform a nearly intractable 100-dimensional problem into a much more manageable problem with only ten parameters.

The example given above is obviously a very extreme and artificial case. However, it is not uncommon in real-world problems to discover that at least *some* parameters are much less important than others. In my experience, the ability to reduce the dimensionality of a parameter space by even one or two parameters can make a problem noticeably easier. On a more theoretical level, if Ferret consistently tells you that a particular parameter of your model is unimportant, then you should probably try to understand why this is

happening, and remove it from the model if at all possible.

Ferret is designed to automatically challenge the importance of parameters during a run whenever its 'Critical Parameter Detection' (CPD) feature is turned on. If there are parameters of little importance in the problem, Ferret very quickly realizes this and effectively reduces the dimensionality of the parameter space by ignoring, or paying less attention, to this parameter in the search. Ferret's interface displays a bar graph that indicates the qualitative importance of each parameter. More accurately, the height of each bar represents the number of fully specified copies of the corresponding 'gene' (or parameter) present in the entire set of populations.[2] Where genes are not fully represented, they are encoded internally as NaN, which Ferret normally interprets as a random number within the parameter range. Section 2.2.6 will show you how to set up this feature.

Of course, the relevance of a parameter is ofen contextual. A parameter that is not relevant early in the search may become more important as the algorithm zooms in on the optimal set, or one that was important during early evolution may lose its relevance later on. Ferret constantly checks the importance of each parameter as a run progresses, and dynamically adapts its internal model of parameter importance, as new regions of the fitness landscape are sampled. Both Anvil and SemiGloSS contain a weaker form of CPD, which may be useful on some problems. However, Locust does not contain any feature analogous to CPD.

I have personally found this feature useful in an astrophysical modeling code called GalAPAGOS, developed in collaboration with my colleague Dr. Jayanne English at the University of Manitoba, which involves modeling hydrogen gas disks in galaxies [Fiege et al., 2007]. For this problem, my colleagues and I have developed a very large model with 23-26 parameters, depending on how we set it up, that represents the rotating gas disk of a galaxy. Our task is to fit it to a three-dimensional 'spectral data cube' with two spatial dimensions and one velocity dimension, and to map out the region of interest within 1-sigma of the χ^2 minimum. This is clearly a daunting task. However, we find that the model is simplified for some data sets, since a significant fraction of the parameters (1/4 to 1/2 of them occasionally) become unimportant within our 1-sigma noise threshold! Only one or two parameters become unimportant for more difficult data sets. Nevertheless, the CPD feature is important to the project as a whole, because it allows us to concentrate our efforts and computing resources on the data sets that are intrinsically difficult.

2.2.7 Linkage Learning

We hear a lot these days about new holistic approaches in medicine, ecosystem management, the environment, economics, and many other fields. These fields share the property that the system as a whole displays complex behavior, which is often unexpected, as a result of a complicated network of interactions between its component parts. This is also a reasonable description of the Qubist optimizers Ferret and Locust (and to a lesser extent Anvil), because they rely on the emergent properties of a population of interacting individuals or swarming particles to search a parameter space. I will even go so far as to say that I took a somewhat holistic approach at times in developing this software, because I spent many hours observing how each subtle change to the dynamics of the interactions affected the behaviour of the population or swarm as a whole - often without *really* being able to reduce the behaviour to a straightforward cause and effect relationship, or being able to fully disentangle it from other effects.

[2]Ferret is a multi-population GA, in which it is possible to run multiple, quasi-independent populations. This is sometimes advantageous because it allows the population to form multiple non-competing groups within the space. This decreases the rate of convergence of the population and tends to encourage more thorough exploration. The set of populations is referred to (somewhat cheekily) as the *world*, and populations exchange individuals only very occasionally through an 'immigration' operator. See Section 4.9.1 and Chapter 8 for details.

A holistic approach is sometimes warranted for problems of the type discussed above, but a reductionist approach is often more efficient and reliable for problems that do not involve such complicated networks of interactions between components parts. Humans naturally solve complex problems by a 'divide & conquer' strategy that relies on our intuitive ability to break large problems into smaller, nearly independent sub-problems that can be solved almost independently and linked together to construct the full solution. This reductionist approach has been a cornerstone of the scientific method, and centuries of innovation have proven that it works.

A major goal in GA research is to build 'linkage-learning' algorithms with this reductionist capability. A linkage-learning algorithm automatically searches for ways to partition problems with many parameters into several semi-independent problems, which each has a smaller number of parameters. Small to medium-sized problems with less than about ten to fifteen parameters are often quite tractable, but many problems encountered in science, engineering, and other technical disciplines are much larger. A linkage learning algorithm offers a powerful new strategy to solve such problems by mirroring the reductionist approach to innovation developed by humans. It intelligently partitions big problems with many parameters into smaller and simpler problems, which are often relatively easy (or at least manageable) on their own, optimizes each component independently or with minimal interaction with other components, and re-assembles the solution from its fundamental component parts. Ferret is very good at doing this, and it does so automatically. In the language of the holistic vs. reductionist discussion above, Ferret takes a reductionist approach wherever it can, and a holistic one on the highly non-linear sub-problems where this doesn't work. Of course, the key is knowing where to make the division between parameters that can be broken into sub-problems, and those that are indivisible. This is the job of Ferret's linkage learning system, and I am happy to report that it works very well.

The basic idea of Ferret's linkage-learning system is simple. Ferret regards two parameters A and B as linked if non-linear saddle-like behaviour is detected, such that variations of A and B independently result in worsening of a solution, but the same variations applied together result in improvement. In such cases, it is clear that A and B should be linked so that they are (usually) traded together during crossovers, to preserve gains made by varying the parameters together. A novel extension of Ferrets linkage-learning algorithm is its ability to search entire sets of parameters A and B for linkage in parallel, which is assigned probabilistically to the parameters within these sets. Thus, Ferret treats linkage as a matrix of probabilities that co-evolves with the population during the search. Parameters that appear linked at the start of a run may not appear linked at the end, when most solutions may be nearly optimal. Conversely, new links can also arise as the code explores previously uncharted regions of parameter space.

Ferret's linkage-learning algorithm is based on a dynamically evolving internal model of a problem's linkage structure. Like CPD, a problem's linkage state is not static, and it is very often the case that parameters that need to be linked early on in a run can be unlinked at later times. This is an opportunity that should not be missed, because unlinking may improve the efficiency of the search very substantially. On the other hand, parameters that were unlinked at early times may need to be linked late in a run when the population begins to converge toward its final solution set. Linkages must be discovered reliably, since a failure to do so could result in the code settling into a false local minimum.

Ferret constantly tries to refine its linkage map each generation throughout the duration of a run. This occurs while the optimization is in progress, and is handled entirely automatically, without any user input. However, the evolving *linkage matrix* is displayed in the user interface, and this often holds clues about a problem's mathematical structure and level of difficulty, because problems with many linked parameters are more difficult that those with few linkages. Each generation, Ferret partitions a problem into sub-problems using its best current model of the linkage structure, performs evolutionary operations on building blocks composed of linked parameters, as well as other operations that are independent of linkage

structure, and re-assembles a full solution for each parameter set. This process is performed iteratively over all of the generations of the run.

The search for linked 'building blocks' amounts to a search for certain non-linear mathematical structures in the parameter space. This search takes place while the parameters themselves are being searched so that the linkage structure and the parameters co-evolve. Ferret's linkage-learning is reliable, even on problems where the linkages are badly scaled. An example found in Ferret's 'Demo' directory called 'Linkage-Learning/LinkageLearning-ScaledDeception' shows how Ferret's linkage-learning algorithm behaves on a very badly scaled problem with thirty parameters, where there are ten building blocks to discover, with three linked parameters each. These building blocks span ten orders of magnitude in their importance to the fitness function. Ferret is able to discover all ten building blocks reliably, and moreover, all building blocks are discovered at the same rate. This performance is outstanding in comparison to other linkage-learning algorithms, which usually can only find the most important few building blocks in badly scaled problems [Goldberg, 2002].

Ferret is the only global optimization package that I know of, which combines multi-objective optimization, critical parameter detection, and linkage learning. This makes it well-suited for many problems that are intractable using other techniques.

2.2.8 Strategy Auto-Adaptation

In computer science, the 'evolution strategy' (ES) algorithm is an alternate type of evolutionary optimization algorithm that differs significantly from a genetic algorithm. Traditional GAs use a binary representation of a problem's parameters, while ES uses a continuous parameter representation. GAs use a combination of mutation, crossover, and selection operators, while ES uses only mutation and selection, where mutation is usually defined as a Gaussian random perturbation of a solution's real-valued parameters. One advantage of ES over typical GAs is that the *mutation strength* is automatically optimized while the algorithm runs, by dynamically evolving the standard deviation of the mutation perturbations.

Ferret is not a traditional genetic algorithm by any means, and borrows heavily from ES. Like ES, Ferret's parameter representation is based on continuous real variables, and the mutation operator is a Gaussian random perturbation, whose strength evolves during the run. Moreover, Ferret's auto-adaptation strategy extends to several other strategy parameters as well. These include the following:

- Strength of the crossover operator, which controls the degree of mixing between two solutions during crossover.
- The crossover 'dispersion' parameter, which controls a mutation-like Gaussian perturbation applied during crossover operators, in order to better explore and fill in the optimal set.
- Ferret's 'mating restriction' parameter, which controls whether mates are preferentially selected from nearby or distant parts of the parameter space during crossover.
- A parameter that controls the strength of Ferret's 'Advanced Lethal Suppression' algorithm, discussed in Section 2.2.9.
- A parameter that controls Ferret's 'niching acceleration' system, which helps Ferret to infer 'good' directions to explore from the current spatial distribution of the population (Section 8.9.3).

With real-valued parameters, heavy borrowing from ES techniques, and all of these fancy techniques discussed above, I would not be all that surprised to hear comments from GA purists that Ferret is not *really* a GA. You won't hear any argument from me because this is absolutely true. However, I still refer to Ferret as a GA because the crossover operator plays a central role in this code, which is a defining feature of GAs. I take a pragmatic view on this topic and I don't *really* care whether Ferret is a true genetic algorithm or not, because the features that make its classification as a genetic algorithm questionable also help to make it so powerful as an optimizer. Ferret's development has always been driven by the scientific problems that I want to solve, and if the code is no longer a 'real' genetic algorithm as a result, then that is fine with me. I will therefore use the term GA rather loosely when talking about Ferret, since this code is clearly very different from classical GAs or what others may refer to a GA in the literature. These differences will be expounded upon fully within the remaining pages of this guide.

2.2.9 Advanced Lethal Suppression (ALS)

Lethals are poor solutions that are sometimes formed during the crossover of two high-quality parents. Ferret's Advanced Lethal Suppression (ALS) algorithm is a unique innovation that allows Ferret to learn what parts of the parameter space are likely to contain lethals, and to avoid these regions. This can improve the efficiency of the algorithm significantly, since fewer lethals are formed and fewer solutions are therefore wasted. This feature is not necessary for most problems, but it is worth trying if you suspect a problem with lethals. Usually, I do not turn it on explicitly, but I do allow it to evolve using the auto-adaptation feature discussed in Section 2.2.8.

2.2.10 Pausing, Stopping, and Resuming Runs

I live in Manitoba, and we experience violent lightning storms here that knock out the power - and my Ferret runs - a few times each summer. Many organizations also seem especially fond of 'planned power outages' that occur with a few days notice at apparently random intervals throughout the year. Most MATLAB users have experienced crashes now and then, and many of us know that system administrators often like to time network maintenance and computer reboots for the middle of two-week long calculations. All cynicism aside, my point is that unexpected things can and all too frequently do happen during calculations that last longer than a few days. I *hate* losing results when my Ferret run crashes, and fortunately I don't.

During a run, Ferret writes 'History' files every few generations, which contain a complete record of the best solutions encountered during the run at each generation. If a run crashes for any reason, it is simple to resume it with no (or very minimal) data loss. Likewise, you can suspend a run at any time and restart it later, even on a different machine. It is even possible to modify most strategy parameters between stopping and restarting a run.[3]

I place a lot of emphasis on the ability to pause, stop, and resume runs reliably. This is absolutely critical for long calculations, and this capability has been a robust feature of Ferret since the earliest version of the code. I have also ported these features to Locust, since it is also suitable for long-duration calculations.

[3]The user interface also contains a component that allows the user to modify strategy parameters on the fly, without stopping the run. See Section 7.2 for details.

2.2.11 Analysis and Integration of Polishers

At the end of a run, the user normally clicks the 'Analyze' button on the user interface, which causes the code to load and examine all of the saved History files, compare every solution saved with every other solution, and construct the final optimal set. This may contain a single solution for a simple single-objective problem. Alternatively, it may contain thousands of solutions spanning the optimal trade-off surface of a multi-objective problem, or a similarly large number of solutions distributed within the optimal region of a fuzzy single-objective problem, where a non-zero 'fuzzy' tolerance has been set.

Genetic algorithms are very good at finding global solutions, but these solutions are often not obtained to high precision. Ferret addresses this problem with several solution 'polishers' based on MATLAB's built-in fminsearch routine (for single-objective problems only), or the Qubist algorithms SAMOSA, Anvil or SemiGloSS which can be used for single or multi-objective problems. The combination of Ferret plus a solution polisher offers the best of both worlds: a powerful multi-objective global optimizer, with enhanced capabilities for obtaining final solutions to high-accuracy.

2.2.12 Integrated Visualization

My goal in developing Qubist is not only to find optimal solutions, but also to provide tools to help understand what they mean by exploring the mathematical and geometrical structure of the optimal set. The Qubist package contains fully integrated and intuitive visualization software that allows you to view any two or three-dimensional projection of the optimal set as a scatter plot, image or contour plot, though a sophisticated imaging interface that allows the user to tease interesting features out of the data set. You can also 'paint' regions of interest, view them from different orientations, save them to disk, and incorporate your own graphics right into the visualization interface. User-modifiable publication-quality figures are produced by the interface automatically by selecting the 'Take Snapshot' item from a menu. All user interface components are actually distributed as user-modifiable 'skins' - if you don't like how something in the interface works or want to add a component for your project, then you are welcome to do so.

Visualization of the optimal set is a cornerstone of the Qubist package and a current area of very active development. I plan to release an especially powerful new tool for exploring multi-objective optimal sets, tentatively called 'nQuest', as an update within a few months of this writing. The existing visualization tools are the topic of Chapter 5.

2.3 Locust

Locust is Qubist's multi-objective particle swarm optimizer (PSO). Like Ferret, this is a biologically inspired global optimizer, which searches a parameter space using a swarm of interacting particles. PSOs are often discussed in terms of the dynamics of flocks of birds, schools of fish, or swarms of social insects searching for food. The commonality is that intelligent behaviour is an emergent property of the system as a whole, even if the component parts are modeled as relatively simple automata that interact with each other through simple rules.[4] Particle swarm optimizers (PSOs) are a relatively new class of global

[4]Many readers would no doubt take exception to referring to birds, fish, or even social insects as 'simple automata'. That is not my intent, as it is absolutely clear that any of these creatures is capable of displaying complex behaviour on its own. The point is that complex individual behaviour is not absolutely required for complex social behaviour to emerge. Even automata obeying simple rules can result in complex, intelligent collective behaviour, as long as sufficient interactions exist that help to reinforce good solutions and explore nearby parts of the space preferentially.

optimizers that are a topic of intense research. Eberhart [2001] provides a good introduction to the PSO technique, along with an interesting discussion of how human intelligence derives from social interactions. The basic PSO technique is relatively simple, and there are many other excellent introductions on the web.

I began developing Locust in 2007 and it has developed rapidly because many of its internal functions borrow heavily from Ferret. Of all of the Qubist optimizers, Locust is the one that has undergone the most fundamental recent developments. I only developed the new Locust-2 code in the summer of 2009, as I wrote the bulk of this user's guide. Locust has surprised me because it is beginning to rival Ferret on tough global optimization problems. It does this without Ferret's linkage learning capabilities, critical parameter detection, strategy parameter auto-adaptation, or other fancy features. Locust has far, far fewer options than Ferret and is probably the simplest to configure of all of the Qubist optimizers. This is the most beautiful aspect of the PSO technique - particle swarm dynamics result in extremely powerful optimizers with relatively little messing around. In some ways, Locust is the opposite of Ferret. Ferret is a huge, pondering machine that crushes problems by analyzing their mathematical structure in detail, but Locust is a smaller, snappier code that does much less analysis internally, and yet has the uncanny ability to perform almost as well as Ferret on many problems.

I see Locust as absolutely complementary to Ferret, and it will be the target of considerable future development. However, these are still early days for this code - especially the new features in Locust-2. Ferret remains the most powerful optimizer in the Qubist package and by far the most thoroughly tested. You should definitely try Locust if you are interested in exploring alternatives, but I recommend Ferret for the toughest and most 'mission-critical' applications.

2.3.1 Locust's Features

Locust is unique as a PSO because it is based on an exact analytical solution to the equations governing the behaviour of the swarm, which improves the stability of the algorithm. 'Particle explosions' are a well-known instability that affects PSOs, in which errors accumulate and particles tend to wander off to infinity. An exact analytical solution helps to control particle solutions, and they do not occur in the Locust code.

As a multi-objective code, Locust is built with unique features that enable it to map the trade-off surfaces of multi-objective problems and the optimal solution set of fuzzy single-objective problems. It accomplishes this by means of a novel algorithm that divides the parameter space into 'neighbourhoods', which represent self-contained 'sub-swarms' that do not interact with particles outside of the neighbourhood. These neighbourhoods are dynamic; they move and possibly merge together as the code converges toward the optimal set. Mergers happen quickly for non-fuzzy problems with only one objective, and all of the neighbourhoods normally merge together so that all particles focus on the same optimal solution. Fewer mergers occur for parameter space-mapping problems, and the particle neighbourhoods will tend to spread out on the trade-off surface or within the optimal set. This was in fact the key innovation that makes the Locust-2 code so much more powerful than the previous Locust-1 code.

Locust has all of Ferret's features for pausing and resuming runs, as well as access to the same visualization system. Locust typically uses a different setup file than Ferret, but it can read Ferret's setup files and combine this information with default Locust settings. Moreover, the formats for the initialization, fitness, and custom graphics functions are identical. This makes it easy to swap optimizers for comparison purposes. Locust is discussed in Chapter 11.

2.4 SAMOSA

SAMOSA stands for 'Simple Approach to a Multi-Objective Simplex Algorithm'. This is a relatively simple, lightweight optimizer that is based on a modified Nelder-Mead simplex algorithm for non-linear optimization. Simplex algorithms sample the parameter space on the vertices of a 'simplex', which is a multi-dimensional generalization of a tetrahedron. The simplex expands, contracts, and deforms as it moves its vertices through parameter space, in a search that is often compared to the contortions of a wandering amoeba. Like the other Qubist optimizer, SAMOSA (and simplex algorithms in general) never evaluate gradients to aid the search, which helps to make them robust in the presence of local minima.

SAMOSA modifies the well-known simplex algorithm to allow for multi-objective optimization, which is unusual for this method. The most interesting feature of SAMOSA is that it can be run in two distinct modes. In its default mode, SAMOSA attempts to map the Pareto front by covering the optimal set with solutions, much like the other optimizers discussed above. However, when Ferret or Locust calls SAMOSA as a solution polisher, it toggles SAMOSA into a 'polishing mode', which targets and rapidly zeros in on the nearest region of the Pareto front and converges to a single point. This behaviour is ideal for a multi-objective polisher, because it can very rapidly refine an entire population of points to improve the resolution of the Pareto front.

SAMOSA was developed quite recently, in the spring of 2009. It is the newest optimizer in the Qubist package and the smallest, but it has already become the default polisher for multi-objective problems, where it truly excels. It can also be used as a standalone multi-objective optimizer, with the caveat that is not as 'global' as the other optimizers discussed above. MATLAB contains a single-objective simplex algorithm called 'fminsearch'. Users experienced with this code will be familiar with the properties of simplex algorithms. Chapter 9 discusses how to use SAMOSA as either a standalone optimizer or as a polisher for Ferret or Locust.

2.5 Anvil

Anvil is a multi-objective simulated annealing code in the Qubist toolbox, which is enhanced significantly by genetic algorithm techniques borrowed from Ferret. This code may be used as a reasonably powerful stand-alone optimizer, as an alternate multi-objective polisher to SAMOSA, or in place of MATLAB's built-in fminsearch routine, as a polisher for single-objective problems. Anvil can be called transparently from both Ferret and Locust.

Simulated annealing codes are inspired by the statistical physics of a slowly cooling metal. Intuitively, the atoms in a hot metal are in constant motion and large thermal fluctuations cause large variations in the energy of the system. As the metal cools, these fluctuations decrease, and the energy stabilizes to a constant value. If the system is cooled slowly, then the system should find its ground state energy, which is characterized by the lowest energy possible for the system. Thus, simulated annealing provides an excellent paradigm for global minimization.

Simulated annealing codes explore a parameter space stochastically by allowing the acceptance of 'bad' fluctuations that cause an increase in the energy function with relatively high probability early in the run. As the system cools, fluctuations that increase the energy are accepted less frequently, until late in the run, almost all of the allowed fluctuations decrease the energy. The occasional acceptance of fluctuations that increase energy is absolutely essential to the success of the technique, since this allows the code to hop over 'mountain ranges' in the energy landscape in order to find possibly deeper valleys on the other side. This is

especially important for exploration early in the run. Without the acceptance of fluctuations that increase the energy function, a simulated annealing code would perform something akin to a stochastic version of a steepest descent minimization algorithm, which would likely find itself stuck in the first local minimum encountered.

A simple simulated annealing code accepts a fluctuation ΔE with probability given by

$$P(\Delta E) = \begin{cases} 1 & \Delta E < 0 \\ e^{-\Delta E/T}, & \Delta E \geq 0. \end{cases} \tag{2.3}$$

Where T is analogous to the temperature of the system. Clearly, fluctuations that increase energy are accepted with high probability when $\Delta E \lesssim T$ and almost never accepted when $\Delta E \gg T$. Typically, the temperature starts off high during the run and gradually decreased according to a *cooling schedule* as the run progresses. This establishes a gradual transition from the initial exploration phase of the algorithm, when the search point jumps wildly through the parameter space, to the much more orderly and directed exploitation phase of the optimization process, when most of the accepted perturbations either decrease the energy or result in only very small increases. Various recipes exist for setting the cooling schedule, and these are discussed thoroughly in the literature, various textbooks, and many websites.

The arbitrariness introduced by the cooling schedule is a clear weakness of the simulated annealing technique, and both the quality of results and performance of the code depend strongly on how the system is cooled. Cooling too rapidly 'quenches' the system, and may result in the search becoming trapped in a local minimum. On the other hand, cooling the system too slowly makes the code very inefficient, because the search will settle too slowly into a minimum. Anvil remedies this situation using a *self-adaptive* cooling schedule, which dynamically sets the temperature to optimize the performance of the algorithm.

Most simulated annealing codes use a single search point that wanders through the parameter space until it settles down into a minimum on the energy surface. Anvil differs from a typical simulated annealing code because it uses multiple simultaneous search points that communicate with each other using crossover, mutation and selection operators borrowed from GAs. Moreover, Anvil also borrows heavily from Ferret's multi-objective optimization system and is capable of mapping out trade-off surfaces. The result is a code that is significantly more powerful and versatile than a typical simulated annealing code.

Anvil does not map solution sets as thoroughly as Ferret, but Anvil is a light-weight code with much less computational overhead. It is most suitable for problems of moderate difficulty, or problems where thorough mapping is not crucial. Anvil may also be a good choice for problems characterized by a fitness or energy function that can be evaluated rapidly, because if its lower overhead. However, if the problem is characterized by an optimization criterion that is computationally expensive, then Ferret would almost certainly be the better choice, since the extra computational overhead required by Ferret is likely to be insignificant compared to the computational cost of the fitness or energy function. Anvil is discussed in detail in Chapter 10.

2.6 SemiGloSS

SemiGloSS is an acronym for the Semi-Global Solution Spray optimizer, which is an experimental multi-objective solution polisher in the Qubist package. SemiGloSS moves from point to point through the parameter space by firing solutions into the space, and keeping track of directions and ranges that have previously been found to be successful. In this way, SemiGloSS learns how to move through the space in a way that increases its chances of finding an improved solution. It contains its own critical parameter detection system that is modeled on Ferret's CPD algorithm, which improves the efficiency of the code.

Although SemiGloSS is not a true global optimizer, it is quite fast and has significant global properties that allow it to hop over moderate sized 'mountain ranges' in the parameter space. Using SemiGloSS as a stand-alone optimizer may lead to accurate global solutions with fewer function evaluations than are required using the other Qubist global optimizers, as long as the problem is not too difficult. However, the price of this performance is that the space is not explored nearly as thoroughly, and SemiGloSS offers little assurance that the true global solution has been found. SemiGloSS should only be used as a standalone global optimizer for relatively easy problems that have already been well-characterized by Ferret, Locust, or Anvil. It is better suited as a polisher for solutions found by Ferret or Locust, since global optimization is not required for polishing.

It is important to realize that SemiGloSS is experimental, and rarely performs as well as Anvil as a standalone global optimizer, and never as well as Ferret or Locust. It is retained by the Qubist package in order to provide an alternative multi-objectiver polisher to Anvil and SAMOSA, and as a area for future growth of the package.

Like Locust, Anvil, and SAMOSA, SemiGloSS understands Ferret's setup files and fitness functions, and can be set as the solution polisher by changing a single line of the problem's setup file. Once this is done, SemiGloSS can be called from either Ferret or Locust by simple pressing the 'Polish' button on the analysis window used by Ferret and Locust to display final results. SemiGloSS remains undocumented in this user's guide due to it's experimental nature and the certainty that it will change substantially in future versions.

2.7 Other Tools

The Qubist toolbox contains several other tools to assist with the analysis of optimization runs and visualization of results. These tools include the following:

- Resume Tool: A tool that allows you to resume a stopped (or crashed) Ferret or Locust run from the History files that were saved to disk during the run.
- Analyze History: A standalone tool that allows you to analyze previously computed History files generated by Ferret or Locust. This also gives provides access to the same visualization interfaces that can be launched automatically from either Ferret or Locust during analysis.
- Merge OptimalSolutions: A simple tool that allows the user to merge OptimalSolutions.mat files generated by two or more Ferret or Locust runs.
- OptimalSolutions Viewer: A standalone tool that allows you to load the final solution set from a Ferret or Locust run - called OptimalSolutions.mat - into the Qubist visualization environment. The OptimalSolutions viewer can also be used for the following files generated by Qubist:
  - MergedSolutions.mat: A solutions file generated by merging two or more Ferret or Locust runs from the same problem.
  - PolishedSolutions.mat: A solutions file containing solutions generated by one of the polishers (SAMOSA, Anvil, or MATLAB's fminsearch code).
- Ferret History Explorer: A tool that loads the History files generated by Ferret and Locust to visualize the progress of a run.

- Qubist Movie Tool: A standalone tool to generate movies from previously completed Ferret runs or other saved images. Ferret is able to generate AVI movie files of a run's progress directly from its console window. However, the Movie Tool provides more flexible options to generate movies after the completion of a run, from either saved frames or from Ferret's History files, which contain information about Ferret's state after each generation. This tool can also be used to assemble static images generated from other sources into a movie file.

Note that the OptimalSolutions Viewer, Analyze History Tool, and Merge OptimalSolutions Tool are also distributed as a set of free Qubist tools to make it easier to share Qubist results between collaborators on a project.

2.8 Customizable Skins

The Qubist package works with Windows, MacIntosh, and Linux, with all versions of MATLAB newer than MATLAB 7.0. However the various combinations that are possible result in some variation in the look and feel of MATLAB interfaces. More importantly, user interfaces generated using newer versions of MATLAB are not always compatible with older versions and can crash the program. In order to address these problems, I have implemented a strategy based on swappable user interface components that can be loaded on demand. At present, Qubist contains the following skins:

- Windows: Most appropriate for newer versions of MATLAB running under Windows. This choice also works well with Linux.
- Linux: Most appropriate for newer versions of MATLAB running under Linux.
- Mac: Most appropriate for newer versions of MATLAB running on an Intel-based or Power PC MacIntosh computer.
- Minimalist: A set of user interface components generated using MATLAB 7.01 on a Windows computer. This interface is useful for older versions of MATLAB running under any of the supported operating systems. The interface contains the same functionality as the other skins, but the interface is somewhat stripped down. Users are cautioned that this skin is not pretty, but it is a good one to try if you use an old MATLAB version, or if you experience frequent crashes or Java errors, and suspect that your version of MATLAB is having problems with the interface.

I might add more skins to Qubist in the future, depending on demand. Additionally, all of the source code is available for the user interface components (though not all of the callbacks) so that you can customize it somewhat to suit your own needs. For example, you can use MATLAB's *guide* tool to move elements around on the interface, delete unneeded elements, and add custom elements required for your specific application.

Chapter 3

Installing Qubist

'If written directions alone would suffice, libraries wouldn't need to have the rest of the universities attached.'

-Judith Martin

Installing Qubist is as simple as copying the files on the installation disk to your computer, and following a few simple instructions. The following instructions are included with the distribution:

```
% ========================================================================
% Qubist: A Global Optimization, Modeling & Visualization Platform
%
% Ferret: A Multi-Objective Linkage-Learning Genetic Algorithm
% Locust: A Multi-Objective Particle Swarm Optimizer
% Anvil: A Multi-Objective Simulated Annealing/Genetic Algorithm Hybrid
% SAMOSA: Simple Approach to a Multi-Objective Simplex Agorithm
% SemiGloSS: An Experimental Semi-Global Optimizer & Polisher
%
% Copyright 2002-2010 Jason D. Fiege, Ph.D.  All rights reserved.
% Distributed by nQube Technical Computing Corporation under license.
% design.innovate.optimize @ www.nQube.ca
% ========================================================================
%
% ------------------------------------------------------------------------
% Qubist Installation Instructions:
%
% *** START ON STEP 1 IF INSTALLING FROM A CD. ***
% *** START ON STEP 4 IF INSTALLING FROM a Qubist.zip file. ***
%
% (Run this file in MATLAB to view it in MATLAB's doc viewer.)
% ------------------------------------------------------------------------
%
% *** INSTALLATION FROM A CD: ***
%
% 1. Identify which version of MATLAB you are running.  This can be done by
% reading the MATLAB splash screen as it starts, or by typing 'ver' at the MATLAB
% command prompt.
%
% 2. The Qubist installation disc contains 2 main directories called R2007a-
% and R2007b+, which contain 2 separate builds for MATLAB version 7.  If
% you have MATLAB version R2007a (MATLAB 7.4) or earlier, then choose the
% R2007a- directory.  If you have version R2007b (version 7.5) or later, then
% choose the R2007b+ directory.
%
% 3. Inside of the R2007a- or R2007b+ directory, there is a directory
% called
% 'Qubist'.  This is the Qubist home directory, which will be copied to your
% computer system in the next step.
%
% *** SKIP TO STEP 6. ***
%
% ------------------------------------------------------------------------
%
% *** INSTALLATION FROM A Qubist.zip FILE: ***
%
% 4. Copy the Qubist.zip file to a temporary location in your file system.
%
% 5. Unzip Qubist.zip.  This should generate a folder called 'Qubist'.  This is
% the Qubist home directory, which will be copied to your computer system in
% the next step.
%
% *** SKIP TO STEP 6. ***
%
% ------------------------------------------------------------------------
```

```
% *** CONTINUED FROM EITHER STEP 3 (CD INSTALLATION) OR STEP 5
% (INSTALLATION FROM Qubist.zip FILE.)
%
% 6. Optional step: Check your p-code distribution by running checkPCode.m
% on the MATLAB command line.  This program is located in the Qubist
% home directory and will generate an error like ‘checkPFiles.p not
% recognizable as a P-file.’ or ‘Corrupt P-file checkPFiles.p’ if the
% Qubist p-code files do not match your MATLAB version.  If you do not see
% any errors, then you have the correct p-code version, and you can
% proceed to the next step.  If not, you should try the other version on
% the CD, or contact nQube for assistance if installing from a Qubist.zip
% file.
%
% 7. Copy this directory and all contents to a convenient directory
% on your computer system.  You may choose any directory.  However, if
% multiple users will be using Qubist, you should put it in a location that
% all users can access.  (i.e. not your home directory or your 'My Documents'
% folder on Windows systems).  If only a single user will be using the
% code, however, the user's home directory can be used.
%
% 8. Make sure that the Qubist directory is writable during installation from
% the system administrator's account.  Installation will fail if files
% cannot be written to the Qubist directory.  In addition, the entire distribution
% must be readable by all users, and you should give full read/write permissions
% to the Qubist/demos directory for all users.
%
% 9. Each user should copy the Qubist/launchQubist.m file to his/her home
% directory or another convenient location.  The location of this file
% should be somewhere reasonably central, since this file will need to be
% invoked every time that Qubist is started.  Normally a user only has a
% single 'launchQubist.m' file, with all of his/her projects listed near
% the bottom.  It is therefore not advisable to put the launchQubist.m file
% in an directory that is specific to the user's project, and this can in
% fact interfere with Qubist's path.
%
% 10. The 'launchQubist.m' file contains the following line:
%
% QubistHome_='C:\Documents and Settings\JFiege\My Documents\Qubist';
%
% Change this line so that QubistHome_ is set to the actual location of the
% Qubist folder on your system.  A global path is recommended.
%
% 11. Run your launchQubist.m file.  If this results in an immediate error, then
% the line that you edited in the launchQubist file is incorrect.  Make sure
% that QubistHome contains the correct path to the 'Qubist' folder.  If you
% are still having problems, try running the function Qubist/startQubist.m.
% Either launchQubist.m or startQubist.m should trigger the license screen.
```

```
% 12. You should see a Qubist license (EULA) screen.  Read the license
% carefully, enter the license key that was provided by nQube, and certify
% that you have read and understand the license agreement by clicking the
% checkbox.  Click the button labelled 'I Accept the Agreement'.  The
% Qubist component that you selected should launch, and the license
% screen will not appear again unless you choose to re-install.  If the
% installation fails, pay close attention to the error message and any
% instructions that are offered to help solve the problem.
%
% 13. Running your 'launchQubist.m' file is the standard method for
% launching any of the Qubist components.  Ferret is by far the most
% powerful optimizer in the package and is the one that you are likely to
% use most often.  Some additional methods for launching Ferret are
% described in the file: Qubist/user/Ferret/doc/FERRET-STARTUP-OPTIONS.

doc('Installation_Instructions');
```

As the instructions indicate, there are separate sets of files for MATLAB versions up to and including MATLAB R2007a (MATLAB 7.4), and versions including and later than MATLAB R2007b (MATLAB 7.5). The reason for this complication is that most of the Qubist distribution is in the form of MATLAB p-files (i.e. myProgram.p), and the format for these files changed between the R2007a and R2007b distributions. You can find out your MATLAB version by typing 'ver' on the command line. Be sure to choose the correct p-file version if installing from a CD.

If you have received your copy of Qubist as a .zip file from Qubist, then nQube should have requested your MATLAB version information, and sent you the correct distribution with p-files that match your MATLAB distribution.

Even if you don't do the optional step #6, you will know if you use the wrong version, because you will see an error message like the following when you try to run the launchQubist.m file:

```
??? File "C:\Users\JFiege\Documents\Qubist\Qubist_Code\tools\installation\ ↘
        ↪ checkPFiles.p" not recognizable as a P-file.

Error in ==> launchQubist at 66
checkPFiles;
```

or

```
??? Corrupt P-file "C:\Users\JFiege\Documents\Qubist\checkPFiles.p".

Error in ==> launchQubist at 66
checkPFiles;
```

Of course, the file location will be different, and the line number of the error might not be the same, but any error that complains about your p-files when you try to run launchQubist.m probably indicates that your p-file version is wrong. You can verify this by following step #6 in the instructions. If you are using the wrong p-code version, then either try the other version from the CD if you have one, or contact nQube to correct the problem if you have a Qubist.zip file.

3.1 End User License Agreement

You can be assured that I dislike legal formalities more than most people, but you can't distribute software without an End User License Agreement. However, I have tried to make this as simple and as painless as possible. Please read it yourself, but the EULA mostly says things that amount to 'Please don't steal my software' and 'It's not my fault if Qubist doesn't work on your particular problem and terrible things happen as a result' - all written in sufficiently opaque legalese, of course.

The first time that you run the launchQubist.m program, you should be prompted by a window containing the Qubist End User License Agreement (EULA), as shown in Figure 3.1. This figure indicates that this is

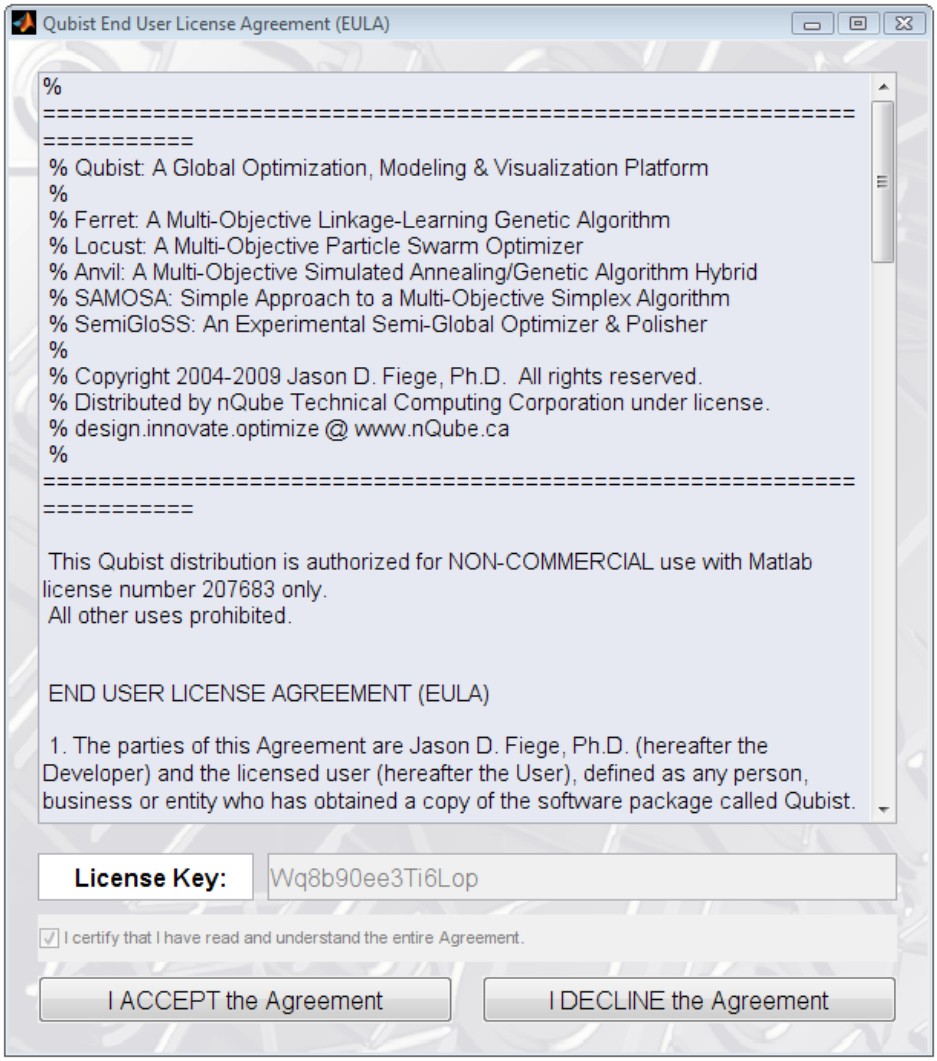

Figure 3.1: The Qubist End User License Agreement (EULA) screen. You must enter your license code and agree to the terms of the EULA in order to install the package.

a non-commercial license and that it was built for a student version of MATLAB. Your version may be commercial or non-commercial, and the associated numerical MATLAB license number will be listed here for non-student versions. The EULA can also be viewed with a text editor by reading the file [Qubist_Home]/LICENSE.txt, or from the 'Display Qubist License' selection in the menu labeled 'Other' in most of the Qubist user interface components.

If the EULA window does not come up, and you see an error dialog box like the one shown in Figure 3.2, then the path to the Qubist directory is probably incorrect in in your launchQubist.m file. Section 3.2 explains how to correctly set your Qubist path.

Figure 3.2: A Qubist launch error indicating that the launcher could not find the Qubist distribution. If you see this, then you probably have an incorrect path set in your launchQubist.m file. Section 3.2 explains how to set this path correctly.

You should have received a license key with your copy of Qubist, which you need to unlock the toolbox. You should enter your key in the appropriate textbox, certify that you understand the EULA agreement, and accept the agreement. The key unlocks Qubist for all users on your system, and will not be required again unless you re-install the package for some reason, or upgrade it with a new version from nQube. *However, you should definitely keep your key because I publish updates and bug fixes fairly often, and it is a good idea to keep this software as up to date as possible.* I test new versions thoroughly, both personally and with the help of collaborators on various projects, so anything I release should be safe to use and at least as bug-free as the previous version. You can always rename your old Qubist directory if installing a brand new version makes you nervous. The installation of an update will never delete or disable a previous version, unless you manually over-write the files during the installation.

It is important to note that each copy of Qubist is locked to a specific MATLAB license number and list of hardware addresses that you specified when you ordered your copy of the package. Qubist cannot be moved to other machines or run under a different MATLAB license, because every copy of the package is custom-built and this information is coded right into the p-files. If you try to do this, it will fail and you will probably see a fairly cryptic error message asking you to contact the developer.

If you need to move Qubist to a different machine or associate it with a different MATLAB license, you should contact nQube and update your information. A new copy will be built for you promptly and at no charge. Users with a current maintenance agreement with nQube will receive an update that works with their new hardware and/or MATLAB license. Users without a maintenance agreement may be updated to an earlier release that was current at the time that their maintenance agreement expired.

3.2 The Launcher File

In order to launch Ferret or any of the other Qubist components, you need a launch file, which is usually called 'launchQubist.m'. The purpose of this file is to initialize the Qubist toolbox, link your projects to Qubist so that they appear in the 'Projects' menu of the user interface, and to start up the package. A template launch file can be found in the [Qubist_Home] or the [Qubist_Home]/user/Qubist/scripts directory. Copy this file to some central location in your file system. A good choice is your home directory or your 'My Documents' folder on Windows computers, or your MATLAB start directory.

Do not put your launch file in the same directory as any of your projects, or attempt to start Qubist from inside of any of your project directories. If you do this and have multiple projects defined in your launch file, you may find that you are only able to run this one project, regardless of which project you select. This occurs because project selection works by manipulating the MATLAB path, but the current working directory is always at the top of the path. This problem is perhaps the most common 'gotcha' of new users, and occasionally even experienced ones. The resulting inability to load other projects is confusing, and I have received a number of false bug reports about this limitation.

You do not need a separate launch file for every project. The best practice is to have a single launch file, in a central location, with all of your projects listed in this file. All of your projects will then appear in the user interface, so that you can select between them. Some users really do seem to prefer having a different launch file for each project. Of course, this is also OK, but then only a single project will appear in the Qubist 'Projects' menu. Note that this is the only case where it is acceptable to put the launch file inside of a project directory.

Inside of the launcher file, you will find the following instructions near the top of the file to point the launcher to your copy of the Qubist distribution and set a few basic options:

```
% MODIFY THIS LINE TO POINT TO THE Qubist HOME DIRECTORY.
% This is normally a global path.  For example...
QubistHome_='C:\Users\JFiege\Documents\Qubist_Builder\Qubist';
%
% Relative paths also work.  For example:
QubistHome_='.';
%
% Use Qubist's compiled functions?  Uncomment one of these lines:
useMex_=true; % Use compiled functions when possible.
% useMex_=false; % Avoid compiled functions.
%
% --- Skins for graphical user interface ---
% Specify the desired skin, which affects the appearance of QUBIST.
% leave it empty or set it to 'default' in order to use the default that
% is appropriate for your operating system.  You can also design your
% own custom skin using MATLAB's 'guide' tool.
QubistSkin_='default';
%
% Uncomment one of the following lines to choose your skin if you do not
% like the default.  Note that all choices work on all operating systems,
% but may not look good.  Very old MATLAB 7.x versions may require the
% 'Minimalist' skin to avoid crashes.
% QubistSkin_='Windows';  % Colourful interface optimized for Windows computers.
% QubistSkin_='Mac';  % Colourful interface optimized for MacIntosh computers.
% QubistSkin_='Linux'  % Colourful interface optimized for Linux computers.
% QubistSkin_='Minimalist';  % Basic interface.  Best choice for old MATLAB 7.x versions.
```

Here, you see that the launcher sets the $QubistHome\_$ global variable to the home directory of my local Qubist installation, which resides in my 'Documents' folder on my Windows computer. Modify the path to point at your own local Qubist installation. Once you have done this, you should be able to launch Qubist by running your launcher file from the MATLAB main window. If you start the code by typing 'launchQubist' at the MATLAB prompt, please make sure that you are in the same directory as your launch file, or that it is on your MATLAB search path.

Notice that the launcher also specifies a global variable called $QubistSkin\_$. This functionality allows the user to select between several groups of interface components, or 'skins', that are designed for different operating systems or different versions of MATLAB. For example, the skin called 'Windows' is a nice skin that looks good under the Windows operating system using MATLAB R2007a and newer. It also looks quite good in Linux, but the skin called 'Linux' is a better choice. If you run Qubist on a MacIntosh computer, the skin called 'Mac' is the best choice.

If you try to use either the 'Windows' or 'Mac' skins with a very old version of MATLAB (i.e. MATLAB 7.0), then you may find that they crash the program. MATLAB is usually reasonably backward compatible with their functions, but in my experience, this does not always seem to be true of GUI components generated using their 'guide' interface builder. I built the stripped-down skin called 'Minimalist' using guide in MATLAB 7.01 for maximum compatibility with older versions. Trust me - it's not pretty, but it usually works.

If you set $QubistSkin\_ =' default'$, or $QubistSkin\_ =''$, Qubist will make a reasonable default choice based on your operating system. Note that it will not select the 'Minimalist' skin though, even if your MATLAB

version is old; this must be selected manually if you require it. I have tested the skins quite thoroughly, using a variety of MATLAB distributions and platforms, but it is not feasible to test every possible combination. If you cannot find a skin that works with your particular setup, please let me know, and I will try to resolve the problem if at all possible.

Note that all of the GUI components are located in the [Qubist_Home]/user/[*component*]/skins directories, complete with source code. You are welcome to modify these files using guide to customize them to suit your needs.

Running the launchQubist.m file starts the Qubist 'Component Selector' window, as shown in Figure 3.3. All Qubist components can be started from this interface by selecting a radio button and clicking the 'Launch' button. Ferret is the default selection because it is the best optimizer in the package for most problems and the most commonly selected Qubist component.

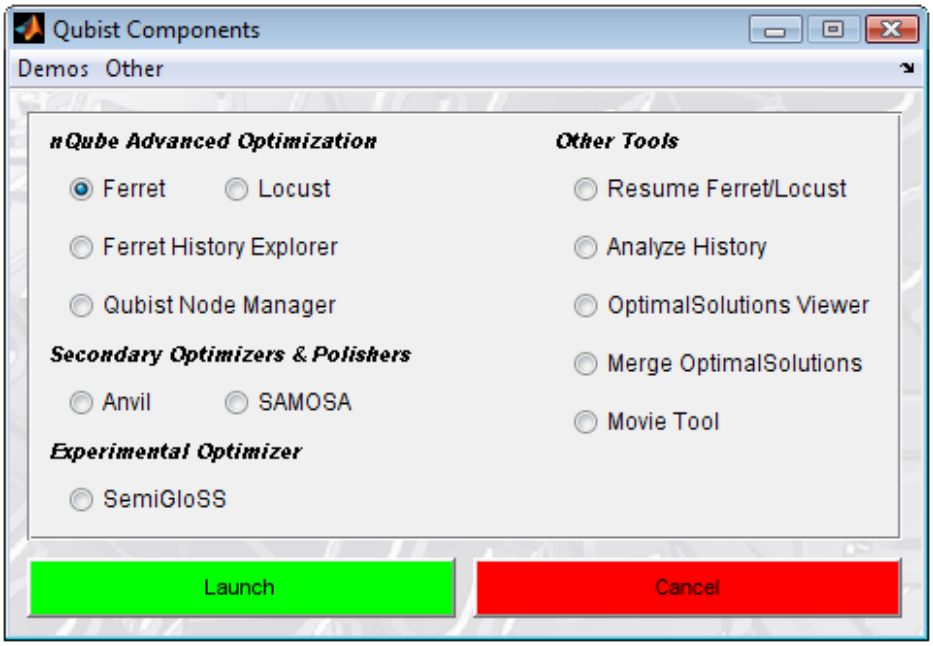

Figure 3.3: The Qubist Component Selector window. This interface can be started by running launchQubist.m, which loads user-defined projects, or startQubist.m, which only loads demos.

Chapter 4

Running Ferret

'The theory of evolution by cumulative natural selection is the only theory we know of that is in principle capable of explaining the existence of organized complexity'

-Richard Dawkins, *The Blind Watchmaker*

This chapter covers the essential information that you need to start running Ferret, the advanced genetic algorithm at the heart of the Qubist package. As discussed in Chapter 2, Ferret is the optimizer of choice within the current Qubist version for difficult real-world or research-level problem. You may have noticed that the chapters on Ferret occupy the bulk of this user's guide. This is partly because Ferret is the most flexible and complex tool in the package, but also because the other optimizers are very similar in their usage. Once you have mastered Ferret, you should have an excellent grasp of the entire Qubist package.

4.1 Overview

The basic structure of Ferret-based application is illustrated in Figure 4.1. Ferret should not be thought of as a traditional optimizer that is called from user-defined code, but rather as an integrated environment for optimization, data-modeling, and visualization. Ferret normally serves as the graphical (or command-line) front end for each project that is defined in the launchQubist.m file (see Section 4.4), which is used to start Ferret. When using the graphical user interface (which I encourage you to do for most applications), Ferret runs the project's initialization code when a project is selected from the interface. You can then start your project from the interface, and Ferret will call the project's fitness, output, and graphics functions, which are defined in a project-specific setup file (see Chapter 8), as required.

Ferret likes to be in charge of managing projects, and is not normally called *from* the user's project, although this *is* possible by setting up a custom launcher, as discussed later in this section. This technique of having the optimizer as the front-end and calling out to user-defined functions may be a slight paradigm shift for some users, but this mechanism has evolved over a number of years, and is extremely convenient and flexible to use. Most users like it once they get used this way of doing things.

The following steps are required to develop a Ferret project and run it:

1. Write any initialization code, usually called the project's 'init' function, which loads data files, sets paths, and performs any preliminary calculations that need only be done only once at the start of a run, and *not* for each parameter set separately.
2. Write a fitness function that can accept parameter sets and the *extPar* structure, evaluate the parameter sets, and return fitness values to tell Ferret how good each solution is.
3. Optionally, write the following:
   - An 'output' function to perform side-calculations or implement a project-specific stopping criterion.
   - A 'myPlot' function to generate user-defined plots in the Ferret interface.
   - Any post-processing code that you want to run on the optimal solutions returned by Ferret.
4. Connect the project to Ferret by editing a few lines in your launchQubist.m file (see Section 4.4).
5. Run launchQubist, select your project from the 'Projects' menu, and click the 'Start' button!

Using Ferret as a project manager means that you do not need to worry about the intricate details of the genetic algorithm, or how all of your project-specific files need to be linked together. Ferret *knows* when it needs to call your initialization, fitness, output, graphics, and post-processing code, and it provides a nice interface to your code. As a developer, letting Ferret look after these details saves you time, which you can spend developing the model evaluated by your fitness function. This division of responsibility is absolutely fundamental to the design of Ferret and the other Qubist optimizers. The user is responsible for making sure that the required project-specific functions do what they need to do and return sensible values, and Ferret is completely responsible for linking the project together, managing the optimization, and visualizing results. Ferret and the user-defined code operate at arm's length and are connected only through a few lines in the launchQubist.m file. A running Ferret project *looks* like it is completely integrated into the Ferret environment, but in reality, Ferret and the user's code remain completely independent, with no blending or intermingling of code whatsoever. Figure 4.1 is a simple flowchart that shows how Ferret communicates with the various functions of a project.

The user-defined functions that appear in Figure 4.1 are discussed later in this chapter, but a few preliminary comments are worthwhile. Ferret communicates with the following functions while it is running:

- The init function is initialization code that is called only once when a project is loaded, and before it is actually running. Anything that needs to be done only once (loading data, setting paths, and preliminary calculations) should be done here. The init function generates a data structure called *extPar*, which stands for 'external parameters' (external to Ferret, that is), which is available to your fitness function when it is called. The init function is discussed in Section 4.6.
- A setup file, which is usually called 'FerretSetup', is called after initialization. This function tells Ferret about your problem and configures the genetic algorithm. Setup files can be very short and simple if you wish to use mostly default settings, but they can become fairly long for complicated projects, or when advanced users want to exert a lot of control over the optimization process. The setup file is discussed in Section 4.7, and Chapter 8 covers every possible option in detail.
- Ferret passes parameter sets to the user's fitness function for evaluation, and expects to receive fitness values back from the fitness function. As Ferret cycles from one generation to the next, it

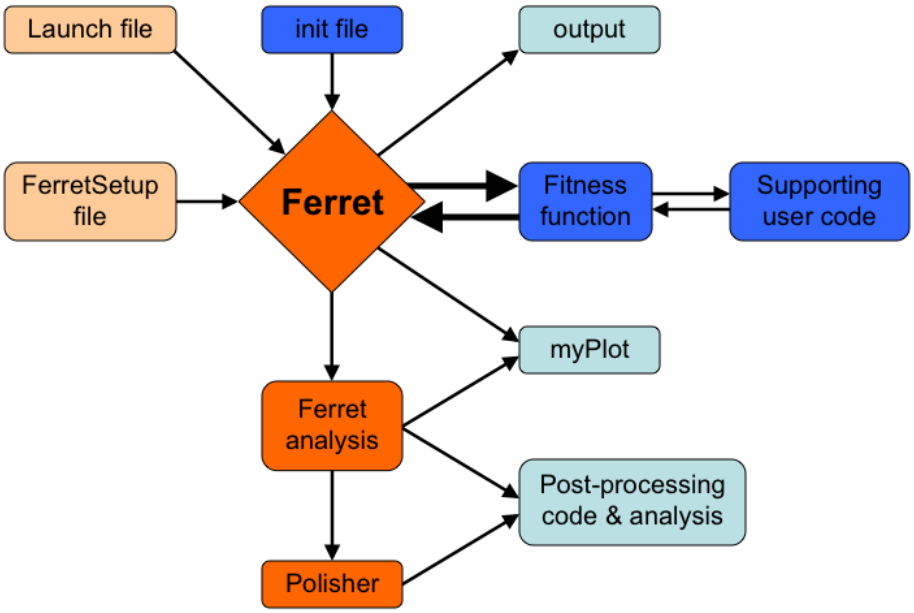

Figure 4.1: Schematic flowchart of a Ferret-based application. Bright orange boxes correspond to major Ferret components and pale orange boxes correspond to Ferret configuration files. Dark blue boxes correspond to essential code that must be developed by the user for his/her project, while light blue boxes represent optional functions. Arrows show the direction that information flows. The heavy black arrows represent the two-way communication between Ferret and the project's fitness function.

gradually learns which parts of parameter space contain the best solutions, as defined by the user's fitness function. The population gradually converges to this optimal region and either maps out its structure or converges to a single best solution, depending on the problem and how Ferret is configured. The fitness function is called by Ferret several times per generation. The number of parameters is always the same, but Ferret will request the evaluation of a different number of solutions each call. The user's fitness function is often only the top-level interface to an elaborate modeling code, which calls other supporting MATLAB functions, or programs written in another language accessed through a MATLAB external interface (See MATLAB documentation for details.).

- An optional output function is called exactly once per generation, and receives the entire *world* - Ferret's top-level internal data structure, which contains a set of populations and other data. Each population contains 'individuals', which are defined by their parameter sets and fitness values. The output function is discussed in Section 4.9, which shows how you can use the output function to do side calculations, monitor for convergence, or stop a run. Normally, the output function does not send back any return arguments. However, a mode is available that allows you to directly modify your *extPar* data structure, or even the top-level *world* structure and send it back to Ferret. This is a special feature for advanced users and requires extreme caution, because sending back an inconsistent *world* could crash Ferret. This capability is sometimes useful for very large optimization

problems where you want to start with a simple function defined by only a few parameters, and gradually refine the function by adding parameters as Ferret converges to the solution. A good example is given by the demo titled 'Advanced/Data-Modeler-modWorld', where terms are successively added to a Fourier series to model some artificial data. The details of the *world* data structure and output function are covered in Section 4.9.1.

- An optional 'myPlot' function is called exactly once per generation. Any graphics commands evaluated in myPlot will appear inside the Ferret console. This is useful for monitoring the progress of a run, since you can show a visual representation of your current best solution, or anything else that you wish to plot. See Section 4.10 for details.

Ferret (and Locust) require an analysis step at the end of a run to compile results, which is discussed in detail in Section 4.15. Analysis loads all of the best solutions encountered during the run and compares all of the solutions with each other by an iterative algorithm to construct the final optimal set. The analysis window launches, which gives you access to Ferret's visualization system, as discussed in Chapter 5. Analysis also generates an OptimalSolutions.mat file, which can be loaded into MATLAB, or re-loaded into the visualization environment using the Qubist 'View OptimalSolutions' tool. *This tool is integrated with the standard Qubist package, but is also available from nQube as a free set of visualization tools.*

You may attempt to refine your solutions by pressing the 'Polish' button in the analysis window. This starts one of several local optimizers, and causes Ferret to write a *PolishedSolutions* data structure to disk. Finally, you may develop other custom analysis code that loads the *OptimalSolutions* data structure and does whatever post-processing analysis or cleanup operations that the your application requires. Ferret can call this code automatically at the end of the analysis process, or you can load your OptimalSolutions.mat into MATLAB and call your post-processing code manually.[1]

The most important concept illustrated in Figure 4.1 is the clear separation between Ferret and the user's application code. Ferret communicates with the application only by sending parameters to the application's fitness function and receiving fitness values. Ferret does not need to know *anything* about the application other than the number of parameters, the range of each parameter, and the names of the fitness function, init function, and other optional functions that it needs to communicate with. Ferret will explore your parameter space thoroughly, minimize your fitness function, and map it out as completely as possible. It does this by learning the detailed structure of the parameter space, purely from experience gained during the run. Conversely, Ferret can be considered a 'black box' optimizer, in the sense that your application doesn't need to know anything about how it is being optimized - it just needs to send back fitness values when asked to evaluate parameters. A consequence of this clean separation is that Ferret and the user's application code are completely separate on disk, and Ferret's EULA ensures that Ferret and the user's application remain distinct and entirely separate software products. They maintain an arm's length relationship, and talk only through a very thin communication layer. This in turn has two consequences:

1. Every Ferret-based project is highly modular. It is often preferable, especially for large projects, to develop the user part of the application independently of Ferret. When the application is working, it can be linked to Ferret with little effort.

2. Relatively little information is exchanged between Ferret and the application code, which makes Ferret ideal for parallel computing. Ferret is packaged with an easy to use 'node manager' tool, which allows you to start a parallel run and manage worker nodes from a convenient user interface. *No code*

[1] Locust also requires an analysis step and can be configured to call post-processing code at the end of the analysis step. The other Qubist optimizers call post-processing code at the end of a run.

modifications are required whatsoever[2]*. You literally write your fitness function for a single CPU, select the desired number of worker nodes from a drop-down menu, click a button to start 'worker nodes', and your Ferret project runs automatically in parallel mode!* The parallel computing system is entirely self-contained in Qubist. You do not need to purchase the parallel computing add-on toolbox from the Mathworks, or install any of the other free parallel computing codes for MATLAB that are available on the web.

My advice for developing Ferret projects is to start with the fitness function, and whatever modeling code it needs to run in order to evaluate parameter sets and return fitness values. Once you are reasonably confident that this is working, you should split off any initialization code that produces the same result regardless of the parameter values, and move it to the init function (see Section 4.6). In other words, any operation that only needs to be done once but produces a result required by the fitness function should be placed in the init function. These initialization operations usually involve setting paths required by the fitness function or myPlot function, loading data, pre-processing data, or other preliminary calculations. The init function should return a single structure, typically called *extPar*, which will be passed to the fitness function each time it is called.

Next, you should get your fitness function working with init, so that the following lines of code work:

```
extPar=init;
F=fitness(X,extPar);
```

where X is a matrix defined such that each column represents a parameter set, and the number of columns represents the number of parameter sets to be evaluated (see Section 4.8). For multi-objective problems, the return argument F should be a matrix defined such that each column represents the fitness values computed from the corresponding column in X. The number of rows in F should therefore equal the number of objectives. It follows that F should be a row vector for single objective problems. You should literally make a 'fake' X with the correct number of parameters and a few columns, with all parameters in the range of the search, and run this independently of Ferret. This allows you to fix errors before connecting your code to Ferret, which makes things simpler for debugging.

You should copy a template of the FerretSetup file into your project directory and start modifying it to better suit your application. At this stage, you should concentrate on the most basic settings in the `par.general` section of the setup file (see Section 8.3), specifically those that tell Ferret about the allowed range of each parameter, its name, and the name of each objective. This is already enough to link your project to Ferret and start running it, following the procedure in Section 4.4.

You should do a very short test run to make sure that that Ferret starts and your project loads, but please be aware that Ferret will be using default settings (in [Qubist_Home]/user/Ferret/defaults/defaultFerretSetup.m; see Section 4.7 for details) at this point for all but the most critical parameters. These may not be very suitable for your project, so don't expect fantastic results on this test run. Setup files are discussed in Section 4.7, and Chapter 8 covers all possible options in detail. Setup files can be complicated, but you should try not to get too bogged down in the details. Try to use as many of the default settings as possible during the early stages of a project by commenting out or deleting non-crucial lines that don't absolutely need to be modified, and gradually refine your setup as the project evolves and you gain experience with it.

Finally, you can write an output function to implement a customized stopping criterion (see Section 4.9.2), a myPlot function to embed customized graphics in the Ferret window if you wish (see Section 4.10), and

[2] One minor change is useful to improve the efficiency of parallel runs, but is not required. See Section 6.4.4 for a discussion of the `isAbortEval` signal.

any necessary post-processing code (see Section 4.19). The overall thrust of my advice is that you should take advantage of Ferret's modular design to develop your project one piece at a time. Chances are good that you have chosen to use sophisticated optimization software because your problem is complicated. You should therefore not expect to build your entire solution at once and just run it when you're done. It is much easier, and I think better, to put your project together a piece at a time and check that things are working as expected at each step.

This concludes my brief overview of Ferret, and hopefully explains why it is designed to operate primarily as the front-end for its projects. All of the other Qubist optimizers are also designed to work like this, and in fact communicate with the same project files that Ferret uses. Most users find this to be an easy, powerful, and productive way of doing things, and I hope that you share this experience. However, it is possible to circumvent my preferred method of using Ferret as a project's front end by building a custom launch file, as discussed in Section 4.14.2. Custom launchers can contain any user-defined code and call Ferret as a regular MATLAB function. I do not personally find this useful, except for problems that require batch-processing - automatically running Ferret many times on different data sets or with different configurations. However, I have encountered users who really prefer this method because they like the fine degree of control that it allows, or regard graphical user interfaces as wasteful of computational resources. Ferret comes with a custom launcher template, which makes it relatively easy to set these up, so give it a try if you prefer this over the standard way of doing things!

4.2 The Ferret Console Window

The most common way to run Ferret is through the graphical user interface (GUI), as shown in Figure 4.2. The GUI consists of several graphs that monitor the progress of a run, as well as menus and buttons that allow you to interact with the program. The figure shows Ferret solving one of the problems included with Qubist in its demo directory, which can be accessed through the Demo menu. A convenient Project Notes/Info panel provides information about the run and associated files, and allows the user to view the setup, initialization, and fitness functions, which will be discussed later in this chapter. Each demo includes a Notes.txt file, which can be viewed from the Project/Notes panel by clicking the button labelled 'Notes'. These files are read-only (from the Ferret window) for demos, but can be edited directly from the Notes/Info panel for user-defined projects.

4.2.1 Full Graphics vs. Minimal Graphics Mode

Note that Ferret can also be run in a minimal graphics mode by clicking on the button labeled 'Turn Graphics Off' in the lower right corner of the console window. This reduces the graphics-heavy console window to a simple interface like the one shown in Figure 4.3, which eliminates the slight computational overhead required to render the console window graphs each generation. The user can toggle back to the full console window by clicking the button labeled 'Turn Graphics On'.

In practice, I find the full graphical display quite useful since it gives me an idea of how my runs are progressing, and whether or not anything has gone wrong. Once you get used to the console figures, you will find that they give significant insight into your problem and how to maximize the efficiency of your runs by tweaking Ferret's setup file. For most real-world problems, the CPU time is completely dominated by evaluating the fitness function, and the time spent running the graphical display is not significant.

If you *really* want to run without graphics, just close the window while you are in minimal graphics mode

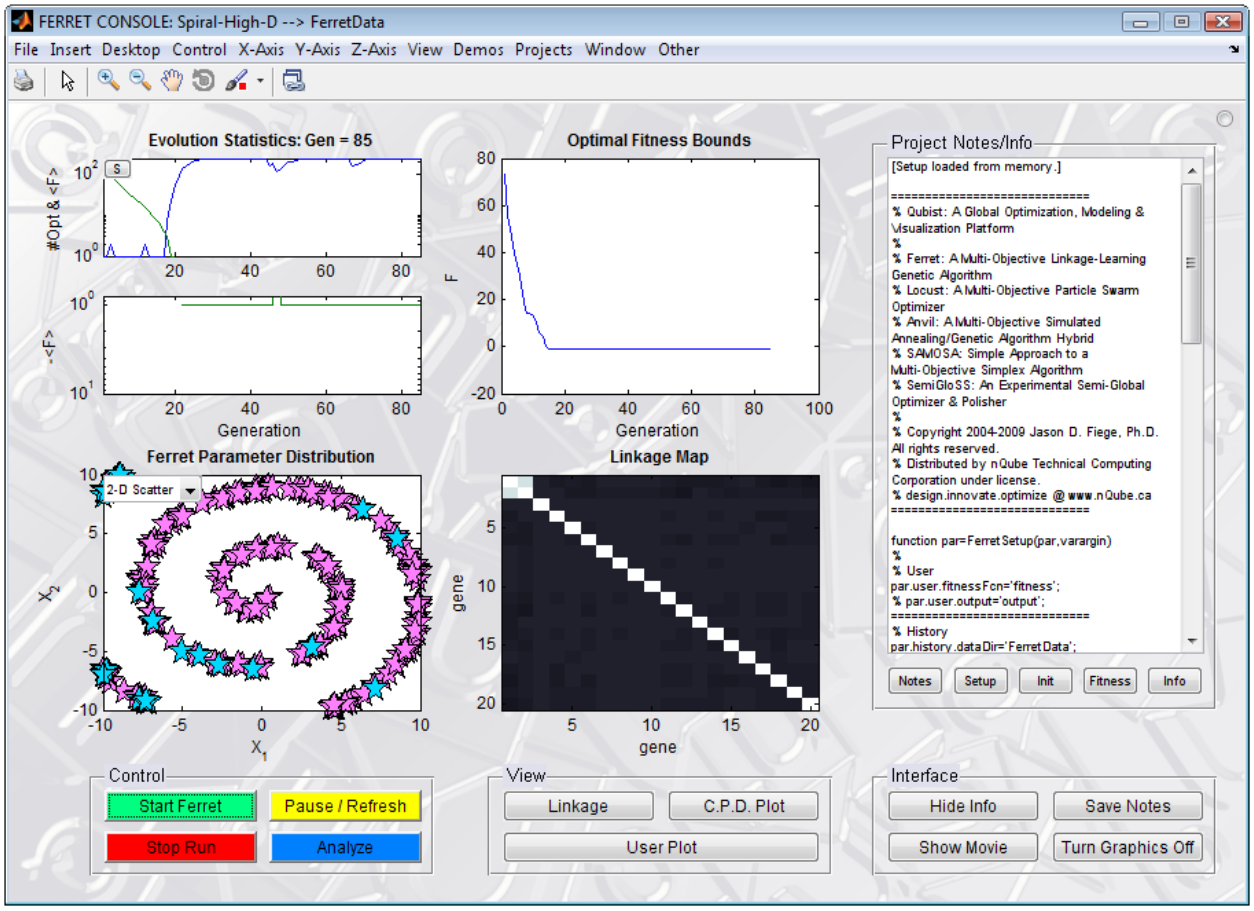

Figure 4.2: The Ferret console window solving a problem from the Demo menu.

and it won't come back. You will still be able to access Qubist's visualization software at the end of your run when you analyze your results. It is also possible to build a custom launch script that starts Ferret as a command line program, without ever starting the console window. This is sometimes useful for batch processing, and the technique for doing this is discussed in Section 4.14.2.

4.3 Running Demos

The Ferret console window contains a Demo menu, which is used to select any demo that is to be run. All of the demos listed are located in subdirectories of the Qubist demo directory: [Qubist_Home]/demos. Each demo is designed to illustrate a particular feature or set of features, and each is accompanied by a 'Notes.txt' file that explains what is being demonstrated. The Notes.txt file is automatically loaded into the Project Info/Notes pane when the demo is run, and every m-file in the demo directory is human-readable. It is a good idea for new users to try some of these demos while reading this guide to understand how the initialization, fitness files, output, and myPlot functions are implemented, as well as

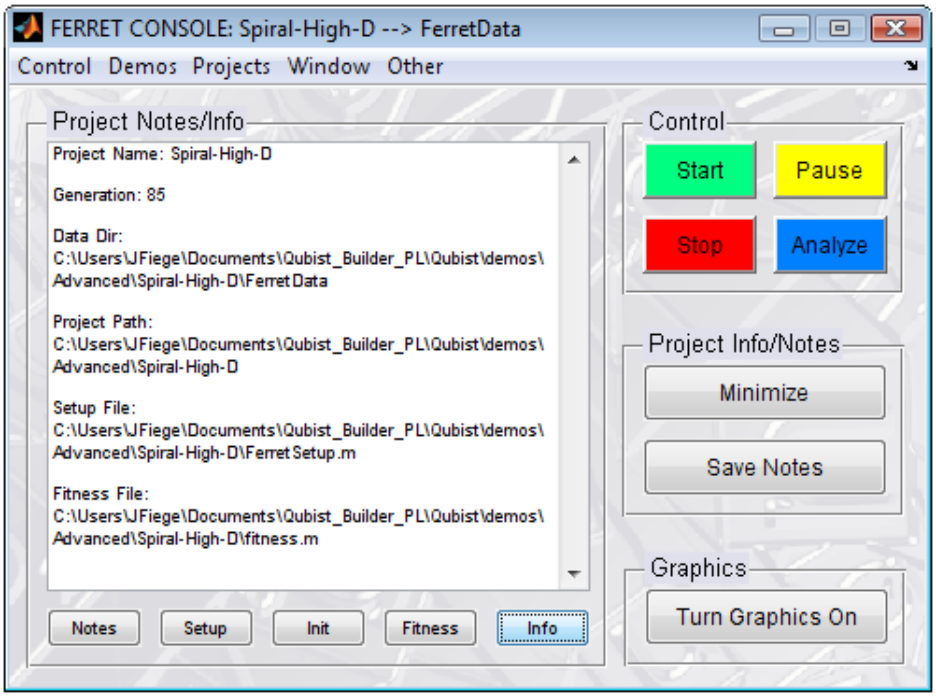

Figure 4.3: The Ferret console window running in minimal graphics mode.

the setup options in the FerretSetup.m file.

It is possible to add your own demos simply by adding a new folder to the demo directory because any folder in this location is assumed to be a demo. However, this is not the preferred method to add actual projects. Adding projects is not much more difficult, but it does offer additional flexibility, as discussed in Section 4.4. In practice, the demo directory is very useful when setting up a new project because it is usually possible to find a demo that is similar in spirit to the actual problem, so that the demo initialization, setup and fitness files can be used as templates.

Demos can be run by the following steps:

- Select the desired demo from the Demo menu. You may see a dialog box like the one shown in Figure 4.4, which asks whether to load the project's 'init file' (see Section 4.6). Click 'Continue'.
- Press the 'Start Ferret' button, or select 'Start Ferret' from the Control menu. You may see a warning like the one shown in Figure 4.5. This is just a warning that MATLAB sometimes has rather poor control over window focus because user interface components run in the same thread as a project's computation. The solution is to include short pauses in the project's fitness function, if the user interface seems sluggish. Occasional calls to `pause(0.001)` are sufficient because this forces MATLAB to check the event queue. For now, just click OK. Note that there is a little box on this window that you can click if you don't want to see this message again.
- While Ferret is running, the 'X-Axis', 'Y-Axis', and 'Z-Axis' menus can be used to control the view shown in the bottom left graphics axes. Note that any parameter or fitness value can be plotted

against any other parameter or fitness value on these axes, using various types of two-dimensional plots (see Chapter 5). three-dimensional plots are also possible, but these take longer to render, and I personally find them less useful. There is a listbox control on the axes that can be used to control the type of graph shown. The listbox blocks part of the figure and can be hidden by selecting '*** HIDE ME***', which is the first selection in the listbox control. To get the listbox back, the user needs to pause the run using the 'Pause' button or the corresponding entry in the Control menu, and then un-pause the run.

The bottom right graphics panel also has several possible views. The user can choose to view the linkage matrix, the critical parameter detection (CPD) plot, or a user-defined plot. The linkage matrix and CPD plots are explained in Section 4.12. Note that all of the demos implement a user plot by setting an option in the FerretSetup.m fill called `par.interface.myPlot`, to the name of a valid m-file on the path (Section 8.20). A myPlot function defined in this way contains graphics commands that plot on the bottom right axis, whenever the 'User Plot' view is selected. I find that this is quite a useful feature for customizing project and monitoring the progress of runs. The myPlot function is discussed in Section 4.10.

- Allow the demo to run for the desired number of generations and then press the 'Stop Run' button, or select 'Stop Ferret Run (Soft Kill)' from the Control menu. This method of stopping a run allows Ferret to finish its current generation and stop cleanly, saving all necessary information that would be required to resume the run If the user wishes to do so. The '*Kill* Ferret Run (Hard Kill)' option, available only through the Control menu, stops the run more quickly, but skips most of the steps required for a clean stop that is resumable without any information loss. It is still possible to resume a run (Section 4.17), but be warned that some information may have been lost and it will probably take Ferret several generations to get back to a state comparable to where it was killed.

  Pressing 'Control-C' in the MATLAB window will stop the run immediately, but Ferret will generate error messages and no cleanup operations will be done whatsoever. The run can still be resumed, but it will revert to the last generation saved in the History files located in the FerretData directory. Typically, it takes a crashed or Control-C'd run longer to recover than one that has been stopped cleanly or killed using the interface. Note that sending a Control-C will not normally harm the graphical user interface. The interface will remain active and can still be used to view results or launch other demos or projects.

- A dialog box like the one shown in Figure 4.6 may pop up asking if you wish to analyze your run, unless you have added the line `par.analysis.analyzeWhenDone=true` to the demo's setup file, or previously checked the box in the dialogue labeled 'Do not show this dialog again'. You can analyze your run by clicking 'Yes' in this dialogue box. You can also analyze at any time - even while Ferret is running - by clicking the 'Analyze' button in the console window, or by selecting 'Analyze History' from the Control menu.

  Analysis is the essential step that produces your final results - the OptimalSolutions.mat file, and gives you access to Ferret's visualization interface, which is the topic of Chapter 5. The details of Ferret's Analysis procedure are covered in Section 4.15, but I will briefly explain here what this does. While Ferret runs, it generates 'History' files, containing all of the best solutions encountered during each generation. Analysis causes Ferret to re-load all of the saved History files and compare all previously encountered solutions with each other in order to construct the final optimal set for the run. Analysis constructs a MATLAB data structure called *OptimalSolutions*, which is saved to disk as OptimalSolutions.mat, and holds the complete set of final results from your run. It can be loaded into MATLAB by clicking on the file in the MATLAB interface, or by using MATLAB's 'load'

command. The *OptimalSolutions* structure contains a great deal of information about the run, and is the main results file that users interact with after a run is completed. Its internal organization is explained in Section 4.18.

Analysis also brings up the Ferret analysis window, which offers two and three-dimensional scatter plots of the run's results, image plots, contour plots, linkage and CPD (Critical Parameter Detection) information, user-defined plots, 'snapshots' for producing publication-quality graphs, and many other visualization features discussed in Chapter 5. The parameter/objective space view of the analysis window is shown in Figure 4.7, and other views are shown in a detailed example later in this user's guide (see Figure 5.2).

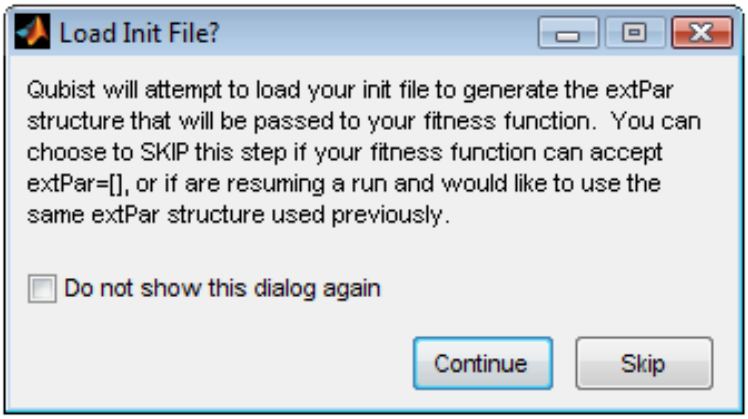

Figure 4.4: Dialog box prompting the user to load a project or demo's initialization file.

4.4 Adding User-Defined Projects

It is very easy to add a user-defined project to the 'Project' menu, simply by editing a few lines of text. *Note that these projects will be visible to all Qubist optimizers, and should run without modification.* I have occasionally thought about developing a more GUI-based approach to adding projects for Qubist, but I don't think there would be much point because Qubist users all write code and edit files anyway. The current file-based approach is very simple and utterly reliable.

4.4.1 Local Projects

A Qubist local project is one that is defined inside of a user's own launchQubist.m file, so that each user of the Qubist package may have a different set of local projects. It is also possible to define global projects that are accessible to all users, as discussed in Section 4.4.2.

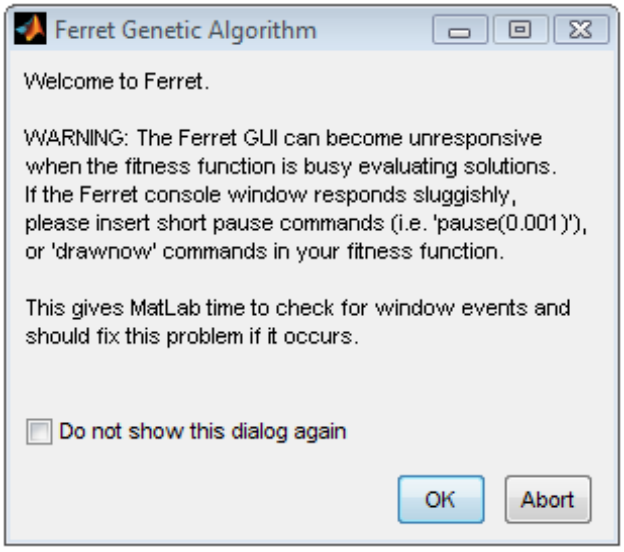

Figure 4.5: Dialog box warning the MATLAB has somewhat poor control over window focus because user interface components run in the same thread as the computation. The solution is to include occasional short pauses in the fitness function.

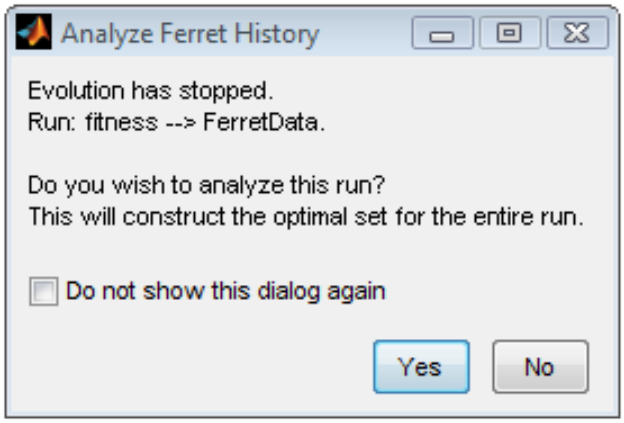

Figure 4.6: Dialog box asking whether of not a demo or project should be analyzed. This is displayed when a run stops if the setup file does not contain the line `par.analysis.analyzeWhenDone=true`, and the user has not previously checked the box in the dialogue that asks MATLAB not to show it again.

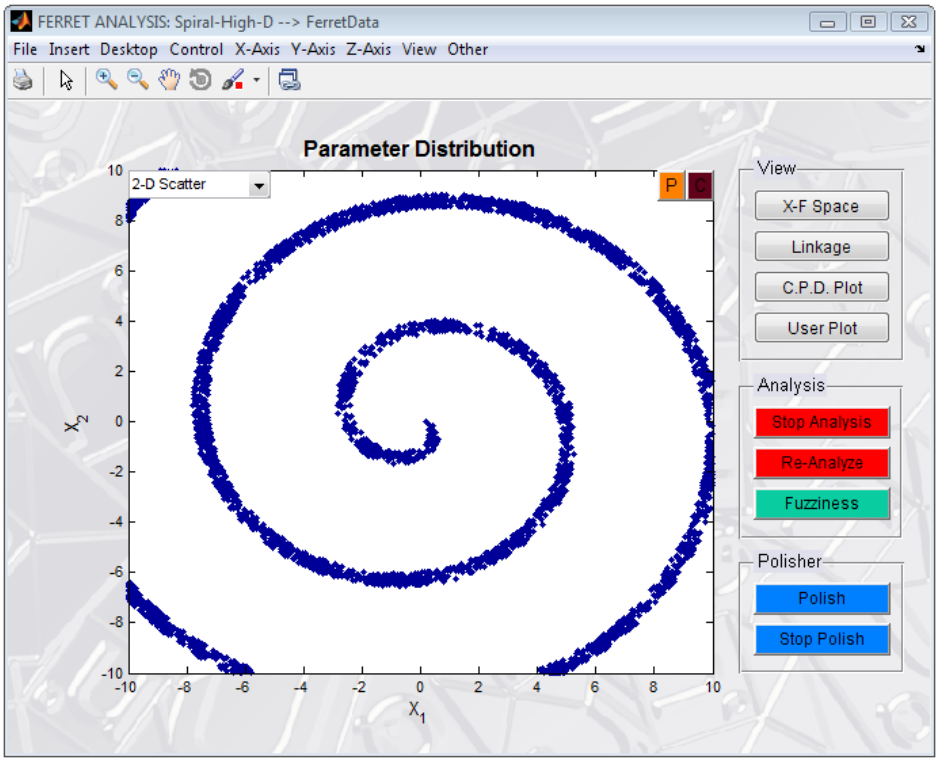

Figure 4.7: Ferret's Analysis window, showing the parameter/objective space view that can be generated by clicking the 'X-F Space' button in the 'View' button group, in the upper right hand side of the window. Three other views are possible, as discussed in an example later in this user's guide (see Figure 5.2).

To add a local project, scroll down to near the bottom of the launchQubist.m file, and look for the following instructions:

```
function projects=Qubist_UserProjects

% To add your own LOCAL (per user) projects to the Projects menu:
%
% Edit this file and add the path info for your projects in the following
% format:
%
% projects{n}.path='/full/path/to/project/directory'; %[default: ./]
% projects{n}.name='Project Name'; %[default: deepest directory in projects{n}.path]
% projects{n}.init='initializationCode'; %[default: none]
% projects{n}.setup='FerretSetupFile'; [default: 'FerretSetup']
%
% Here, n is an integer giving the number of the entry in the projects
% menu.  Start at n=1 for the 1st entry, n=2 for the 2nd, etc.
% Path should be the full directory path to the project, which contains
% the setup file and the fitness m-file.  Name is the arbitrary name of
% the project.  Init is an optional initilization m-file that produces
% an external parameter structure that is passed to Ferret.  Setup is
% the name of the Ferret setup file.
%
projects={};
%
% ========================================================================
% MODIFY THESE LINES TO ADD LOCAL PROJECTS VISIBLE ONLY TO THE CURRENT USER:

projects{1}.path='C:\Documents and Settings\JFiege\My Documents\myQubistProject';
projects{1}.name='Sample Qubist Project';
projects{1}.init='init';
projects{1}.setup='setup';
```

To add your first project, simple follow along and change the four lines near the bottom to reflect your project files. To add a second project, add similar lines for projects{2}, and so on for projects 3, 4 ,5, etc.:

```
projects{2}.path=...
    'C:\Documents and Settings\Jason D. Fiege\My Documents\anotherSampleFerretProject';
projects{2}.name='Another Sample Ferret Project';
projects{2}.init='init';
projects{2}.setup='setup';
```

Note the line 'MODIFY THESE LINES TO ADD LOCAL PROJECTS VISIBLE ONLY TO THE CURRENT USER'. Each user should have his or her own launchQubist.m file, located in some central location like their home directory. Projects defined here will appear only for the owner of the launchQubist.m file.

4.4.2 Globally Accessible Projects

In addition to local projects, you can also add *global* projects that are available to all users of your Qubist distribution. This is done by editing the file [Qubist_Home]/user/projects/QubistGlobalProjects.m, in which you will find the following lines:

```
% ========================================================================
% MODIFY THESE LINES TO ADD *GLOBAL* PROJECTS VISIBLE TO *ALL* USERS:
%
% projects{1}.path='/Users/fiege/Documents/myGlobalProject';
% projects{1}.name='Sample Global Qubist Project';
% projects{1}.init='init';
% projects{1}.setup='FerretSetup';
%
% ========================================================================
```

The format for adding globally accessible projects is identical to the format for adding local projects. The only difference is that the QubistGlobalProjects.m file is visible to *all* users because it is located inside of the Qubist home directory, while each user can define local projects in his or her own launchQubist.m file.

4.4.3 The Project Definition Fields

Each project added requires a data structure called `project{`n`}`, where n is a unique number assigned to the project. So what do the fields of `project{`n`}` represent? The `project{`n`}.name` line is the name of the project, as you want it to appear in the Projects menu. The `project{`n`}.path` line is obvious; this is just where your project can be found on your file system. *However, it is critically important to understand that Ferret will load this path* only *for your project, but not any sub-directories.* The `project{`n`}.init` and `project{`n`}.setup` fields are respectively the names of your init and setup files, which are discussed below. If you need subdirectories to be loaded, then this must be done by putting 'addpath' lines in your init file, or another m-file called by your init file, as discussed in Section 4.6.

4.5 Stopping Your Run - The Simple Approach

For many runs, it is often best to set the maximum number of generations (`par.general.NGen`) to an arbitrarily high number and just run Ferret until the graphs on the console indicate sufficient convergence. This is especially true of problems with multiple solutions (nearly all multi-objective problems, and some single-objective problems), because these problems require an entire region of the parameter space to be mapped. Such problems often arrive at the optimal region quite rapidly, but the optimization should continue for as many generations as possible (within reason) in order to map the optimal part of parameter space in detail. For such problems, the optimal set gets progressively more detailed, and runs are limited by how long the user is willing to wait, the detail required in the optimal set, and how nice you want your figures to look.

Stopping a run manually is more qualitative than many people like, but it is fine for problems that focus on parameter space mapping, because the results don't *really* change once the optimal region has been found - they just get progressively more detailed. I often compare these types of problems to 'integrating photons' when observing an astronomical object with a telescope. The longer you wait, the more detailed

and less noisy the image becomes, and the decision to stop observing is based in part on your image requirements, and in part by how long you are willing to wait. For this type of problem, just do an analysis now and then to check the quality of the solution set, and stop the run when you see fit. At this point, just click the 'Stop Ferret' button on the interface, wait for Ferret to stop, and do an analysis to obtain your final results. Analysis is discussed in Section 4.15.

Ferret can also be stopped by two other automated methods:

- You can specify stopping tolerances on the parameters and/or the fitness values, which will automatically terminate your run when your criteria are met. This is done by setting `par.stopping.XTol` and/or `par.stopping.FTol` in the Ferret setup file, as discussed in Section 8.15. Note, however, that this method is not usually useful for parameter space mapping problems, because they are not intended to converge to a single solution, but rather to some extended region of parameter space.
- You can also implement a custom stopping criterion using the output function, which gives you a very fine degree of control over how your run terminates. See Section 4.9.2 for details.

4.6 The init Function

The 'init' field in the user projects section of the launcher file contains the name of an initialization m-file for your project. If this field is found, then Ferret will call this m-file *once* during the initialization of your project. This is where you should load any additional paths required for the project, *including sub-directories of the project directory*[3], and any code that only needs to be called once during the run. A good example is if your calculation requires static data to be loaded from disk, which might also require some pre-processing. Loading and processing data are often time-consuming operations and should not be done in the fitness file, since this is called multiple times each generation. It is better to load the data set in an init.m file, and pass the data to the fitness file though Ferrets external parameter mechanism. This works as follows:

1. The name of the init file is arbitrary, and is given by the character string defined by `projects{n}.init`, where n is the project number. The customary name (and the name of the init file for all demos that require one) is 'init.m', but any name can be used. The init file must reside in the project directory.
2. The init function accepts no arguments and creates a MATLAB structure that is customarily called 'extPar', which stands for 'external parameters'. The parameters are considered external because they belong to the project, and are therefore external to Ferret, while Ferret's internal parameters are called 'par'. The *extPar* structure is passed to Ferret by one of two methods, depending on whether you use the standard technique, or a less commonly used one that relies on global variables.

4.6.1 The Standard Technique

The usual method for defining an init file is to write a function with the following signature:

```
function extPar=init
```

[3] I often find it useful to load extra paths from a separate m-file that is *called* by the init file. This works just as well.

The *extPar* structure is passed back to Ferret when init is called, and the entire *extPar* structure will be available to your fitness function when it is evaluated.

4.6.2 Advanced Technique: *extPar_* as a Global Variable

The standard method passes *extPar* to your fitness function as a local variable, as discussed in Section 4.6.1. This is the preferred method, and works well for most problems. However, there is also an advanced technique to make the data from your init file accessible as a global variable called *extPar_*. I developed this method for a specific problem, which called SAMOSA to handle an internal optimization of a sub-problem *inside* of a Ferret fitness function. It is possible to handle this problem using the standard local variable technique, but this comes at a heavy price. The problem is that Ferret and SAMOSA each have their own *par* structure, and *extPar* is carried inside *par* as `par.user.extPar`. This means that a problem like this requires *two* copies of *extPar*: Ferret's `par.user.extPar` and SAMOSA's `par.user.extPar`, and they can't be shared. This can be expensive in terms of memory if *extPar* contains a lot of data, as it often does. This situation is much better handled by storing the data from the init file in a global variable called *extPar_* that can be accessed by both Ferret and SAMOSA (or Anvil), and excluding it from `par.user.extPar`.

Let me offer this advice on the global *extPar_* technique. Please don't use this as your standard technique for sending *extPar_* to your fitness function. Global variables are considered to be bad form by most programmers, and can lead to confusion when debugging. The standard method based on passing *extPar_* as a local variable is the established technique, and is known to be robust. I believe that the global *extPar_* technique is reliable, but it has not yet been tested nearly as thoroughly as the standard method. It should be reserved for weird situations when you *really* need it.

I have included an example of this technique in the demo titled 'Advanced/Ferret-SAMOSA-Anvil Insanity', which is designed for Ferret but also works with Locust. The name reflects the confusion that can result when embedding one optimization inside of another if you are not careful about it (and sometimes even if you are). The following instructions are slightly adapted from the Notes.txt file in the demo directory: [Qubist_Home]/demos/Advanced/Ferret-SAMOSA-Anvil-Insanity/Notes.txt.

1. Write a single init file that contains everything. You should define *extPar_* as a global variable, and you do not need to return *extPar* from the init function:

```
function init
global extPar_
```

2. Write separate fitness functions for Ferret and for the internal optimization problem. Give these good names to avoid confusion. The demo uses 'fitnessFerret' and 'fitnessInternal' respectively for the Ferret fitness function and the fitness function for the internal problem, which is optimized by either SAMOSA or Anvil.

3. Create a separate FerretSetup and a setup file for the internal optimization: either SAMOSA_setup to use SAMOSA, or AnvilSetup to use Anvil. The FerretSetup file must call call the SAMOSA_setup or AnvilSetup file by the following steps:

```
% @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
% @  EXTRA SETUP INFO FOR SAMOSA AND Anvil
% @ All extra setup info MUST go into par.user if you want
% @ this demo to also work with Locust.  par.user is copied
% @ into LocustSetup verbatim by the translation program
% @ translateFerretToLocust.  Other user-defined fields are
% @ not copied.
% @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
%
% !!!!! ADD SETUP INFO FOR SAMOSA. !!!!!
% This info is required for the SAMOSA call that is embedded in Ferret's
% fitness function.
%
% Load an INCOMPLETE default setup file from
% Qubist/user/Ferret/defaults/defaultSAMOSA_setup.
par.user.SAMOSA_par=defaultSAMOSA_setup;
%
% Add remaining fields by translating Ferret --> SAMOSA.
par.user.SAMOSA_par=translateFerretToSAMOSA(par, par.user.SAMOSA_par);
%
% The translation program will use Ferret's setup file (specifically the
% par.general.min and max fields) to determine par.general.X0, and this
% will be wrong for SAMOSA.  Destroy the X0 field and let SAMOSA sort it
% out for itself.
par.user.SAMOSA_par.general.X0=[];
%
% Add modified fields from SAMOSA_setup.
par.user.SAMOSA_par=SAMOSA_setup(par.user.SAMOSA_par);
%
% ====================================
% !!!!! ADD SETUP INFO FOR Anvil. !!!!!
% This info is required for the Anvil call that is embedded in Ferret's
% fitness function.
%
% Load an INCOMPLETE default setup file from
% Qubist/user/Ferret/defaults/defaultAnvilSetup.
par.user.AnvilPar=defaultAnvilSetup;
%
% Add remaining fields by translating Ferret --> Anvil.
par.user.AnvilPar=translateFerretToAnvil(par, par.user.AnvilPar);
%
% Add modified fields from AnvilSetup.
par.user.AnvilPar=AnvilSetup(par.user.AnvilPar);
%
% ====================================
```

Of course, you would not normally require both SAMOSA and Anvil for the inner optimization problem. You can just comment out either the `par.user.SAMOSA_setup` or the `par.user.AnvilPar` fields, if you only ant to set the problem up for one or the other.

4. Remember to turn off graphics in the SAMOSA_setup and AnvilSetup files (`par.interface.graphics=-1`). Also remember to give the names of the two fitness functions:
   - In FerretSetup: `par.user.fitnessFcn='fitnessFerret'`
   - In SAMOSA_setup: `par.user.fitnessFcn='fitnessSAMOSA'`
5. The *par* structure for SAMOSA and Anvil will be available from within Ferret's fitness function as `extPar.QubistPar.user.SAMOSA_par` and `extPar.QubistPar.user.AnvilPar` respectively.
6. From within the Ferret fitness function, you can call SAMOSA or Anvil by:
   - `OptimalSolutionsInternal=SAMOSA(extPar.QubistPar.user.SAMOSA_par)`
   - `OptimalSolutionsInternal=Anvil(extPar.QubistPar.user.AnvilPar)`
7. Extract a single parameter set or fitness value by doing the following:
   - `X_SAMOSA=OptimalSolutionsInternal.X(:,1)`
   - `F_SAMOSA=OptimalSolutionsInternal.F(:,1)`
8. Use these values from within the Ferret fitness function as you wish!

4.6.3 Advanced Technique: Specifying Members of the Initial Population by Hand

Every once in a while, you might encounter a problem where you have *a priori* knowledge of good solutions, based on physical grounds or previous runs. In these cases, it is sometimes useful to specify some or all of the initial population by hand, rather than allowing Ferret to populate the parameter space randomly, within the bounds specified by `par.general.min` and `par.general.max`, which is the default behaviour. To specify initial parameters by hand, you simply include them as a matrix `extPar.X0` returned your init file. For example, consider the following init function for a problem with for parameters:

```
function extPar=init
X1=[0.1; 0.1; 0.1; 0.1];
X2=[0.5; 0.5; 0.5; 0.5];
X3=[0.9; 0.9; 0.9; 0.9];
extPar.X0=[X1, X2, X3];
```

In this case, the user specifies three solutions $X1$, $X2$, and $X3$, and joins them as columns of a matrix `extPar.X0`, which is returned by the function. These three parameter sets are guaranteed to be present in the initial population, and if they are indeed good solutions, they should propagate throughout the population and give your run a head start. A single parameter set can be specified as a column vector. Note that it is not necessary to evaluate the fitness values of `extPar.X0`, since Ferret will do this automatically during initialization. The matrix `extPar.X0` will be truncated if the number of columns exceeds the population size.

4.6.4 Minimizing Disk Usage

The structure returned by the init function (*extPar*) is passed to the user's fitness function every time it is evaluated. For many problems, *extPar* can be quite a large, complicated structure, with many branches. Nevertheless, it is almost always more efficient to put loaded data in *extPar* and pass it to the fitness function, rather than re-loading the data with each call to the fitness function. The function signature of the fitness function is given in the section on fitness functions.

The location of the History file directory is determined by the `par.history.dataDir` field of the setup file, but the default location is [projectDir]/FerretData/History, where [projectDir] is the project directory specified in the launch file. History files contain cell arrays of MATLAB structures that can be loaded and explored at the MATLAB prompt, as discussed in Section 4.16.4.

It is important to realize that, by default, a copy of *extPar* is saved with *every* generation in the History files that Ferret generates while it is running. These files can become quite large if *extPar* contains a lot of data, and this can cause problems on systems without much memory, when the History files are later reloaded and compared to construct the optimal set (see Section 4.15). Fortunately, there is a mechanism to limit the amount of data saved by telling Ferret to save only one copy of selected data-rich branches of the *extPar* structure, in the first generation of the first History file. Later generations and other History files will not contain the selected fields and will be much smaller. This is done by including a field called `extPar.noSave` in the *extPar* structure, and setting it to '1' or 'true', as follows:

```
function extPar=init
%
% Physical Constants.
extPar.K.c=299792458; % Speed of light in m/s
extPar.K.G=6.67300e-11 % Gravitational constant in SI units [m^3 kg^-1 s^-2]
extPar.K.hbar=1.05457148e-34; % Heisenberg's constant in SI units [m^2 kg s^-1]
%
% Data.
extPar.data=load('hugeDataFile');
extPar.data.otherData=load('moreData');
extPar.data.noSave=true;
```

Note that 'noSave' must be specified *on the same level* in the structure as the data field that is not to be saved with every generation. In this case, Ferret will not save multiple copies of 'data' or the 'otherData' field below it, but every generation will be saved with a copy of the physical constants in `extPar.K`. In practice, this is a reasonable way of doing things because it is convenient for post-run analysis to have quick access to small structures like physical constants, etc. in each saved generation, but saving multiple copies of data or other large structures is wasteful of disk space. If you are a minimalist and don't want multiple copies of *any* of the *extPar* fields to be saved, except for in the first generation of the first History file, just include the noSave directive on the top level of the *extPar* structure defined in your init file:

```
function extPar=init
...
extPar.noSave=true;
```

4.7 The Setup File

The projects{n}.setup line of the launch file (see Section 4.4) gives the name of the setup file that contains all of the control parameters for Ferret. If this field is absent from the launch file, then it defaults to the name 'FerretSetup.m'. The setup file must be located in the specified project directory. There are templates for the FerretSetup file in the directory [Qubist_Home]/user/Ferret/templates, and examples in each of the demo directories. Setup files can become quite complex because Ferret can be configured with many options, as seen in the templates. In practice, they are usually much shorter and simpler than the templates because the user does not have to specify every field. Any field takes on a default value from the 'defaultFerretSetup.m' file if it is omitted from the setup file, and these defaults are usually sensible for most projects. It is a good practice to make your setup files as simple as possible by including only the lines that you really need to modify from the defaults, and just leaving the rest out. When I write a setup file, I normally just copy one from a demo that I think is similar to my real problem and modify it. This can save considerable time.

The defaultFerretSetup.m file is located in the [Qubist_Home]/user/Ferret/defaults directory and can be modified by users. This should only be done by advanced users, however, since modifying the default values could have far-reaching consequences for all of your projects, and anyone else using the same Qubist distribution. Generally, it is a much better idea to change fields in your project-specific setup file. The setup file is discussed at length in Chapter 8.

It is possible to view your setup file from either the console or analysis windows by selecting 'View FerretSetup' in the menu labeled 'Other'. If you use a Windows or MacIntosh computer, this will pop up a window like the one shown on the left of Figure 4.8. This also works on Linux machines, as shown on the right side of the figure, but the FerretSetup window isn't nearly as useful or pretty.

4.8 The Fitness Function

At some very simplified level, Ferret could be described as a code for choosing new sets of parameters intelligently, based on the performance of previously evaluated parameter sets. Of course, the key word here is 'intelligently'. The success or failure of a parameter search and optimization problem hinges on how 'intelligent' the algorithm really is at learning the structure of the parameter space, so that it can choose parameters that are close to the global minimum, while ignoring uninteresting regions of the space. Moreover, it must be able to do this reliably, without any knowledge about the internal workings of the fitness function.

Ferret does not know or care what your parameters actually represent. It only matters that some parameter sets are better than others, as judged by your fitness function. As Ferret searches through the parameter space, regions that contain good models become well-explored, while other regions containing inferior models are quickly ignored. The most important thing to remember is that Ferret maintains an 'arm's length' relationship with your problem, and the division of responsibility between Ferret and your application is absolutely clear. It is your application's responsibility to give Ferret an appropriate range to search for each parameter, to be able to evaluate the function or model that each parameter set represents, and to tell Ferret quantitatively how good each model was. It is Ferret's responsibility to keep track of where the best models are located in the parameter space and to choose the next parameter sets carefully to maximize the chance of finding even better solutions.

Every demo and user-defined project must contain a fitness function that evaluates sets of parameters

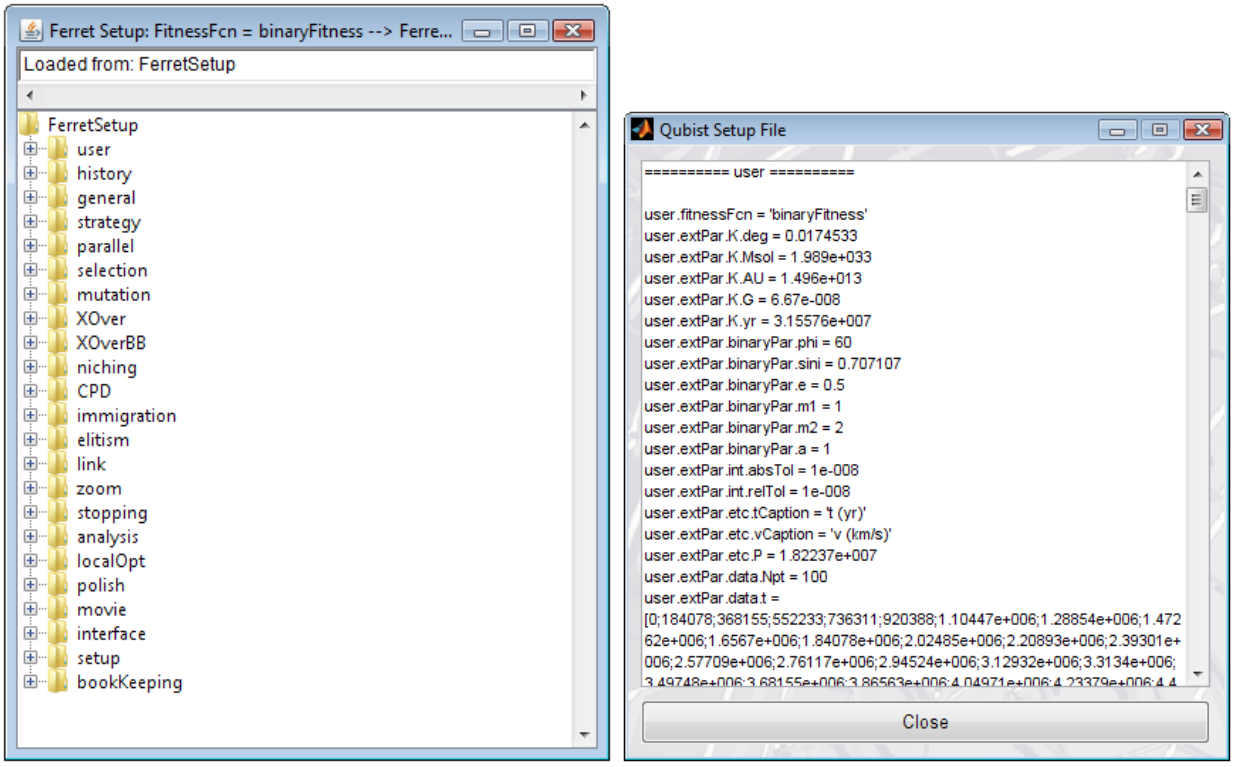

Figure 4.8: Two views of the Ferret setup file. These figures can be viewed and explored by selecting 'View FerretSetup' in the menu labeled 'Other' from either the console or analysis window. The left panel shows the setup file as viewed from Windows or MacIntosh computers, and the panel on the right shows the corresponding figure on Linux computers.

requested by Ferret from the parameter ranges specified by the `par.general.min` and `par.general.max` fields of the setup file. The basic form of the fitness function is as follows:

```
function F=fitness(X, extPar)
% A simple fitness function.
% X --> Array of parameter values received from Ferret.
% extPar --> structure made by the init.m file.
%
[NGenes, N]=size(X);
% NGenes=number of parameters.
% N=number of parameter sets requested.
%
for model=N:-1:1
        % Loop over the number of parameter sets.  It is a good
        % idea to loop backwards so that the array F is initialized
        % to the right size the first time through the loop.
        %
        % Grab a single column --> one set of parameters.
        X1=X(:, model);
        %
        % Evaluate the fitness of this parameter set and save to F(model).
        F(model)=evaluateFitness(X1, extPar);
end
```

Here, *extPar* represents the external parameter structure generated by the init file (see Section 4.6), and X is a two-dimensional array of size $NGenes \times N$, where $NGenes$ is the number of parameters and N is the number of solutions requested by Ferret. Note that Ferret uses the words 'gene' and 'parameter' interchangeably. This is different from a traditional genetic algorithm, where a 'gene' might represent a single 0 or 1 on a string of binary bits, and a 'parameter' might be defined by many such genes. Ferret is different because the binary bit strings are completely absent; all of Ferret's genes are real numbers, and its genetic operators are defined over real vector spaces.

Each column of X represents a single set of parameters that the fitness function must evaluate. Usually, the fitness function contains a loop over the number of columns, so that each column can be evaluated as a set of parameters, unless the fitness function is simple enough to vectorize over the entire X array. This is done by many of the simple demos, but is unlikely for a real problem. It is important to note that the number of rows $NGenes$ in X is guaranteed to always be the same, since this represents the number of parameters in your problem. *However, the number of columns N represents the number of solutions currently required by Ferret and varies unpredictably from one evaluation to the next.* You must therefore always design fitness functions that are flexible enough to evaluate an arbitrary number of solutions and return the corresponding number of fitness values.

The fitness function must return a matrix F, which contains the fitness values. Each column of matrix F contains the fitness values for the corresponding column of X, and F is therefore of size $NObj \times N$, where $NObj$ is the number of objectives being simultaneously optimized. For multi-objective problems, the order of the rows does not matter, but it must be consistent from one evaluation to the next. **Note that Ferret is a minimizer. Good solutions correspond to low values of F, which is opposite to the usual convention in the genetic algorithm literature.**

F is a mandatory output field, but the fitness function may also return three other optional fields. The full signature of the fitness function is as follows:

```
[F, {auxOutput}, {saveData}, {XPhysMod}]=fitness(X, {extPar})
%
% *** Braces {} indicate optional fields. ***
%
% Input fields:
% X          -->   [NGenes x N matrix]
% extPar     -->   MATLAB structure
%
% Output fields:
% F          -->   [NObj x N matrix] (required)
% auxOutput  -->   {N-element MATLAB cell array} or {empty cell array} (optional)
% saveData   -->   {N-element MATLAB cell array} or {empty cell array} (optional)
% XPhysMod   -->   [NGenes x N matrix] (optional)
```

The optional output fields are discussed below. Of these, the *auxOutput* field is the most useful and most commonly used. The *saveData* field is used much less often, and $XPhysMod$ is only used under very uncommon circumstances.

4.8.1 auxOutput - the Auxiliary Output Cell Array & Application to Penalty Functions

The *auxOutput* output argument is a $1 \times N$ cell array, which is a good place to put the results of any useful intermediate calculations or side-calculations that you might want to examine at the end of the run. It is important to note that anything that you include in the *auxOutput* cell array will be saved inside of your History files, and will ultimately appear in your OptimalSolutions.mat file for the best solutions that are evaluated. Therefore, it is a good idea to limit the amount of data that you include in *auxOutput*. If you record large amounts of data in *auxOutput*, your History files will be very large, and loading them for analysis will be slow and require a lot of memory.

The auxOutput array is carried around with solutions, just like fitness values. Any data that you put in *auxOutput* will also be added to your OptimalSolutions.mat file at the end of the run, for those solutions that are included in *OptimalSolutions*. I often use *auxOutput* on data modeling problems that require a penalty function to avoid regions that violate a constraint. In that case, I would base my fitness function on the following pseudo-code:

```
function [F, auxOutput]=fitness(X, extPar)
% Fitness function for a data fitting problem with a penalty function.
%
% A big number to scale the penalty function.
hugeNumber=1e10;
%
% Count backwards to avoid resizing arrays!
for i=size(X,2):-1:1
        %
        % Calculate the chi-squared of the fit and store in auxOutput, NOT F.
        auxOutput{i}=calcChiSquared(X(:,i), extPar);
        %
        % Calculate the degree of constraint violation.  Return 0 if no
        % constraints are violated, and about 1 for the maximum constraint
        % violation that is likely to be encountered.
        constraintViolation=calcViolation(X(:,i), extPar);
        %
        % Calculate the penalty.
        penalty=hugeNumber*constraintViolation;
        %
        % Calculate the fitness.
        F(:,i)=auxOutput{i}+penalty;
end
```

The actual fitness values returned by this pseudo-code would be meaningless in terms of χ^2 for solutions in the disallowed region. However, the χ^2 values would be available in the *auxOutput* cell array from within the output function, the History files, and the OptimalSolutions.mat file (if any penalized solutions are actually present in *OptimalSolutions*).

A penalty function should always be implemented in such a way that F increases with the degree of constraint violation. Users often try to implement a simple penalty function like the following:

$$penalty = \begin{cases} 0, & if \ \ no \ \ violation \\ bigNumber, & if \ \ violation \end{cases} \tag{4.1}$$

This does not work nearly as well because this penalty function is completely flat over the disallowed region, and therefore offers no hints about which way to go in order to satisfy the constraints. Ferret never calculates gradients, but it does work better when the parameter space offers some indication about where good solutions reside, without having to accidentally land solutions in the allowed region! *The art of designing a good penalty function, and perhaps a good fitness function in general, is to build in as many hints as possible to help guide the population to the optimal region.*

Note that *auxOutput* field is optional, and an empty set ($auxOutput = \{\}$) is also allowed.

4.8.2 saveData - Extra Data Saved during a Run

The *saveData* output argument is a $1 \times N$ cell array, which behaves like *auxOutput*, except that the results are saved to a special '*saveData*' sub-directory of the data directory rather than inside of the History files. The *saveData* field is intended for large data structures computed during the evaluation of

your fitness function that you want to save to disk, but are too large to include in your History files. This field is optional, and an empty set ($saveData = \{\}$) is also allowed.

4.8.3 XMod - Parameter Values Modified by the Fitness Function & Application to Normalization Constraints

The fitness function is allowed to modify X and return the modified values back to Ferret through $XMod$. This is an advanced feature that is rarely used, but is occasionally useful for handling constraints. For example, if some values of X are not allowed due to a constraint, it is possible for the fitness function to move these values to an allowed region by modifying them and returning them as $XMod$, and they will appear in the population in the new location.

It is usually better to handle most types of constraints by imposing fitness penalties for wandering into regions that are not allowed, as discussed in Section 4.8.1. However, the $XMod$ feature is sometimes better for normalization constraints that must be enforced exactly. A simple example is a problem in which the sum of the parameters (or a subset of them) must add up exactly to a constant value. To be specific, if the sum of the parameters must add up to 1, then you can enforce this normalization by the following:

```
XMod = X./repmat(sum(X,1), size(X,1), 1);
```

4.9 The Output Function

The output function is an optional function that is called exactly once per generation and gives you access to Ferret's top-level data structure, called the *world*, which encapsulates Ferret's entire internal state. This includes the parameters and fitness values of all individuals present in the current generation, as well as a lot of other information used internally by Ferret. This can be used to perform any required side-calculations to monitor the progress of your run, but you will need to save any results to disk, since the output function does not have any return arguments that can be used for this purpose. The output function is most commonly used to implement a custom stopping criterion, as discussed in Section 4.9.2.

The most commonly used signature of the output function is as follows:

```
function output(world)
```

4.9.1 Organization of the World Data Structure

The *world* data structure contains all populations[4], all of the parameter sets evaluated, all fitness values, the *auxOutput* cell array, ranking information, the linkage matrix, the *extPar* structure, and a great deal of other internal data. This is essentially a snapshot of Ferret's entire internal state.

The basic structure of the *world* is as follows:

[4]Note that Ferret can run multiple parallel populations if `par.general.NPop > 1`.

```
world.pop{p}  % Cell array of populations, indexed by p.
     |      |
     |      .indiv  % The individuals
     |      |     |
     |      |     .genome  % The genome information of each individual
     |      |     |      |
     |      |     |      .X                % Scaled parameter values, with NaN's from
     |      |     |      |                 % CPD.  Use this field to determine which
     |      |     |      |                 % parameter values are fully specified,
     |      |     |      |                 % and which ones (NaN's) are treated as
     |      |     |      |                 % irrelevant (random values) by Ferret's
     |      |     |      |                 % CPD system.
     |      |     |      |
     |      |     |      .XFullySpecified  % Scaled parameter values.  NaN's replaced
     |      |     |      |                 % by scaled values used internally.
     |      |     |      |
     |      |     |      .XPhys            % Actual parameter values sent to the
     |      |     |                        % fitness function.  These are the
     |      |     |                        % parameter values actually used during
     |      |     |                        % evaluation (not X or XFullySpecified).
     |      |     |
     |      |     .F          % Matrix of fitness values, as determined by the user's
     |      |     |           % fitness function.
     |      |     |
     |      |     .auxOutput  % Auxiliary output (see fitness function syntax).
     |      |                 % auxOutput is empty {} when not used.  When used, each
     |      |                 % cell index corresponds to its respective columns in
     |      |                 % the X and F matrices.
     |      |
     |      .BBList   % List of building blocks for each individual.  This should be
     |      |         % viewed as an evaluation of the probabilistic linkage matrix.
     |      |
     |      .ranking  % Ranking of individuals
     |      |       |
     |      |       .rankPareto  % Actual ranks
     |      |       |
     |      |       .FPareto     % Ranks modified if fitness sharing is used
     |      |
     |      .elites   % Elite members of the population
     |
     .link.P % The linkage matrix.
     |
     .par  % The par structure.  Includes all control parameters.
         |
         .user.extPar  % User-defined parameters from init
```

4.9.2 Implementing a Custom Stopping Criterion

The most common purpose of an output function is to implement a customized stopping criterion. Section 4.5 shows how to stop a Ferret run using simple tolerance criteria, but these criteria are not completely general. They may be insufficient for complicated problems, and are seldom useful for problems that focus on parameter space mapping, as discussed in that section. When the built-in method is inadequate and you don't want to stop your run manually, then you should implement your own stopping criterion. Every problem is different, and my philosophy is that you know best when your run should stop.

You can implement your own stopping criterion very easily by calling the `abortQubist` function, as discussed below. In principle, it is possible to stop Ferret from the fitness function, or any other function called by Ferret for that matter, by calling `abortQubist(1)`. However, it really is best to implement your stopping criterion in the output function. The fitness function is an especially poor place to do this because it is called several times per generation, with varying numbers of solutions, and sometimes only a few. Thus, the fitness function only sees a fraction of the individuals present in the *world* at a time, and would be making a termination decision based on incomplete information.

The output function is called only once per generation, and receives the current state of all individuals within Ferret's top-level *world* data structure, as shown above. Other fields may also be present, but only the fields shown would be of any conceivable use for designing a stopping criterion. The following pseudo-code can be modified to implement a customized stopping criterion:

```
function output(world)

if checkStop(world)
        abortQubist;
end
```

Here, 'checkStop' is a user-defined function that should examine the population for convergence and return the logical value *true* to stop Ferret, or $false$ to continue. The output function calls the Qubist function `abortQubist` if checkStop returns true, which sets a global variable $abort\_ = 1$, which causes Ferret to stop cleanly at the end of the present generation. It is also possible to call `abortQubist(2)` to set $abort\_ = 2$, which caused an 'emergency stop'. This stops Ferret more quickly, but this is not recommended because some cleanup operations are omitted, and you may find that some information has been lost if you try to resume the run later on. You should always use `abortQubist(1)` instead, unless there really is an 'emergency' condition.

Note that you can also get the same effect by directly setting a global variable $abort\_ = 1$, rather than calling `abortQubist`. However, you should use `abortQubist` instead for greater compatibility with the other Qubist optimizers. As you will see in Chapters 9 and 10, SAMOSA and Anvil use different global variables for their abort signals - specifically $abortSAMOSA\_$ and $abortAnvil\_$. The `abortQubist` function automatically sets the appropriate variable by determining which optimizer is running. The full syntax of `abortQubist` is as follows:

```
abortQubist({abortStatus}, {optimizerName})
```

where *abortStatus* is 1 (normal stop) or 2 (emergency stop), and optimizerName is a string containing the name of the intended optimizer: 'Ferret', 'Locust', 'SAMOSA', 'Anvil', or 'SemiGloSS'. The braces indicate that both arguments are optional. The `abortQubist` command has no effect if the optimizer that is running does not match $optimizerName$, with the exception that Ferret and Locust both use $abort\_$ and are therefore respond to eachother's abort signals. You can also call `abortQubist(abortStatus,'all')` to

abort any optimizer that is running. For example:

- `abortQubist` → cleanly stops the current optimizer, whatever it is.
- `abortQubist(1)` → cleanly stops the current optimizer, whatever it is.
- `abortQubist(2)` → emergency stops the current optimizer, whatever it is, without cleanup
- `abortQubist(1,'SAMOSA')` → only stops SAMOSA, and does so cleanly.
- `abortQubist(1,'Ferret')` → stops Ferret *or* Locust, and does so cleanly.
- `abortQubist(1,'Locust')` → stops Ferret *or* Locust, and does so cleanly.
- `abortQubist(1,'all')` → stops any Qubist optimizer, and does so cleanly.

This fine level of control may be useful for complicated problems in which one optimizer is called from inside the fitness function of another. The demo 'Advanced/Ferret-SAMOSA-Anvil-Insanity' gives an example of such a problem, and is discussed in Section 4.6.2. *Note that Ferret and Locust cannot call each other, which explains why it is OK to use abort_ for both of them.*

Ferret's custom stopping mechanism is perhaps less convenient for some problems than a hard-wired stopping criterion bases on fitness values and convergence (see Sections 4.5 and 8.15), but having this option ensures maximum flexibility. It is often required for problems that focus on mapping out a region of parameter space, where the built-in method discussed in Section 4.5 is not usually adequate.

4.9.3 Advanced Features: Modifying *extPar* and the *world*

Advanced users may occasionally find the following forms useful:

```
function extPar=output(world)
function [dummy, world]=output(world)
```

The first form allows you to modify the *extPar* parameter, generated by your init function, and send it back to Ferret. Any changes made here will be visible to your fitness function during the next generation. The *extPar* data structure is available to the output function by accessing `world.par.user.extPar`.

The second form should be used with extreme caution, because it allows your output function to change Ferret's top-level *world* data structure and send it back to Ferret. If you make a change that results in an inconsistent *world*, you might crash your run, or - even worse - Ferret might continue and your results will not be valid. Make sure that you know what you are doing if you attempt this, or at least be prepared to do significant testing.

Qubist includes a demo called 'Advanced/Data-Modeler-modWorld', which shows how to use this feature. The demo generates some artificial data, and then tries to model it with a Fourier series, like all of the other 'Data-Modeling' demos. The unique twist to this particular demo is that the initial optimization starts off by modeling the data with only two terms in the Fourier series, which requires four parameters. Every 25 generations, the output function refines the model by adding another Fourier term, which requires an additional two parameters. This is done by modifying $world.pop\{p\}.indiv.genome.XPhys$ directly and sending the changes back to Ferret. Ferret then does some repair operations to $world.pop\{p\}.indiv.genome.X$ and $world.pop\{p\}.indiv.genome.XFullySpecified$ to make the *world*

self-consistent, evaluates the fitness function for all individuals, repairs the linkage matrix to reflect the increased number of parameters, adds the new parameters to the graphical display, and carries on until the next refinement!

The following output function shows how the 'Advanced/Data-Modeler-modWorld' demo modifies the *world*:

```
function [dummy, world]=outputFerret(world)
dummy=[];

% Add parameters every NRefine generations:
NRefine=25;

if mod(world.par.bookKeeping.gen, NRefine) == 0
    [NGenes, N]=size(world.pop{1}.indiv.genome.XPhys); %#ok
    world.par.general.min=[world.par.general.min, 0, 0];
    world.par.general.max=[world.par.general.max, 1, 2*pi];
    world.par.general.cyclic=[world.par.general.cyclic, NGenes+2];
    for p=1:length(world.pop)
        %
        % Add 2 new parameters by adding 2 rows to world.pop{p}.indiv.genome.XPhys:
        XMod=[rand(1,N); 2*pi*rand(1,N)];
        world.pop{p}.indiv.genome.XPhys=[world.pop{p}.indiv.genome.XPhys; XMod];
    end
    %
    % Tell Ferret that you want to restart.
    setGUIData('restart', true);
end
```

As you can see, it's actually not all that complicated to set up a model refinement scheme that adds parameters as the run progresses. However, there are some details and default behaviours that you need to be aware of before attempting this.

Your output function requires a *trigger* that tells your code when to modify the *world*. The demo is set up to refine the model every 25 generations, but this is too simplistic for most real problems. Normally, you should monitor the convergence of the population and refine when some condition is met. The actual condition is of course entirely up to you, but note that modifying the *world* like this is computationally expensive because it requires all individuals to be re-evaluated, and usually at least a little disruptive to the populations. Therefore, your trigger should be designed so that it does not activate very often.

The basic requirements for modifying the *world* are as follows:

- Modify `world.par.general.min` and `world.par.general.max` to reflect the change in the parameter limits. If you are adding parameters, the best strategy is to add them at the end of the parameter list, since this is the least disruptive to the linkage matrix and critical parameter detection (CPD) system, as discussed below.
- Modify `world.par.general.cyclic` and `world.par.general.discreteStep` to tell Ferret which parameters are cyclic or discrete, if this information has changed. Every second parameter of the demo is cyclic on the interval $[0, 2\pi]$, so I add one new cyclic parameter each time a refinement occurs.
- Modify `world.pop{p}.indiv.genome.XPhys` as required by your problem. Here, I initialize the two

new parameters to random values within their respective ranges by adding two rows to $XPhys$. Note that you must loop over all populations if your problem uses multiple populations. The best practice is to add new parameters to the end of the parameter list (bottom rows of $XPhys$), for reasons discussed below. Likewise, if any parameters are to be deleted, you should try to set up your problem so that they are deleted from the end, so that the indices assigned to other parameters do not change. (i.e. Parameters 1:9 remain parameters 1:9 when you delete parameter 10. However, deleting parameter 1 changes all of the indices. This is to be avoided.)

- Send a 'restart' signal to Ferret using the command `setGUIData('restart', true)`. This does not *really* restart Ferret, but it it does tell Ferret that you have changed the *world*, which requires some internal systems to be re-initialized. Ferret will probably crash if you modify the *world* and forget to send the restart signal. Note that the restart signal should only be sent when your trigger is *true* and you have actually modified *world*.

Ferret will automatically fix `world.pop{p}.indiv.genome.XFullySpecified` to reflect changes to your parameters, so you do not need to modify this field in your output function. Ferret will also fix `world.pop{p}.indiv.genome.X`, by assuming that the CPD information is still valid for parameters that are still present. Thus, `world.pop{p}.indiv.genome.X` will be set equal to `world.pop{p}.indiv.genome.XFullySpecified`, and then NaN values will be inserted wherever they are present in the previous `world.pop{p}.indiv.genome.X`. Thus, CPD information is preserved by default for all parameters that are still present. You can override this behaviour if it does not suit your problem by setting `world.pop{p}.indiv.genome.X` manually inside of your output function. A reasonable choice would be to restart the CPD system by setting `world.pop{p}.indiv.genome.X(:)=0` to remove all NaN values. When Ferret rebuilds `world.pop{p}.indiv.genome.X`, after the restart, it will be set to the correctly scaled parameter values, but all previous CPD information will be destroyed.

Ferret will automatically re-evaluate all fitness values for the entire population, so you do not need to do this manually inside of the output function. This may be a computationally expensive process, but it is necessary because modifications to the world in the output function likely mean that the underlying model has changed.

Ferret will assume that all linkages are still valid for parameters that are still present in `world.par.general.min` and `world.par.general.max`. For example, the demo initially has four parameters, so when this is increased to six during the first refinement on generation 25, Ferret expands the linkage matrix to a 6×6 matrix, but assumes that any linkages previously discovered for the first four parameters are still valid. Rows five and six, and columns five and six of the new linkage matrix will be assigned the initial linkage value `par.link.initialLinkage`, as discussed in Section 8.13. This might be a poor choice for some problems if your changes disrupt the problem's linkage structure, and you probably won't know if this is the case without some experimentation. You can restart the linkage learning system manually with the following lines of code:

```
NGenes=length(world.par.general.min);
world.link.P=world.par.link.initialLinkage+zeros(NGenes);
world.link.P1=world.link.P;
world.link.P2=world.link.P;
```

Note that a side-effect of a restart is that elites are destroyed, because they are solutions from a previous generation from before the model was changed, and are therefore no longer valid. As a result, you may discover that fitness values increase slightly for single-objective problems immediately after a restart, when all solutions are re-evaluated. This is not normally a serious problem, and fitness values should continue to

decrease monotonically between restarts.

4.10 The myPlot Function

Ferret contains a mechanism that allows you to plot your own function in both the console and analysis windows. This is done by setting `par.interface.myPlot` in the setup file to a string corresponding to the name of the user's plotting routine, which must be on the path loaded by the project. The possible signatures of the myPlot function are as follows:

```
function myPlot(X)
function myPlot(X,extPar)
function myPlot(X,F,extPar)
function myPlot(X,F,auxOutput,extPar)
function myPlot(X,F,auxOutput,rankPareto,extPar)
```

The most common usage, and the use used by nearly all of the demos, is the third one: `myPlot(X,F,extPar)`. Here, the user-defined myPlot function receives the current parameter and evaluated fitness arrays X and F for all individuals, from all populations in the *world* data structure, as well as the *extPar* data structure that is loaded from the init file. By using the more elaborate signatures, the *auxOutput* field from the fitness function is also available as a cell array, and Pareto rank is available via the *rankPareto* argument. Note that there are no values returned to Ferret by the myPlot function, since this function should be used for plotting only. Occasionally, a myPlot function requires other information from the FerretSetup file besides *extPar*. Ferret's full internal *par* structure from the setup file (minus the *extPar* structure) is contained within `extPar.QubistPar` to accommodate such rare cases.

Note that because the *entire* X and F matrices are received, the myPlot function often contains lines like the following to extract the single 'best' solution for the purpose of plotting, assuming a single-objective problem:

```
function myPlot(X,F,extPar)
[FMin, index]=min(F);
XMin=X(:,index);
```

A multi-objective problem might select the individual to plot randomly with a line like the following:

```
index=ceil(rand*size(X,2)),
```

or by some other selection criterion.

The myPlot function should only contain plotting commands that ultimately result in graphical output. Note that Ferret gives control to the relevant graphics axes before calling the myPlot function. *It is crucial that myPlot does not contain commands that attempt to set the figure or axis directly.* If your myPlot command contains such commands, then it is likely that your plot will pop up in its own window, which is probably not the desired effect!

Finally, it is important to note that Ferret's call to the myPlot function is protected by a try/catch structure that captures and ignores all errors that occur in myPlot, because this function is always regarded as non-critical. If your plot fails, the error will be handled silently, and Ferret will continue. The 'User Plot' axes on the console and analysis windows will be coloured red and contain a 'Plot Failed' message, but no other information will be available. The reason for this heavy-handed error trapping is that the myPlot function must *never* contain any processing that is important enough to crash the run if an

error occurs. The lack of error messages makes it slightly less convenient to debug the myPlot function, but this can be done by placing a debugging breakpoint in the file if you see the 'Plot Failed' message. In any case, this behaviour is much better than crashing a long run because of an inconsequential plotting error.

4.11 Running Your Project

Section 4.4 discussed how to add user-defined projects to Ferret by editing a few lines of the launch file. Before trying this with your own project, I would recommend trying it by copying one of the demo directories in the [Qubist_Home]/demos directory to a different directory outside of the [Qubist_Home] path and adding it as a project. If you have added this project correctly, it should appear in the projects menu. If it is not there, then you have most likely given the wrong directory location in the `projects{n}.path` line of the launch file. Check the MATLAB command line for messages indicating what may have gone wrong and try again.

To launch your project, simply select it and follow the instructions given above for running demos. History files and the final optimal set, called 'OptimalSolutions.mat' will be saved in the data directory specified by the `par.history.dataDir` field of the setup file. Note that the data directory is usually given as a local path, which refers to a directory inside of the project directory. Such a local path is specified as follows:

```
% A directory inside the project directory:
par.history.dataDir='myDataDirectory'; % --> [project_dir]/myDataDirectory
%
% To collect runs inside of a subdirectory called 'FerretRuns':
par.history.dataDir='FerretRuns/myDataDirectory1';
%      Saves to: [project_dir]/FerretRuns/myDataDirectory1
par.history.dataDir='FerretRuns/myDataDirectory2';
%      Saves to: [project_dir]/FerretRuns/myDataDirectory2
```

It is also possible to specify a global path to an arbitrary location on your file system using lines like the following examples:

```
par.history.dataDir='C:\Users\JFiege\Documents\myDataDirectory'; % on a Windows system.
par.history.dataDir='/Users/JFiege/Documents/myDataDirectory'; % on a MacIntosh system.
par.history.dataDir='/home/jfiege/myDataDirectory'; % on a Linux system.
```

4.12 What Do the Figures Represent?

The four figures in the Ferret console window give the user a detailed view of the status of the population at the end of each generation. This is interesting to watch and can provide important information about the progress of your run. With experience, it is possible to troubleshoot many problems with the Ferret setup file by examining these figures.

Let us examine the figures in more detail:

1. Top Left: 'Evolution Stats': Like all genetic algorithms, Ferret is an iterative algorithm that cycles through generations as it runs. Each generation is characterized by a set of 'populations'[5] (see

[5]Ferret can run multiple populations simultaneously, which interact only weakly through an 'immigration' operator, as

Section 4.9.1), and each population contains 'individuals' that encode the parameters being searched and their fitness value. The group of populations is the current group of solutions that Ferret is gradually evolving toward the optimal set.

- The number of individuals (solutions) in the optimal set ($rank = 1$) is shown for each generation, as well as the average fitness of the population. The average fitness is replaced by the median rank of the population for multi-objective problems.
- Multi-objective problems generally contain more than one optimal solution, and even single-objective problems can contain more than one optimal, especially if you are using one of the tolerance options (`par.selection.FAbsTol > 0` or `par.selection.FRelTol > 0`), which allow a certain amount of fuzziness in the optimal set (see Section 8.6). For these types of problems, you should see the number of individuals in the optimal set climb to some threshold value and approximately level off once a stable population is achieved. For single-objective problems with a single minimum and no fuzziness (`par.selection.FAbsTol=0` and `par.selection.FRelTol=0`), it is normal for the optimal set to contain only a single solution for most generations during the run.
- The average fitness or median rank should decline as the solution is optimized, and eventually level off. *Recall that Ferret minimizes the fitness function, which is opposite the convention used by most other genetic algorithms.*

2. Top Right: 'Optimal Fitness Bounds': The optimal set generally contains multiple solutions for multi-objective problems, and may also contain multiple solutions for single objective problems.
   - This figure gives the maximum and minimum bounds of each objective for the population (or multiple populations) at each generation. The minimum is just the fitness of the best individual of the generation for single-objective problems. The minimum and maximum bounds are equal for non-fuzzy single-objective problems, with `par.selection.FAbsTol = par.selection.FRelTol = 0`.
   - The minimum and maximum bounds are not equal for fuzzy single-objective problems with `par.selection.FAbsTol > 0` or `par.selection.FRelTol > 0`. These settings indicate that distinctions in rank should not be made between individuals that are within a certain fitness tolerance of each other. In this case, the optimal set (rank = 1) often contains a multitude of solutions with fitness values that lie within the specified tolerance, and the minimum and maximum values are plotted. These setup parameters are discussed at length in Chapter 8.
3. Bottom Left: 'Ferret Parameter Distribution': This figure can display any two or three-dimensional projection of the parameter and objective spaces.
   - The X-Axis, Y-Axis and Z-Axis menus are used to control which parameters or fitness values are plotted.
   - There is a choice of display option available in the menu in the upper left corner of this figure. These include a simple two-dimensional scatter plot, a three-dimensional scatter plot, an image plot, a contour plot, two-dimensional 'combo plots' that combine image or contour plots with a scatter plot, and a three-dimensional combo plot. The three-dimensional options require significant computation to render, and are not recommended if you are running with a population size greater than a few hundred.

discussed in Section 8.11. This can sometimes be helpful because it tends to slow down convergence while encouraging more thorough exploration.

- The figure options available for this panel of the Ferret console are a subset of the options available for the analysis window, which will be discussed in greater detail in Section 4.15.

4. Bottom Right (multiple views): This figure is controlled by the buttons below it, and can take one of three possible views:

   - The 'Linkage' view shows the linkage matrix. This is Ferret's current working model of how parameters are linked and how the problem naturally divides into sub-problems. White squares indicate tightly linked parameters, while dark squares indicate parameters that are weakly linked. The linkage matrix evolves during the run as Ferret discovers the non-linear structure of your problem. Note that the linkage model is *dynamic*: linkages are discovered during the optimization process, and are gradually forgotten if Ferret evolves toward a region of parameter where the linkage is no longer valid. Linkage-learning options are discussed in Section 8.13.
   - The 'C.P.D. Plot' is a histogram showing Ferret's working model of the relative importance of each parameter in the problem, as determined by its Critical Parameter Detection (CPD) algorithm. The bars represent the number of active copies of a gene (parameter) in the population. When a parameter is of low importance, there are few active copies, and the corresponding bar in the histogram drops. This should be thought of as a qualitative statement of relative importance only. Ferret uses this information internally to focus on 'important' sub-spaces of the parameter spaces, but you should not try attempt to use the height of the bars in any quantitative or statistical sense. Please refer to Section 8.10 for an explanation of Ferret's CPD options.
   - The 'User Plot' is a MATLAB axes component that you can access for custom plotting. This is done by setting `par.interface.myPlot` to the name of a valid m-file containing graphics commands, as explained in Section 4.10. If you do not implement a plotting routine, then this figure displays 'No User Plot'. The window displays 'User Plot Failed' if an error occurs while evaluating the myPlot graphics routine.

4.13 Movies

You may find it useful to show a movie of your Ferret run in talks, or view them yourself to help understand how your run converged. This is sometimes useful for diagnosing problems. It is easy to generate a movie of your run right from the Ferret console window by pressing the 'Show Movie' button in the lower right part of the window. This will bring up a dialogue box like the one shown in Figure 4.9. You can safely make a movie at any time during your run, whether it is complete or still active.

As you can see from the dialogue box, three different movie options are available:

- Show: In my opinion, this is probably the most useful movie option. All this does is load your History files sequentially, reconstruct the Ferret window from the saved data, and show it before moving on to the next generation. I use this feature often on runs that are still going in order to get an overview of how the run has been progressing and how well the population is converging. This option does not save any files to disk.
- Save Frames: This option is like the 'Show' option, except that snapshots of the Ferret window will be saved as .mat files, using the same format that MATLAB's 'getframe' function uses for frames. The

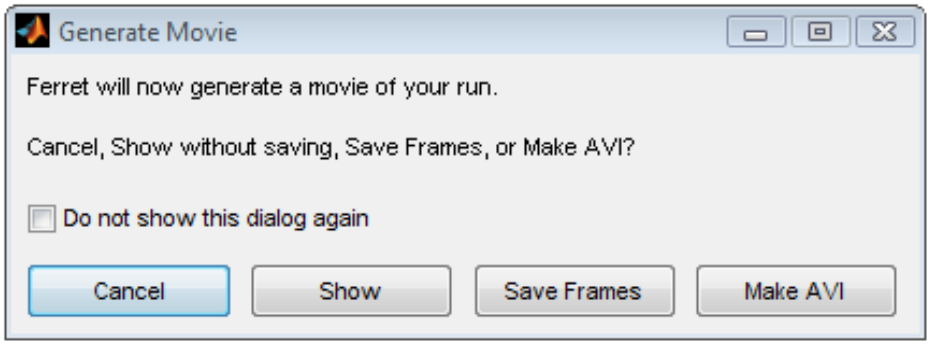

Figure 4.9: The dialogue box that results from clicking the 'Show Movie' button in the Ferret console window.

frames will be saved in the 'movieFrames' folder of the run's data directory. Frames can be viewed in MATLAB by double-clicking on a frame .mat file to load it, and issuing the following commands:

```
figure(1);
image(movieFrame.cdata);
set(gca,'XTick',[],'YTick',[]);
axis image
```

Frames can be later combined into an AVI movie using the Qubist Movie Tool, which is discussed in Section 12.6.

- Make AVI: This option will load your history files and generate frames like the 'Save Frames' option, and will compile them into an AVI movie. The movie will be located inside of your run's data directory, and will have the same name as the data directory with an .avi extension. The default movie name is therefore 'FerretData.avi'. No movie compressor will be used, for maximum compatibility between systems. If you want more control over how your movie is created (choice of compressor, framerate, and quality), then you should just save frames and run the Qubist Movie Tool on the saved frames. The Movie Tool is discussed in Section 12.6.

Please note that when using the 'Show' option, only the data actually present in the History files will be displayed. If you are only saving the optimal solutions and elites each generation (the default), then only these solutions will appear in the movie. If you think you might want to view movies, you might consider setting `par.history.saveFrac=1` to save *all* solutions (see Section 8.2). Alternatively, you could set `par.movie.makeMovie=true` to automatically save a frame each generation (see Section 8.19).

Note that you can stop the generation of a movie by clicking the 'Stop Ferret' button. If Ferret is still running, this won't actually stop it. Ferret will stop collecting movie frames and generate an AVI file if you chose this option, and your run will continue automatically.

4.14 Running Ferret Without Graphics

Most users prefer to run Ferret through the user interface because it gives a clear picture of the progress of the run, and consumes only minimal resources, especially when left in minimal graphics mode (see Section

4.2.1). Full graphics mode is especially useful when you have not set up a stopping criterion following the methods described in Sections 4.5 and 4.9.2. In practice, many problems that require parameter space mapping are best stopped manually, because the limiting factors are often the length of time that the population has been stable, the desired quality of the final figures, and how long you are willing to wait to build up your solution set. It is usually better to run such problems through the graphical interface, although it is certainly possible to run them on the MATLAB command line and use Qubist visualization tools to analyze and view them periodically. *Note, however, that pressing 'Control-C' is the only way to stop Ferret prematurely when running without the interface.* This performs no cleanup operations, and you will lose any generations that have been computed since the last History file was saved.

The best reason to turn off graphics is to make very long-duration runs more robust against MATLAB crashes. This is rarely a problem under Windows or MacIntosh, but in my experience, some previous versions of MATLAB have not always been completely stable when doing graphically intensive operations under Linux (See Section 4.16.1 for an example.). If you suspect that you may be experiencing these kinds of problems, it is worth trying to run the code without the graphical user interface for diagnostic purposes.

4.14.1 Turning Graphics Off

The simplest way to run Ferret without graphics is to start the run through the graphical user interface, switch to minimal graphics mode by pressing the 'Turn Graphics Off' button in the console window, and close the window. If you close the console in full-graphics mode, it will re-appear on the next generation. However, if you close the console while running in minimal graphics mode, it's gone for the duration of the run.

4.14.2 Running Ferret from the Command Line: Custom Launchers and Batch Processing

If you want to start Ferret from the MATLAB command line without *ever* starting the graphical interface, you need to build a custom launcher for your project. Fortunately, this is not difficult, and I have included an example with the Qubist distribution. The following is a verbatim copy of the file [Qubist_Home]/user/Ferret/scripts/customLauncher.m, which runs the 'Data-Modeling/Data-Modeler' demo three times, writing History files to three different data directories. This file can be modified to build a custom launcher for your problem. It would be easy to modify this file to run Ferret several times to model multiple data sets, which is probably the best and most commonly used custom launch technique.

```
global QubistHome_ isRunning_ currentProjectPath_ currentComponent_ useMex_

% Modify this script if you need to make a custom launcher for Ferret.
% This is useful if you need to run Ferret multiple times in a customized
% batch mode.

% Set the current component to Ferret.
currentComponent_='Ferret';

% ----------------------------------------
% MODIFY THIS LINE TO POINT TO THE Qubist HOME DIRECTORY.
% This is usually a global path, but a relative path also works (even for parallel ↘
      ↪ runs)
% because a side-effect of setQubistPath is to convert QubistHome_ to a global path.
QubistHome_='../../..';
addpath(QubistHome_);
setQubistPath(QubistHome_);
%
% Give the project path and setup file name.
currentProjectPath_=[QubistHome_, '/demos/Data-Modeling/Data-Modeler'];
setupFile='FerretSetupCustom';
NRuns=3;
% ----------------------------------------

% Use compiled functions or not?
useMex_=true;

% Add Qubist home.
addpath(QubistHome_);
%
% Force Qubist abort status to 0 so that Ferret can run.
forceAbortQubist(0);
isRunning_=0;

% Add the project path.
addpath(genpath(currentProjectPath_));

% Get the Matlab version.
MatlabVersion=getMatlabVersion;

% Parallelization info.
setGUIDataRoot('parallel', false);
launchDir=fileparts(which(mfilename));
setGUIDataRoot('launchDir', launchDir);
```

```
% --------------------------------------------------------------------
% Initialize the random number generator.  It may occasionally be useful
% to re-generate runs *exactly* by using exactly the same pseudo-random
% sequence.  If this is desired, just comment out the first line below
% that initializes the random number generator using the clock, and
% uncomment the second line, which resets the random number generator to
% its default state.  Random number initialization is not done anywhere
% else in Qubist.
%
% Uncomment for DIFFERENT random sequence each time.
if MatlabVersion >= 7.7
    s = RandStream.create('mt19937ar','seed',sum(100*clock));
    RandStream.setDefaultStream(s);
elseif MatlabVersion >= 7.4
    rand('twister', sum(100*clock)); %#ok
else
    randn('state', sum(100*clock)); %#ok
end
%
%     Uncomment for SAME random sequence each time.
%     if MatlabVersion >= 7.7
%         s = RandStream.create('mt19937ar','seed',5489);
%         RandStream.setDefaultStream(s);
%     elseif MatlabVersion >= 7.4
%         rand('twister', 5489);
%     else
%         rand('state', 0);
%     end
%     randn('state', 0);
%
%     Other options are possible.  Refer to the Matlab's documentation on
%     random number generators by typing 'help rand'.
% --------------------------------------------------------------------
```

```
for i=1:NRuns
    %
    % ----------------------------------------
    % Load the init file.  You should use getExtPar to do this, because
    % getExtPar sets options that are required for parallel runs.  If you
    % don't want to do parallel runs, you *can* simple do:
    % extPar=init;
    %
    runInfo.projectPath=currentProjectPath_;
    runInfo.initFile='init';
    extPar=getExtPar(runInfo);
    % ----------------------------------------
    % *** Additions to extPar here... ***
    %
    % These extPar fields are required.
    extPar.mode='RunNoGraphics'; % Uncomment to run with no graphics (normal for ↘
        ↪ batch processing).
    % extPar.mode='Run'; % Uncomment to run with full graphics.
    %
    % extPar.mode='Run'; % Uncomment to run with full graphics.
    extPar.projectPath=currentProjectPath_; % ESSENTIAL!
    % ----------------------------------------
    % The user can also make his/her own modifications
    % to the template here.
    extPar.runNumber=i;
    % ----------------------------------------
    %
    Ferret(extPar, setupFile);
end

% ====================================

function MatlabVersion=getMatlabVersion
% Get the Matlab version.
MatlabVersion=version;
MatlabVersion(MatlabVersion == '.')=[];
MatlabVersion=MatlabVersion(1:3);
MatlabVersion=str2num(MatlabVersion)/100;
```

There are a few key points to keep in mind when designing a custom launcher for a batch job:

- Ferret requires a global variable called $QubistHome\_$, which is a string that contains the location of the Qubist home directory on your file system. This is normally a global path, but you *can* use a local path if you want. A program called 'setQubistPath.m' is called to set all of the paths required for Qubist, and a side-effect of this program is that $QubistHome\_$ will be converted to a global path.

- Ferret requires a few other global variables that are listed near the top of the file. These must be included in a custom launcher. The meaning of each global variable is clear from its name.

- The random number generator should be initialized with the system clock if you want multiple runs of the custom launcher to be different. If you forget to re-seed the random number generator, you may find that two runs of your project in different MATLAB sessions are identical in every detail. This is occasionally useful for testing, but not otherwise, because the point behind doing multiple runs is usually to build up statistically independent solution sets that can be combined later on using the 'Merge OptimalSolutions Tool' discussed in Section 12.4. If you really do want subsequent runs to be identical, uncomment the appropriate lines (starting with 'Uncomment for SAME random sequence each time.') to put the random number generator into a fixed state.

- If your project requires an init file, you must call it from in your custom launcher to generate the *extPar* data structure. If you want to run in parallel mode, then you need to obtain *extPar* using the `getExtPar` command, which also sets some essential options to prepare Qubist for parallel computation. Specifically, you should include lines like the following:

```
runInfo.projectPath=currentProjectPath_;
runInfo.initFile='init';
extPar=getExtPar(runInfo);
```

  If you don't want to do any parallel computing, then you can call your init file directly with a line like `extPar=init`, and also skip the steps labelled 'Parallelization Info'.

  The *extPar* structure and the name of the setup file (*setupFile*) must be passed to Ferret by the call Ferret(extPar, setupFile). Normally, *setupFile* is just a string containing the name of the file if it is located in the project directory. Alternatively, *setupFile* can contain a path relative to the project directory, or it can be given as an absolute path.

- Ferret returns three optional output arguments: `[X,F,History]=Ferret(extPar,setupFile)`. Here, X is the final set of parameters from the last generation of the run, F is the final set of fitness values, and History is the final History data structure that is written to disk. In practice, these output arguments are seldom used because your final results will be obtained later when you perform an analysis, as discussed in Section 4.15.

- Runs produce History files that can be analyzed after the end of the run using the Qubist 'Analyze History Tool', which can be started from the component selector. See Section 12.2 for detailed information about this tool.

4.14.3 Returning Values from Ferret

Note that the custom launcher above calls Ferret, but does not receive any values back. Ferret always writes History files to disk as it runs, which preserve its state and all optimal solutions that are discovered at each generation. These History files must be analyzed to calculate the final optimal set. This can be done by using the Analyze History tool discussed in Section 12.2. So what if you prefer to receive the results as an an output argument of the custom launcher, rather than performing the extra analysis step for each Ferret run? You can do this by including the line

```
par.analysis.analyzeWhenDone=true;
```

in your setup file, to trigger automatic analysis when Ferret finishes, and calling Ferret using the syntax

```
[X{i}, F{i}, OptimalSolutions{i}]=Ferret(extPar, setupFile);
```

Analysis will be done automatically, and `OptimalSolutions{`i`}` will contain the *OptimalSolutions* structure for the i^{th} run. If you forget to set analyzeWhenDone to true, then you will receive the final History data structure when Ferret finishes, instead of *OptimalSolutions*. The data stored in a History cell array is discussed in detail in Section 4.16.4, and the *OptimalSolutions* structure is discussed in Section 4.18. *You should be very careful when using this method for batch runs because OptimalSolutions structures can be large, and you will accumulate one for each run.* It is usually better to just set `par.analysis.analyzeWhenDone=true` to trigger analysis, but call Ferret without output arguments. When your custom launcher finishes, you will have to read an OptimalSolutions file from disk for each Ferret run, but you are much less likely to run out of memory!

4.15 Analysis: Generating Your Final Results

As Ferret runs, it generates 'History' files in the project's data directory (called 'FerretSetup' by default), which contains detailed information about the state of the run at each generation and the solutions that were considered to be optimal at that generation. The optimal solutions discovered at later generations are always at least as good, and usually superior to the 'optimals' recorded in History files from early generations.

A run can be analyzed after it is stopped, or at any point while it is running. There are several methods for doing this, but all methods trigger an identical sequence of events that compute the final optimal set of solutions from your run. Analysis also starts Ferret's integrated visualization system, which is designed to help you to explore your solution set and understand its structure and distribution in the parameter space.

The easiest way to analyze a run is to click the 'Analyze' button on the Ferret console window or select 'Analyze History' from the 'Control' menu, as discussed in Section 4.3. This will bring up Ferret's analysis window, as shown in Figure 4.7. Other views of this window are shown in Figure 5.2, which shows the results from a detailed data-modeling example presented in Section 5.1. Analysis also generates an 'OptimalSolutions.mat' file in the data directory (`par.history.dataDir`), which contains contains all of the final results from your run, including the final optimal set and other information generated during the analysis of the run. The OptimalSolutions.mat file is discussed in detail in Section 4.18.

In addition to the method discussed above, it is also possible to trigger the re-analysis of a run from the analysis window by pressing the 'Re-Analyze' button located in the 'Analysis' window. Re-analysis is usually only necessary if the 'Fuzziness' button is used to add or modify a fuzzy tolerance for the generation of the optimal set. Fuzzy tolerances are discussed in Section 5.6, and the Fuzzy Tolerance interface, which is opened by pressing the Fuzziness button, is shown in Figure 5.9.

Finally, you should be aware that the Qubist package contains an 'Analyze History Tool' that can be launched from the component selector menu when you run launchQubist. This tool is *extremely* useful, because it enables you to put off analyzing your runs until you are ready to do so. I most often use this feature when I'm doing several similar runs, often over the course of several days, and want to do all of the analysis at the same time when everything finishes. The Analyze History Tool is discussed in Section 12.2 of the 'Other Tools' chapter.

4.16 Polishing Solutions at the End of a Run

Ferret is very good at finding globally optimal solutions, but local optimization techniques are often superior for polishing solutions to high accuracy if they are started in the neighbourhood of a global solution. Ferret is able to call SAMOSA, Anvil, SemiGloSS, or MATLAB's built in optimizer fminsearch, as polishers after a run is analyzed. All of the Qubist optimizers can be used on multi-objective problems where SAMOSA is the default polisher, while fminsearch is limited to single-objective problems. You can choose which polisher you want by setting `par.polish.optimizer` to the name of the desired optimizer in your setup file (see Section 8.18), or to 'default' to allow Ferret to choose. I recommend that you use the default polisher for most problems because both SAMOSA and fminsearch are fundamentally simplex algorithms and relatively fast compared to Anvil. Anvil may be useful occasionally, but it is quite a powerful global optimizer in its own right and may take longer to finish.

Polishing your solution set is easy. It is normally done after a run has stopped, but can also be done while the run is still active. After analyzing your run, you simply press the polish button and polishing begins automatically. If you are using a Qubist optimizer with graphics enabled in its setup file (`par.interface.graphics=1` in SAMOSA_setup.m, AnvilSetup.m, or SemiGloSS_setup.m), then the polisher's window will pop up and you can watch polishing as it proceeds. Polishing typically involves looping over multiple optimal solutions from Ferret and sending them one by one to the polisher. Polishing can be stopped at any time by pressing the 'Stop Polish' button, but you will be asked to wait until the polisher finished with the current solution. If you are in a hurry to stop the polisher and the polisher's interface is visible, then you can stop it immediately by clicking on the 'Stop Polish' button in the Ferret analysis window and then polisher's 'Stop' button. This will stop the polisher, but your Ferret run will continue if it is still running.

The default setup files for all of the polishers are located in [Qubist_Home]/user/Ferret/defaults. *Note that this is not the same default directory that the polishers used when called as stand-alone optimizers.* This may seem confusing, but it is best to keep the polisher and stand-alone defaults distinct, because it allows different default options for these very different applications. By keeping them separate, you can change the defaults for one use without affecting the other. You will notice that many of the usual options are commented out if you open the polisher default files. This is done to make explicit the fact that Ferret will modify the commented-out fields automatically when the polisher starts. Of course, you can also copy and modify the full setup file for your polisher to your project directory, where you can modify it as you wish. Settings modified in this manner will override the default file.

Polishing your solution set will result in a 'PolishedSolutions.mat' file, which will be located alongside your OptimalSolutions.mat file in your run's data directory. A PolishedSolutions.mat file has exactly the same structure as an OptimalSolutions.mat file, but contains only solutions from the polisher. The PolishedSolutions.mat file will therefore work with all other Qubist tools designed to interact with *OptimalSolutions*, such as the 'View OptimalSolutions' tool and the 'Merge OptimalSolutions' tool, which are discussed in Chapter 12. However, you can usually just discard the PolishedSolutions.mat file because the final step of polishing is to add the contents of PolishedSolutions to OptimalSolutions, re-analyze the combined data to produce the final optimal set, and save it as OptimalSolutions.mat.

Please refer to Section 8.18 for a complete explanation of polisher options. You should also note that polishers can also be called by Locust following exactly the same procedure.

4.16.1 History File Options and Crash Recovery

The parameters that control the generation of History files are specified by just three lines in the `par.history` section of the setup file or in the default Ferret setup file ([Qubist_Home]/user/Ferret/defaults/defaultFerretSetup.m). These parameters, copied from defaultFerretSetup.m, are as follows:

```
% History
par.history.dataDir='FerretData';
par.history.NGenPerHistoryFile=25;
par.history.saveFrac=0;
```

The `par.history.dataDir` field specifies the data directory for your run. The data directory will contain several subdirectories, including the History folder [par.history.dataDir]/History, which holds the History files that are required for analysis (see Section 4.15). If `par.history.dataDir` is specified as a local path (i.e. `par.history.dataDir='FerretData'`), then the data directory will be generated as a sub-directory inside of your project directory, which is specified either by your launchQubist.m file (see Section 4.4.1) or the QubistGlobalProjects.m file (see Section 4.4.2).

If you are performing many runs and want to keep them separate from your program files, then it is best to specify the data directory by giving a global path to a data directory outside of your project directory. In either case, it is important to ensure that your file permissions allow you to create and write to the specified directory. If you try to run Ferret and you do not have sufficient file permissions, then you will see an error dialog box like the one shown in Figure 4.10: .

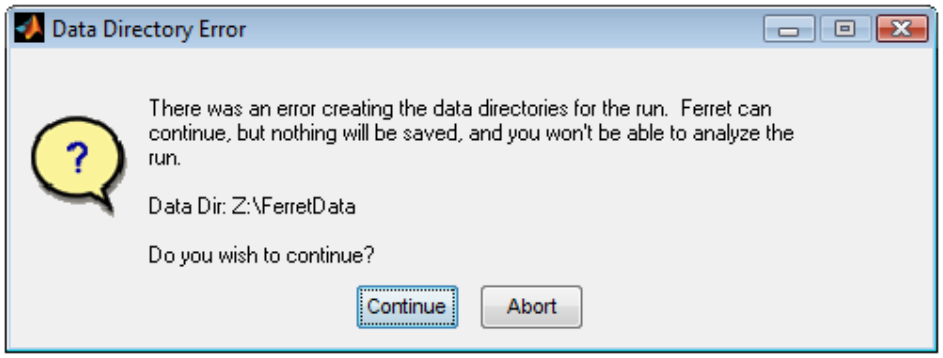

Figure 4.10: Dialog box indicating that the data directory could not be created. In this example, I have set my data directory to a global path 'Z:\FerretSetup', which generates an error because I do not have a disk 'Z:' on my Windows system.

The dialog box indicates that there was a problem generating the run's data directories, including the one that normally contains History files. You can continue if you wish, but you won't be able to analyze your run, because analysis requires the History files that aren't being written. This error can be caused by an invalid path or insufficient file permissions to create the directory. If you see this, you should probably click 'Abort' and fix the problem, rather than continuing. If you continue, you will be able to watch the optimization of your project in the console window, but without the History files, you will not be able to analyze your run or compute the final optimal set. This may be of limited use for testing things early in

the development of a project, but really you should just stop and fix the path or permission problem if you see this error dialog.

MATLAB users who do long-duration calculations know all too well that it is possible to crash MATLAB occasionally, whether it is due to an internal segmentation fault, a Java exception, or some external reason like a power failure or operating system glitch. Crashes during long runs are extremely serious, and I have placed a great deal of emphasis on making Ferret's crash recovery relatively painless. So when your three-month long Ferret run crashes for some unexplained reason, please remember these two important things:

- *Don't panic.* You will be able to resume your run from the last History file that was successfully written to disk, using the method discussed in Section 4.17. Very little data will be lost, if any.
- *It's not my fault.* Well, in reality, I suppose it could be - Ferret is a big machine with a lot of moving parts, and users occasionally find surprising ways to break things. But at least I've done my best to make your crash recovery relatively painless, and your actual data loss will be minimal. In addition, I would be grateful for a bug report with all error messages and perhaps a screenshot, if you really think the problem occurred in Ferret and not your own code.

Two of the options in `par.history` affect your ability to recover from crashed runs, and I turn to these settings next.

The field called `par.history.NGenPerHistoryFile` specifies how many generations to include in each History file. The default, which is specified in defaultFerretSetup, is normally set to about 25, but this parameter can be changed in the setup file to suit your problem. You should consider the following when deciding on a value for this parameter:

- If $NGenPerHistoryFile$ is a small number (< 25), then fewer generations will be included in each History file and you will generate more of them. This will probably use slightly more disk space, and your analysis will be slower because you will have more files to load. On the other hand, if $NGenPerHistoryFile$ is set to a larger number (> 25), then you will generate fewer History files, use slightly less disk space, and be able to analyze your runs more quickly.
- Lower values of $NGenPerHistoryFile$ allow you to recover more quickly from a crash than higher value, because Ferret will restart from the solutions contained in the last History file that was successfully saved.
- History files can sometimes include a large amount of information, especially if you are loading a lot of data in your init file without using the 'noSave' feature discussed in Section 4.6. If you use a large value for $NGenPerHistoryFile$, you may require more memory to analyze your run, since information will be loaded from disk in larger chunks. I therefore recommend lower values for systems that do not have much memory. You should also consider using the *noSave* feature if this is a problem.

My advice is to use the default value for $NGenPerHistoryFile$ or slightly larger if your fitness function can be evaluated rapidly, but to set it to a smaller value if your fitness function takes a long time to evaluate. Generally, I get annoyed if I lose more than a couple hours of run time during a crash, so I typically set $NGenPerHistoryFile$ to a rather small value (5-10) for my most computationally intensive projects. I rarely set this parameter to a value much greater than 25 because the benefits of doing so are minimal.

The `par.history.saveFrac` field controls how much information is saved in your History files. By default, `par.history.saveFrac=0`, which indicates that only the optimal solutions from each generation should be saved. This saves enough information to compute the global optimal set from all generations at the end of the run, and this is therefore the most common setting for this parameter. However, it is possible to save the optimals plus some non-optimal members of the population if you set *saveFrac* to a value between 0 and 1. In this case, *saveFrac* indicates the minimum fraction of the population to be included in the History files. For example, if `par.history.saveFrac=0.25`, then the History files will contain the *intersection* of the optimal set and the best 25% of the population. If the optimals account for greater than 25% of the population, then all optimals will still be saved. It is reasonable to ask why you might want to consider saving more solutions that just the optimals at each generation. There are two possible reasons:

- If your run should crash, it can be resumed from the last saved History file. If $saveFrac < 1$, then only a fraction of the population is saved in each History file, and it does not contain enough information to fully re-populate the *world* from the last saved generation. In this case, missing individuals are set to random solutions, which are likely to be poor, within the search space. This is not a major problem in practice, since the best solutions are always saved in the History files, but it will take Ferret a few generations to improve these relatively poor solutions by crossovers with the saved optimals. If you choose to save all of your solutions or a large fraction of them, this won't be necessary and recovery will be faster.

- Ferret, or the Qubist Movie Tool (see Section 12.6) can show movies of your run by loading History files and re-generating the appearance of the Ferret console window at any previous generation. However, movie frames produced in this way only contain solutions that were actually saved in the History file. Therefore, your movie will only show the optimals if you use `par.history.saveFrac=0`, which is good enough for most purposes. However, you should always set `par.history.saveFrac=1` if you want to generate a high-quality movie that shows the evolution of the entire population.

- The analysis interface allows you to modify the fuzzy tolerances on the optimal set (`par.selection.FAbsTol > 0` or `par.selection.FRelTol > 0`), *even after the completion of a run.* This can be used to slightly blur the boundaries of the optimal set, which tends to fill it out and sometimes better shows its structure. This feature can also help you to understand your parameter error in a χ^2 minimization problem. However, if you increase the tolerance, only solutions actually stored in the History files can appear. If you think your problem might benefit from the flexibility allowed by modifying fuzzy tolerances after the completion of the run, then it may be beneficial to set `par.history.saveFrac > 0` so that some non-optimals are included in the History files.

In practice, I normally run Ferret with `par.history.saveFrac=0`, because this minimizes disk usage. I do not recommend increasing *saveFrac* just to improve your ability to recover after a run, because this recovery will occur over just a few generations anyway, and the introduction of diversity into the population from some random solutions might even be beneficial. Ferret actually includes an optional operator called 'superMutation' (see Section 8.7), which wipes out the entire group of non-optimals over all populations every now and then, for exactly this reason. A crash and subsequent recovery with `par.history.saveFrac=0` is roughly equivalent to the application of this superMutation operator. Anecdotally, I only added the superMutation operator after noticing rapid improvement of a previously stable population after such a crash! In my opinion, you should only consider crash recovery time when setting *saveFrac* if your fitness function is computationally very expensive. The ability to make better movies or effectively add fuzzy tolerances to the optimal set at the end of a run are much better reasons to use $saveFrac > 0$.

4.16.2 Problems Encountered with Very Large History Files

An especially confusing problem arises occasionally when users run projects that involve very large numbers of parameters (hundreds or thousands) or require very large data structures, which may result in truly enormous History files. By default, MATLAB saves files using `save -v7` for backward compatibility with all version-7 MATLAB releases, which imposes a 2 Gb limit on file sizes. If Ferret tries to save a History file that exceeds this limit, you may see a warning like the following:

```
Warning: Variable 'History' cannot be saved to a MAT-file whose version is older than 7.3.
To save this variable, use the -v7.3 switch.
Skipping...
```

If you ignore this warning, you may be quite confused (and maybe a little angry) at the end of your run when you discover that all of your History files are empty and that you can't analyze your run. Assuming that your MATLAB version is fairly new (R2006b or later), the quick fix for this problem is to change your MATLAB MAT-file preferences for compatibility with version 7.3 and later, which allows files that exceed the 2 Gb version-7 limit. In my R2009b MATLAB version, this option is found under File >> Preferences >> General >> MAT-Files.

A better solution is to determine *why* your History files are so large, and make them smaller if possible. The 'noSave' option discussed above helps to decrease the size of History files if their large size is due to data structures loaded by your init file, but this technique will not result in significant improvement if the problem is caused by saving large parameter sets. For example, if you are optimizing a large 'non-parametric problem' (see Section 4.20) with $P = 500$ parameters, saving the entire population of 500 each generation (`par.general.popSize=500; par.history.saveFrac=1`), and have specified 100 generations per History file (`par.history.NGenPerHistoryFile=100`), then the amount of parameter data *alone* in each History file is given by

$$NGenPerHistoryFile \times popSize \times P \times 8 = 2 \times 10^9 \text{ bytes} = 2 \text{ Gb}. \tag{4.2}$$

Your History files will exceed the 2 Gb MATLAB version 7 limit once you add the other data included in each History file. Obviously, this problem only arises for a very specific type of problem, but this issue really has resulted in bug reports. The best solution is this case is to reduce $NGenPerHistoryFile$, and/or $saveFrac$. This will also have the pleasant side effect of freeing up memory while your project is running, which may improve performance.

4.16.3 Crash Recovery - General Comments

This section has spent some time discussing recovery after a crash. At this point, you are probably wondering how often this occurs, and how important these crash recovery features really are. The answer really depends on two factors: the operating system that you are using and the real-time duration of your runs. In practice, I have found that MATLAB is very stable under both the Windows and MacIntosh operating systems, especially when running without a lot of user interaction, and actual MATLAB crashes are infrequent. I have experienced many MATLAB segmentation faults under Linux, however, so you may wish to save your History files more often if you are a Linux user. These problems were eventually traced to a slight, and as far as I know undocumented, incompatibility between my MATLAB version and the glibc library installed on my system. MATLAB would often run for days on my system without a problem, and then crash. Segmentation faults, which kill the entire MATLAB session, are problems (or

incompatibilities) with MATLAB itself, and are not caused by errors or bugs in Ferret or your application. They are to be distinguished from less disruptive bugs in your fitness function, or perhaps Ferret[6], which produce error messages on the command line without causing MATLAB to exit.

Other types of problems can occur if your problem requires very long run times. I use Ferret on some problems that can take more than a month to run, and in my experience, it is difficult to run for a month without requiring a system re-boot, a network disk coming disconnected, a power interruption, or some other random event that requires a re-start. A good crash recovery scheme is essential for long-duration runs, and this has been a core feature of Ferret since its earliest days. If a crash does occur during a run, don't worry about it, because Ferret is designed to let you resume your run with minimal effort and information loss. Procedures for resuming a run are outlined in Section 4.17.

4.16.4 Anatomy of a History File

History files contain detailed information about the progress of a Ferret run at each generation. These files are used internally by Ferret during the analysis of a run, or to resume a run that has crashed or been suspended. Most users do not directly load or interact with History files, because the final results from a run are encoded in OptimalSolutions.mat and not History files. Nevertheless, I will outline the structure of a History file for the occasional curious user who wants to examine them.

History files are numbered in the order that they are saved, so you will find files named History-1.mat, History-2.mat, etc. in the History subdirectory of your run's data directory, which is determined by `par.history.dataDir`. Loading one of these files into MATLAB's workspace results in a MATLAB cell array called History (*not* History-1, History-2, etc.), where the size of the cell array corresponds to `par.history.NGenPerHistoryFile`. Each cell represents a Ferret generation, and you can determine the generation number from `History{`*n*`}.par.bookKeeping.gen`, where n is the index of a cell. The detailed structure of a single History cell is as follows:

[6]If you think you have encountered an error that is caused by a bug in Ferret, please consider reporting it to nQube using the contact information provided in this user's guide. Please also save the exact error trace from the command line and provide as much information as possible about what you were doing when the error occurred. Sample code that reproduces the error is especially useful for diagnosing and fixing such problems.

```
History{n}
          |
          .graphics{p}   % Graphics information for each population p, as
          |           |  % displayed in the upper left axes of the Ferret console.
          |           |
          |           .popStats      % Matrix used to encode the values shown in the
          |           |              % graphics axes of the Ferret console window.
          |           |              % Includes all generations up to the current one.
          |           |
          |           .FOptimalSet   % Contains information about the minimum and
          |                       |  % maximum  values of the fitness function.
          |                       |  % Includes all generations up to the current one.
          |                       |
          |                       .min  % Matrix of minimum values.  Each row
          |                       |     % corresponds to an objective, and columns
          |                       |     % correspond to the generation number.
          |                       |
          |                       .max  % Matrix of maximum values.
          |
          .pop{p}        % Information about population number p. (Recall that Ferret
          |      |       % can be configured to run with multiple populations that
          |      |       % only interact weakly through the 'immigration' operator.
          |      |
          |      .indiv    % Structure containing information about the individuals in
          |      |         % the population, their parameters, and fitness values.
          |      |         % Most fields in world.pop{p}.indiv are also present here.
          |      |
          |      .BBList   % Cell array of active building blocks for each individual.
          |      |
          |      .ranking  % Ranking information for each individual.  Similar to
          |      |         % world.pop{p}.ranking.
          |      |
          |      .elites   % The elites that were present at this generation in the
          |                % run.  Similar to world.pop{p}.elites.
          |
          .trackers      % Information used internally by Ferret for auto-adaptation
          |              % of strategy parameters.
          |
          .link          % Information used internally by Ferret's linkage-learning
          |              % system.
          |
          .par           % The full par structure returned by FerretSetup.  The
              |          % generation number is in par.bookKeeping.gen.
              |
              .user.extPar  % The external parameters from the user's init file.
```

Each History cell n contains a cell array called `History{`n`}`.graphics that contains convergence information

for all populations, which is displayed in the Ferret console window. `History{n}.graphics{p}` is a matrix with two rows that correspond to the plots displayed at generation `History{n}.par.bookKeeping.gen` in the top left panel of the Ferret console, as shown in Figure 4.2 and discussed in 4.12. The number of columns in `History{n}.graphics{p}` corresponds to the generation number. The plot for generation `History{n}.par.bookKeeping.gen` and population p can be reproduced by the command:

```
plot(History{n}.graphics{p}.popStats');
```

I have used this technique occasionally with older versions of Ferret to generate a graph showing the code's convergence for the entire run. In this case, I would load the highest numbered History file in my projects data directory, examine the History cell array, and set n to the number corresponding to the last non-empty cell. I would then run the above code to generate a plot. *Note that the current version of Ferret makes this information more convenient to extract, since it is copied into the OptimalSolutions.mat file, as described in Section 4.18.*

`History{n}.pop` is a cell array that contains the saved state of all populations at generation `History{n}.par.bookKeeping.gen`. The structure of this cell array is very similar to `world.pop{p}` (see Section 4.9.1), with the notable difference that only current optimal set at each generation is saved by default. The behaviour can be overridden by setting `par.history.saveFrac=1`, as discussed in Section 8.2, if you require non-optimals to be saved. Note, however, that this may increase the size of the History files substantially.

4.17 Resuming a Run

Resuming a crashed or suspended Ferret run is easy because the History files save enough of the state of the *world* - Ferret's top-level data structure - to regenerate *at least* the state of the optimal solutions, even when resuming from a crash that did not perform any cleanup operations. The entire state of the *world* is restored if Ferret stopped in an orderly fashion. The easiest way to resume a run is just to point Ferret at the project's original data directory, by setting `par.general.dataDir` in the setup file, and re-start it. Ferret will notice that the directory is not empty and will present you with the dialogue box shown in Figure 4.11. To resume your run, just click the 'Resume' button and your run will be re-constructed automatically from the saved History files!

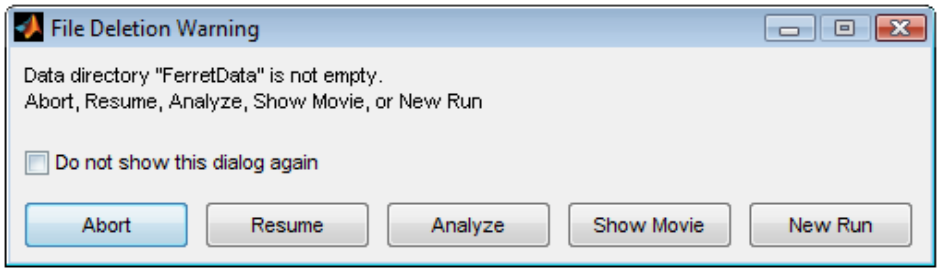

Figure 4.11: Dialog box indicating that the data directory already exists. Note the option that allows users to resume their run from the saved History files.

.

This technique is very convenient for resuming runs, but it does not work if the project or the History files have been moved, which actually comes up more often that you might think. For example:

- If you are in the habit of moving files to archive them, and you suddenly feel the need to restart an old run.
- If you need to migrate a long-duration run to a different computer that does not share the same file system.
- If you work with colleagues or students and send them your History files, and they need to restart your run for some reason.
- If your project is coded such that it names the run automatically using the date, and you need to restart it at a later date.

I have personally encountered all of these situations, and I developed the Qubist Resume Tool out of necessity. The Resume Tool is started from the component selector by running the launchQubist.m file and is very easy to use. It will guide you through a series of dialogue boxes that are designed to load the last saved History file of your project from disk, reconstruct Ferret's *world* data structure, set all necessary paths for the run, and start Ferret right where you left off with minimal effort. The Resume Tool also works for Locust runs, and is discussed in Section 12.1.

4.18 The OptimalSolutions.mat File

The OptimalSolutions.mat file is the single most important Ferret-generated file that users deal with, because this file is produced when a run is analyzed, and encapsulates all of the final results from the run. It is important to understand the structure of an OptimalSolutions.mat file because many problems require post-processing of solutions by other problem-specific MATLAB functions, and this is the file containing the results that you would send to your post-processing code. OptimalSolutions.mat contains a MATLAB structure that is organized as follows:

```
OptimalSolutions
                |
                .X          % Matrix of parameter values: each column represents the
                |           % parameters of a single solution.  The number of rows
                |           % equals the number of parameters, and the number of
                |           % columns equals the number of solutions.
                |
                .XCPD       % Matrix of parameter values, including CPD information:
                |           % each column represents the parameters of one solution,
                |           % but may contain NaN values.  These are parameters that
                |           % were treated as unspecified by Ferret's CPD system.  XCPD
                |           % is the same size as X, and is rarely needed by users.
                |
                .F          % Matrix of fitness values: each column represents the
                |           % fitness of a single solution.  The number of rows will
                |           % equal the number of objectives.
                |
                .rank       % A row vector containing the true rank of each solution,
                |           % without regard to fuzzy tolerances (FAbsTol, FRelTol,
                |           % etc.).  The best solutions have the lowest rank.
                |
                .auxOutput  % Cell array of auxOutput values returned by the fitness
                |           % function.  This will be an empty cell array {} if
                |           % auxOutput is not returned.  When used, each cell index
                |           % corresponds to its respective columns in X and F.
                |
                .std.X      % Column vector: standard deviation of each parameter
                |           % over all optimal solutions.
                |
                .LM         % The linkage matrix, as displayed by the Ferret console
                |           % and the analysis window.
                |
                .graphics{p}   % Graphics information for each population p, as
                |           |  % displayed in the top left and top right axes of the
                |           |  % Ferret console.  Contains all generations up to the
                |           |  % current one
                |           |
                |           .popStats      % Matrix used to encode the values shown in
                |           |              % the top left graphics axes of the Ferret
                |           |              % console window.
                |           |
                |           .FOptimalSet   % Structure that contains information about
                |                       |  % the min and max values of the fitness
                |                       |  % function(s).  Shown in the top right axes.
                |                       |
                |                       .min  % Matrix of minimum values.  Each row
                |                       |     % corresponds to an objective.  Columns
                |                       |     % correspond to the generation number.
                |                       |
                |                       .max  % Matrix of maximum values.
```

```
OptimalSolutions (continued)
                 |
                 .par           % The full par structure from the setup file.
                 |   |
                 |   .user.extPar  % The external parameters returned by the init file.
                 |
                 .benchmark     % Useful information for benchmarking.
                           |
                           .gen    % Vector of sequential generation numbers
                           |
                           .NEval  % The total number of solutions evaluated by
                           |       % the fitness function.
                           |
                           .time   % Numerical times corresponding to when each
                           |       % generation finished.  The format is the same
                           |       % as MATLAB's 'now' function.
                           |
                           .timeString  % The corresponding human-readable
                                        % formatted time strings.  The format is
                                        % the same as MATLAB's 'datestr' command.
```

It is apparent from the above tree diagram that an OptimalSolutions.mat file is actually quite simple, with most of the important data that a user would normally access organized as a nearly flat structure with few branches. The most important fields are `OptimalSolutions.X` and `OptimalSolutions.F`, which contain the parameters of all solutions in the optimal set and their fitness values. `OptimalSolutions.X` is a matrix of size $[NGenes \times N]$, where $NGenes$ is the number of parameters (or 'genes'), and N is the number of solutions. Thus, each column j of `OptimalSolutions.X` represents the parameters of the j^{th} solution in the optimal set. `OptimalSolutions.F` is a matrix or row vector of size $[NObj \times N]$, where $NObj$ is the number of objectives. Column j of `OptimalSolutions.F` represents the fitness values for solution j.

`OptimalSolutions.XCPD` is a matrix of *scaled* parameter values, of the same size as `OptimalSolutions.X`, which contains information from Ferret's critical parameter detection system (CPD; see Section 2.2.6) about which parameter values were fully specified or unspecified when the solution was discovered. Internally unspecified parameters are flagged as NaN in `OptimalSolutions.XCPD`, but the *actual* values used to compute `OptimalSolutions.F` are stored in `OptimalSolutions.X`, which should not contain NaN values. Note that the difference `OptimalSolutions.X - OptimalSolutions.XCPD` is always a matrix containing only zeros and NaN values. `OptimalSolutions.XCPD` is rarely accessed by users.

`OptimalSolutions.rank` is the true rank of all solutions within the optimal set, without regard to fuzzy tolerances `par.selection.FAbsTol`, `par.selection.FRelTol`, and `par.selection.FRankTol` (see Sections 5.1 and 8.6). Low rank numbers correspond to better solutions than high rank numbers, and solutions with `OptimalSolutions.rank=1` are the very best solutions that were found. It is likely that not all solutions in the optimal set will be of $rank = 1$ if fuzziness was enabled during analysis (Section 5.6) or during the run (section 8.6).

`OptimalSolutions.auxOutput` is a cell array containing the *auxOutput* data for the optimal set. Recall that *auxOutput* is an optional field of 'auxiliary data' returned by the user's fitness function, as outlined in Section 4.8. Cell j contains the auxOutput data for solution j.

`OptimalSolutions.std` is a column vector that represents the standard deviation of each parameter across the entire optimal set. You can use Ferret's built-in visualization tools to examine the detailed distribution of parameters in the optimal set, but this field gives the user a rough idea of how spread out the parameter values are.

`OptimalSolutions.LM` is the linkage matrix that is plotted in the Ferret console (see Figures 4.2 and 5.1), or in the analysis window (See Section 5.2 and Figure 5.2 for an example.). You can also plot the cumulative linkage matrix for your run in a separate window by typing the following commands at the MATLAB prompt:

```
imagesc(OptimalSolutions.LM);
colormap bone;
axis image;
```

`OptimalSolutions.graphics` contains convergence information for each population, and can be used to reproduce the final graphs from the top left and top right axes in the Ferret console window. `OptimalSolutions.graphics` is identical to the corresponding structure from the last generation present in the last saved History file. The details of `OptimalSolutions.graphics` are discussed in Section 4.18. A graph showing the convergence of the run for all populations can be generated with the following commands:

```
for p=1:length(OptimalSolutions.graphics);
     % Loop over populations.
     plot(OptimalSolutions.graphics{p}.popStats'); hold on;
end
hold off;
```

`OptimalSolutions.par` is the full *par* structure generated by the project's setup file. The external parameters generated by the init file are in `par.user.extPar`.

`OptimalSolutions.benchmarking` contains useful information for benchmarking your run. The following lines of code show how to plot the number of function evaluations and the time as a function of the generation number:

```
figure(1);
subplot(2,1,1);
plot(OptimalSolutions.benchmark.gen, OptimalSolutions.benchmark.NEval);
ylabel('# of evaluations');
subplot(2,1,2);
%
startTime=OptimalSolutions.benchmark.time(1);
time=24*3600*(OptimalSolutions.benchmark.time-startTime);
plot(OptimalSolutions.benchmark.gen, time);
xlabel('generation');
ylabel('time (seconds)');
```

4.19 Automatic Post-Processing of Results

Many projects require some post-processing and analysis of the solutions contained within the OptimalSolutions.mat file. Post-processing is usually done manually after a run is complete by loading the

OptimalSolutions.mat file into the user's post-processing code. However, Ferret includes a feature that allows post-processing code to be run *automatically* whenever an analysis is done. To enable this feature, you must write a function that accepts *OptimalSolutions* as an argument, or requires no arguments:

```
function myPostProcessingCode(OptimalSolutions)
function myPostProcessingCode
```

This function should not return any arguments, since they will be ignored by Ferret. Therefore, it is up to the post-processing function to save any results to disk, or to a global variable, before the function exits. *Note that this must be a function; scripts are not allowed and will cause an error message.* Two common paradigms for post-processing involve looping over all solutions in *OptimalSolutions* for further analysis, or selecting the 'best' solution for single-objective problems. The following code snippets illustrate these two techniques:

```
function myPostProcessingCode(OptimalSolutions)
global myResults

% To loop over entire optimal set:
for n=1:size(OptimalSolutions.X, 2) % Number of columns = number of solutions.
        X=OptimalSolutions.X(:,n);
        F=OptimalSolutions.X(:,n);
        <<< USER CODE: SHOULD WRITE TO DISK OR SET GLOBAL VARIABLE myResults >>>
end

% ------------------------------------------------------------------------------
% To select the 'best' solution in the optimal set, for single objective problems
% where fuzziness is enabled via par.selection.FAbsTol, par.selection.FAbsTol, or
% par.selection.FRankTol:
[FMin,n]=min(OptimalSolutions.F);
n=n(1); % In case there is more than one optimal.
X=OptimalSolutions.X(:,n);
F=OptimalSolutions.X(:,n);
<<< USER CODE: SHOULD WRITE RESULTS TO DISK OR TO GLOBAL VARIABLE myResults >>>
```

The name of the post-processing code is arbitrary, but must be indicated in the Ferret setup file as follows:

```
par.analysis.postProcess='myPostProcessingFunction';
```

Note that an empty string is the default setting: `par.analysis.postProcess=''`, which indicates that no automatic post-processing is required. See Section 8.16 for details on par.analysis.

Automatic post-processing is most useful when combined with a custom launch script as discussed in Section 4.14.2, which can be configured to run Ferret multiple times (perhaps using different data sets), do the analysis for each run, and run any post-processing code without user input. To perform these latter two steps automatically, the Ferret setup file must contain lines of code like the following:

```
par.analysis.analyzeWhenDone=true;
par.analysis.postProcess='myPostProcessingFunction';
```

Naturally, 'myPostProcessingFunction' should be replaced with the actual name of the user's post-processing code, which should be on the project's path. Please refer to Section 8.16 for more analysis

options.

4.20 Non-Parametric Problems

We close this chapter with some brief comments regarding so-called 'non-parametric' models, which do not assume a specific mathematical form for the underlying model. A simple example is given by the demo 'Advanced/Non-Parametric-Image-Modeling' which adapts about 600 pixel values of a model image until they agree with a real image, by minimizing the χ^2 statistic. The solution is trivial of course, since I could just assign the actual, known pixel values to the model, but I perform a full χ^2 minimization to illustrate this class of problems in general. The term 'non-parametric' model is somewhat of a misnomer because such problems often have several hundred parameters, which makes them - from a computational point of view - the most 'parametric' of all. They are non-parametric in a mathematical sense that they do not impose a specific functional form on the solution, as is typical of most other data-modeling problems.

Non-parametric models may require some special handling to find solutions efficiently. It is important to realize that such problems have extremely large linkage matrices (600×600 for the non-parametric image modeling demo), and as such, there is not much point in trying to find all of the linkages. Therefore, I normally set the initial linkage to some reasonable value (less than about 0.5) and turn linkage learning off:

```
par.link.PLink=0;
par.link.initialLinkage=0.25;
```

Non-parametric models work best when the parameters are uncorrelated, like the pixels in my demo, so this is really not a huge restriction.

It is *very important* to turn off niching acceleration (see Section 8.9), and auto-adaptation of this strategy parameter, for very large non-parametric problems:

```
par.niching.acceleration=0;
par.strategy.adapt.niching.acceleration=false;
```

This is necessary for efficiency, since niching acceleration is extremely expensive when the number of parameters is large. The default setting, with the acceleration turned on, would require much more overhead than it is worth.

Finally, I have noticed that restricting mutations and crossovers to single building blocks (discovered by linkage learning) seems to allow Ferret to solve non-parametric problems more rapidly than the defaults settings, where Ferret chooses automatically whether to restrict crossovers to individual building blocks or do unrestricted crossovers over all parameters. In effect, building block restriction means that any given mutation or crossover only involves a single building block, which would include about 25% of the parameters if `par.link.initialLinkage=0.25`, as in the example settings above. Building block restriction is enabled for mutation and crossover by the following lines in the setup file:

```
par.mutation.BBRestricted=true;
par.XOver.BBRestricted=true;
```

Please refer to Sections 8.7 and 8.8 for additional comments about these settings.

The demo 'Advanced/Non-Parametric-Image-Modeling' contains a FerretSetup file that you can use as a template for non-parametric models. There are a few other slightly atypical settings there, such as no auto-adaptation, a relatively high mutation rate, and a low elite fraction. However, these settings are not

as critical as those described above and you should feel free to modify them to suit your application.

Chapter 5

Visualization of Solution Sets

'We shall not cease from exploration. And the end of all our exploring will be to arrive where we started and know the place for the first time.'

-T. S. Eliot, *Little Gidding*

Section 2.2 offered an anecdote from immediately before Ferret's inception in 2002, when I attempted a difficult multi-parameter data modeling problem, and failed miserably using simple grid and random search techniques. This resulted in some terribly confusing plots, and I understood for the first time - I mean really understood - that the ability to visualize multi-dimensional solution sets effectively is almost as important as being able to find them in the first place. Ferret has had limited graphical capabilities since Ferret-1, but visualization was elevated to the status of a Qubist core feature midway through the development of Ferret-3.

Qubist's visualization system is not really all that complex, but it is quite effective at helping users to find interesting features and understand multi-dimensional structures in their parameter spaces. This section discusses the visualization options that are integrated with the analysis window used by both Ferret and Locust, and how you can use these features to tease information out of complicated optimal sets.

My academic field of research is in astronomy, so it seems fitting to demonstrate Qubist's visualization features by means of a data-modeling example chosen from this field. I have chosen this example because the underlying model is quite simple, but has enough mathematical structure to illustrate some of the features of more complicated data-modeling problems. Moreover, it provides a good demonstration of Qubist's visualization capabilities, while also showing how simple visualization techniques can reveal features in parameter space that point the way to a better understanding of and model's mathematical structure. The MATLAB m-files for this demonstration are included in the Qubist demos, under the name 'Advanced/Spectroscopic-Binaries' in the Demos menu. The example uses Ferret, which performs better than Locust on this problem, but Locust's visualization features are almost identical.

5.1 Spectroscopic Binaries: Visualization as a Knowledge Discovery Tool

I teach a third year course on the physics of stars, in which we cover many topics including radiative transfer in stellar atmospheres, nuclear fusion, convection, solar physics, stellar evolution, supernovae, and exotic stars like white dwarfs and neutron stars. One of the topics that I cover early in the course is binary star systems - pairs of stars that are in orbit around a common centre of mass. There are many different kinds of binaries, but this example will focus on spectroscopic binaries, where the stars are too distant to actually be seen moving around on their orbits, but the motion along the line of sight can be observed by the Doppler shift of the starlight. Doppler shift is a familiar phenomenon to anyone who has ever heard the change in pitch of a car horn as a vehicle passes. The horn sounds higher in pitch when it approaches you, and then abruptly decreases in pitch as the car passes. Likewise, the light from a star appears higher in frequency when it moves towards the observer, and therefore any spectral lines are shifted toward the blue. The opposite happens when the star moves away from the observer, when spectral lines are shifted towards the red. Therefore, even if we can't see the stars in a spectroscopic binary system moving back and forth on the sky, we can obtain some information about the state of their motion by observing how their spectral lines shift back and forth in frequency as the stars go about their orbits. This information is incomplete because we only see the effects of their motion in one dimension along the line of sight, and we know *nothing* about the motion in the other two dimensions.

Spectroscopic binaries are interesting because their observed Doppler shifts can reveal information about the masses of the stars and their orbital parameters. A question that always arises when I teach this topic is the following: 'if we only know the motion of the stars in one dimension by measuring the Doppler shift, then what information can we actually obtain about their masses and the orbital parameters'? Spectroscopic binaries have been studied for a long time and the answer to this question is well known. The system is simple enough that a bright student could probably even figure this out from the structure of the equations. However, I find it more effective from a teaching perspective to address this question by means of the data-modeling simulation that I show here, rather than attempting to address it from a purely mathematical analysis. This is more realistic for data-modeling problems in general, because it is rarely possible to determine the mathematical degeneracies and overall parameter space structure of a more complicated model *a priori*. However, it is often easier to find the physical reasons behind such features by mathematical analysis, once Ferret shows you where to look in the math!

We start off by generating 'artificial data' by simulating the orbits and calculating the velocity toward or away from the observer as a function of time, to which I add random Gaussian noise at a level equal to 5% of the maximum velocity magnitude. This artificial data is shown as the red jagged line in panel d) of Figure 5.2. In reality, it is possible to constrain the mass of a star by measuring its spectral type, but we will assume a broad range of masses from 0.1 to 10 times the mass of the Sun, for the purpose of this demonstration. Why use artificial data rather than real data? First of all, I don't actually have any real data for binary star systems. Secondly, and more importantly, *using artificial data constructed with known parameters and added noise offers an excellent way to test a modeling code, because a correct code should produce an optimal set of models whose parameters are distributed about the original artificial data model.* Of course, the inherent danger of this method is that the model itself could be wrong or there could be an error in the code that models it, in which case you might blindly go forward fitting incorrect models to junk data. However, the physics behind this particular model is simple enough that I'm certain that it is correct. The artificial data is produced by the problem's init.m file:

```
function extPar=init
% Initialization function for the spectroscopic binary modeling demo.
%
% Load physical constants.
extPar.K=const;
%
% Prepare artificial data using the following parameters:
extPar.binaryPar.phi=60; % orbital phase (degrees): rotates the orbital plane
extPar.binaryPar.sini=sin(pi/180*45); % inclination (degrees): tilt wrt. plane of ↘
        ↪ the sky.  i=0 ==> orbit in the sky plane.
extPar.binaryPar.e=0.5; % eccentricity: how elliptical are the orbits?
extPar.binaryPar.m1=1; % mass of star #1 (solar mass units)
extPar.binaryPar.m2=2; % mass of star #2 (solar mass units)
extPar.binaryPar.a=1; % semi-major axis of the orbits in AU (astronomical units):
%   --> 1/2 of the long dimension of the elliptical orbits.
%
% Options for the numerical integrator:
extPar.int.absTol=1e-8; % Absolute tolerance
extPar.int.relTol=1e-8; % Relative tolerance
%
% Captions for plotting:
extPar.etc.tCaption='t (yr)'; % x-axis
extPar.etc.vCaption='v (km/s)'; % y-axis
%
% Convert orbital parameters to cgs units:
a=extPar.binaryPar.a*extPar.K.AU; % semi-major axis
m1=extPar.binaryPar.m1*extPar.K.Msol; % Mass #1
m2=extPar.binaryPar.m2*extPar.K.Msol; % Mass #2
%
% Period (from Kepler's Law):
extPar.etc.P=sqrt( 4*pi^2*a^3/extPar.K.G/(m1+m2) ); % Period (seconds)
%
% Prepare artificial data.
extPar.data.Npt=100; % Number of data points requested
extPar.data.t=linspace(0,extPar.etc.P,extPar.data.Npt); % Times at which data are taken
extPar.data.relErr=0.05; % Relative error for added noise
[extPar.data.t, extPar.data.V]=binary(extPar); % Creates the "data"
[extPar.data.V, extPar.data.dV]=addNoise(extPar.data.V, extPar.data.relErr); % Add ↘
        ↪ some Gaussian noise
```

Parameter #	Symbol	Parameter Name	Value	Units	Range
1	ϕ	orbital phase angle	30	degrees	[0, 360]
2	$\sin(i)$	sin of inclination angle	$\sqrt{2}/2$	dimensionless	[0, 1]
3	e	eccentricity	0.7	dimensionless	[0, 0.9]
4	$\log_{10}(m1)$	log of mass of star #1 [1]	$\log_{10} 0.5$	dimensionless	[-1, 1]
5	$\log_{10}(m2)$	log mass of star #2 [1]	$\log_{10} 2$	dimensionless	[-1, 1]
6	$\log_{10}(a)$	semi-major axis [2]	2	AU	[-1, 1]

Table 5.1: The actual parameters used to construct the artificial data for the spectroscopic binary problem. 5% noise was added, and the code was tested by attempting to recover these parameters by modeling the artificial data.

[1] Masses are relative to the Sun (solar mass units).
[2] The semimajor axis is in units of astronomical units (mean distance between the Earth and Sun).

The fitness function receives all of the parameters in extPar, which I have organized into several substructures.

- `extPar.K`: physical constants loaded from the const.m file, which is included in the demo directory.
- `extPar.binaryPar`: all of the orbital parameters that Ferret will (hopefully) solve for.
- `extPar.int`: Tolerances for the ODE integrator.
- `extPar.etc`: Captions for plotting. This field also includes the period of the orbit, which is calculated using Kepler's Law.
- `extPar.data`: The artificial data that we plan to model. Velocities for the two stars are contained within the two rows of matrix `extPar.data.V`, as a function of the times in vector `extPar.data.t`. These same times will be used for all models calculated during the Ferret run, because the reduced χ^2 function given by equation 5.1 below requires that the model being tested is evaluated at exactly the same times 'observed' in the artificial data.

The actual parameters used to generate the artificial data for this example are given in table 5.1, and panel d) of Figure 5.2 shows plots of the velocity vs. time data, including the added noise, for both stars. Note that the logarithm of the mass is used as a parameter, rather than the mass itself, because we plan to explore a large dynamic range of possible masses during the parameter space search. Using the sine of the inclination angle as a parameter, instead of i, provides a slight simplification because the inclination angle enters the orbit equations as $\sin i$.

The fitness function is a reduced χ^2 function that compares the velocity-time functions for each candidate model to the artificial data:

$$\begin{aligned} F &= \chi^2_{reduced} = \frac{1}{D.O.F.} \sum_{s=1}^{2} \sum_{i=1}^{N} \frac{(v_s^{model}(t_i) - v_{s,i}^{data})^2}{\sigma_{s,i}^2} \\ D.O.F. &= 2N - 6 \end{aligned} \tag{5.1}$$

where s is an index over the two stars, $v_{s,i}^{data}$ is the sequence of velocities observed at times t_i, and the model velocity $v_s^{model}(t_i)$ is evaluated at these same times. The 'observational error' on the artificial data is

given by $\sigma_{s,i}$, which I have set equal to a constant value of 5% of the maximum velocity magnitude of the two stars. The number of degrees of freedom $D.O.F.$ of the system is equal to the total number of data points ($2N$), minus the number of parameters (6). A good solution should have $\chi^2_{reduced} \approx 1$.

The fitness function is given by the following code:

```
function F=binaryFitness(X,extPar)
% Fitness function for the spectroscopic binary modeling code from the
% Qubist user's guide.
%
% Determine number of parameters and the number of solutions requesed by
% Ferret.
[NPar,N]=size(X);
for i=N:-1:1
    %
    if mod(i,10)==0
        % Pause occasionally to keep user interfaces responsive.
        pause(0.001);
    end
    %
    % Convert Ferret's 'X' matrix to a parameter structure that the
    % binary.m program understands.
    extPar.binaryPar=X2Par(X(:,i), extPar.binaryPar);
    %
    % Compute the velocity curves:
    % t --> time (unused).
    % V --> velocity.
    [t,V]=binary(extPar);
    %
    % Degrees of freedom:
    DOF=numel(extPar.data.V)-NPar;
    %
    % Fitness: just the usual reduced chi^2 formula.
    F(i)=sum(sum((V-extPar.data.V).^2/extPar.data.dV^2))/DOF; %#ok
    %
end
```

Note that the fitness function calls a program called binary.m, where all of the astrophysical modeling is done. Interested readers can examine this function to understand the details of the physics involved. However, I anticipate that most readers will be content with understanding the binaryFitness.m function printed above, since this gives a good template for χ^2 data-modeling problems in general, while avoiding the details of this particular system.

The model and artificial data curves would be identical if a set of parameters could be found such that $F = \chi^2_{reduced} = 0$, but of course, this is not possible because I have added noise to the data. The best fit that we can reasonably expect would have $\chi^2_{reduced} \approx 1$. In fact, it can be argued that it is difficult to distinguish between models within one of the $\chi^2_{reduced}$ minimum: $\chi^2_{reduced} \leq \chi^2_{reduced,min} + 1$, because all of these models are within the error bars of the data. The set of models satisfying this property is a family of solutions in parameter space, whose distribution provides a good indication of the uncertainty of each parameter. The link between $\chi^2_{reduced}$, maximum likelihood, and error estimation is outlined concisely in

the parameter estimation chapters of the Numerical Recipes books, some of which are offered online free of charge at the time of this writing (http://www.nrbook.com).

I have chosen to map out the set of models within one of the $\chi^2_{reduced}$ minimum by setting a fuzzy tolerance: `par.selection.FAbsTol=1`. Ferret ignores differences that are less than `par.selection.FAbsTol` when the selection operator is evaluated. When niching is enabled (`par.niching.X > 0` or `par.niching.F > 0`), a stable population should develop, which fills out the region within `par.selection.FAbsTol` of the minimum of F. In our case, this is effectively the region containing all of the models that are allowed within our noise limit. When the run is analyzed, the OptimalSolutions.mat file will contain all solutions discovered with $\chi^2_{reduced} \leq \chi^2_{min,reduced} + 1$. However, the OptimalSolutions.rank substructure will contain true, non-fuzzy ranks that ignore the $FAbsTol$ parameter. Thus, the optimal set will contain many solutions within the error tolerance, but only the solution (or solutions) with $\chi^2_{reduced} = \chi^2_{min,reduced}$ will be tagged as $OptimalSolutions.rank = 1$.

5.2 Visualizing Results with the Analysis Window

Figure 5.1 shows the Ferret console window at the end of a long run of the binary star modeling project. I used four populations of 250 individuals each and ran for 500 generations. This is really overkill for this problem, but I wanted nice figures with a very well-defined optimal set and a lot of points. It is clear from the 'Optimal Fitness Bounds' panel that a stable population emerged very early on in the run, since the lower curve, which is the $\chi^2_{reduced}$ of the best model in the population, is almost completely flat past about 40 generations. The upper curve represents the $\chi^2_{reduced}$ of the worst model in the optimal set at a given generation, which is just $\chi^2_{min,reduced} + 1$ in this case.

For more than 90% of this run, there was no significant improvement in $\chi^2_{reduced}$. However, this is a common for projects where the goal is to map the structure of an extended optimal set. Additional solutions are discovered as the code continues to run past the time of convergence, and these extra solutions help to define the structure of the optimal set. Analysis and visualization of the optimal set reveal greater detail as the run progresses past convergence.

The four main views of the Ferret analysis window are shown in Figure 5.2. You can toggle between these views using the buttons in the 'View' button group in the upper right hand side of the figure.

- Clicking on the button labelled 'X-F Space' produces a parameter space plot like the one shown in panel a). Any two or three dimensional projection of the parameters or objectives is possible, and several different plot types can be generated. This is the richest view in the interface, and we will return to its discussion later in this section.

- The 'Linkage' button shows a two-dimensional view of Ferret's linkage matrix, averaged over all of the solutions present in the optimal set. I often joke with new users that this figure is an 'X-ray of Ferret's brain', which is true in the sense that the linkage map represents what Ferret has learned about the problem's mathematical structure. It visually represents Ferret's best working model of how the problem at hand can be divided into sub-problems, and therefore Ferret's strategy for solving it. The linkage map shown in the analysis window differs from the corresponding panel of the console window (see Figure 5.1) in that the console window shows a snapshot of the current linkage map at every generation, while the analysis window shows a linkage map that is averaged over all solutions in the final optimal set.

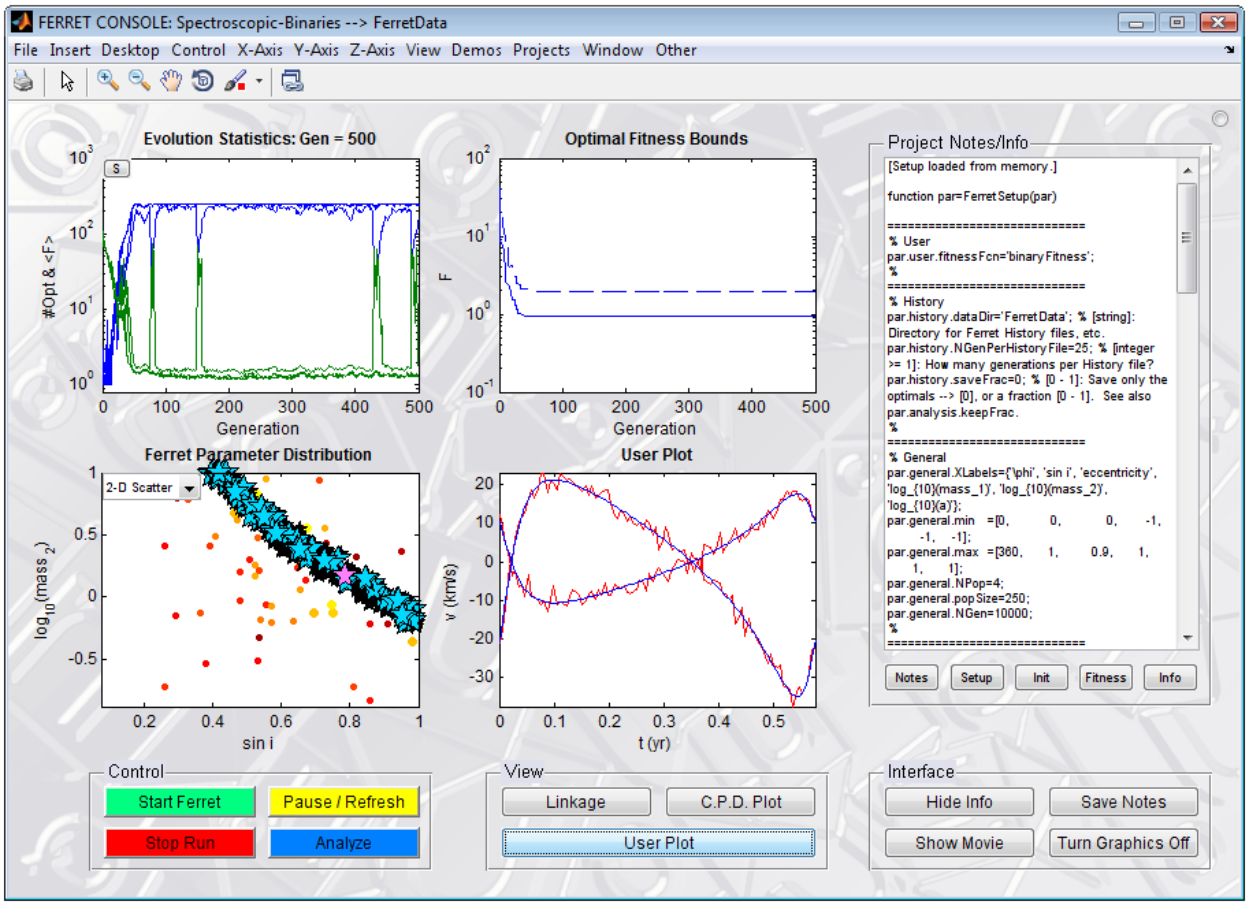

Figure 5.1: The Ferret console window after running the binary modeling project for 1000 generations.

To understand the linkage map, note that the shading of each square is proportional to the probability of linkage: light coloured squares indicate strongly linked parameters, while dark squares indicate that parameters are unlinked or linked with low probability. The overall brightness of the figure is a good indication of the difficulty of the problem. When the figure is very dark and only a small fraction of the squares are lit up significantly, this means that Ferret found the problem to be quite easy because it naturally divides into many smaller sub-problems. On the other hand, a bright linkage map with many cells lit up indicates that Ferret could not find many natural sub-problems, so the problem had to be solved as a whole. This usually implies that the problem is more difficult.

Failure to partition a problem into subproblems does not indicate that Ferret's linkage-learning system has failed, because some problems are not solvable by partitioning the parameter space. Our binary star modeling example cannot be partitioned, and the linkage map shown in panel b) of Figure 5.2 is very bright. However, with only six parameters, Ferret had no trouble solving this problem as a whole. Linkage learning becomes more important for larger problems, where some partitioning is usually possible. Some good examples are given in the 'Linkage-Learning' demos,

which are available from the Demo menu of the Ferret console window.

- The 'C.P.D. Plot' button shows a histogram of the number of each fully specified gene (or parameter) for every solution present in the optimal set. Ferret's critical parameter detection (CPD) feature is turned on by setting `par.CPD.PDeactivateGenes > 0`. Whenever CPD is enabled, Ferret attempts to minimize the size of its parameter space by turning genes (parameters) off, and storing a NaN as a placeholder for each deactivated parameter. Such NaN values are interpreted as parameters that are unspecified because they are unimportant, and are normally replaced by random numbers (when `par.CPD.NaN2Random=1`) chosen within the range of the search (`par.general.min` to `par.general.max`). All parameters were important for this problem. The Multi-Objective/Triangle-Subspace-CPD demo is a good example of a problem where CPD simplifies the parameter space rather dramatically.

- The 'User Plot' button causes Ferret to evaluate and display a custom graphics plot, if `par.interface.myPlot` is the name of a valid user-defined m-file containing graphics commands. This same functionality is also available in the console window through the 'User Plot' button there. A message is displayed on the user plot axes if the myPlot function has not been specified, or if an error occurs during its evaluation.

Panel a) of Figure 5.2 is only one possible way to view projections of the optimal set. On Windows and Linux systems, there is a small drop-down menu in the upper-left of the analysis window that allows you to select between several types of graphs. This is replaced by a button on MacIntosh systems, which causes the same options to pop up as a separate window containing a listbox control. These options include the following:

- 2-D Scatter: Scatter plots of all members of the optimal set, or the fuzzy optimal set if fuzziness is enabled (`par.selection.FAbsTol > 0` or `par.selection.FRelTol > 0`), as discussed in Section 5.6. If fuzziness is turned on, then points are shaded according to their true (non-fuzzy) rank, such that darker points have better (lower) rank than points displayed in lighter, more washed out colours. Points are shown in orange if one of the two parameters in the current projection is not specified due to Ferret's CPD system. Points are displayed in red when neither of the projections parameters shown in the projection are specified. The projection can be chosen from the X-Axis and Y-Axis menus, and any combination if parameters or fitness values is possible.

- 3-D Scatter: A scatter plot showing members of the optimal set. The projection can be chosen from the X-Axis, Y-Axis, and Z-Axis menus, and any combination of parameters or fitness values is possible. This option is not recommended for large optimal sets on systems that do not have much memory, and a warning dialog pops up when this option is selected if there are more than 1000 points. In my experience, you can reasonably plot up to about 10,000 points with some patience, but you may not be able to interactively rotate the axes using the MATLAB figure controls.

- Contour: A contour plot showing the optimal set. Any combination of parameters or fitness values can be displayed; the X-Axis, Y-Axis and Z-Axis menus control the projection, and it is the parameter or fitness value specified by the Z-Axis that is actually contoured. The z values are interpolated onto a uniform grid prior to contouring. The grid size, contour levels, colormap, and shading options are chosen using the 'Contour/Image Plot Options' control panel, shown in Figure 5.6, which pops up when this option is chosen. These options and the Contour/Image Plot Options control panel are discussed in Section 5.4.

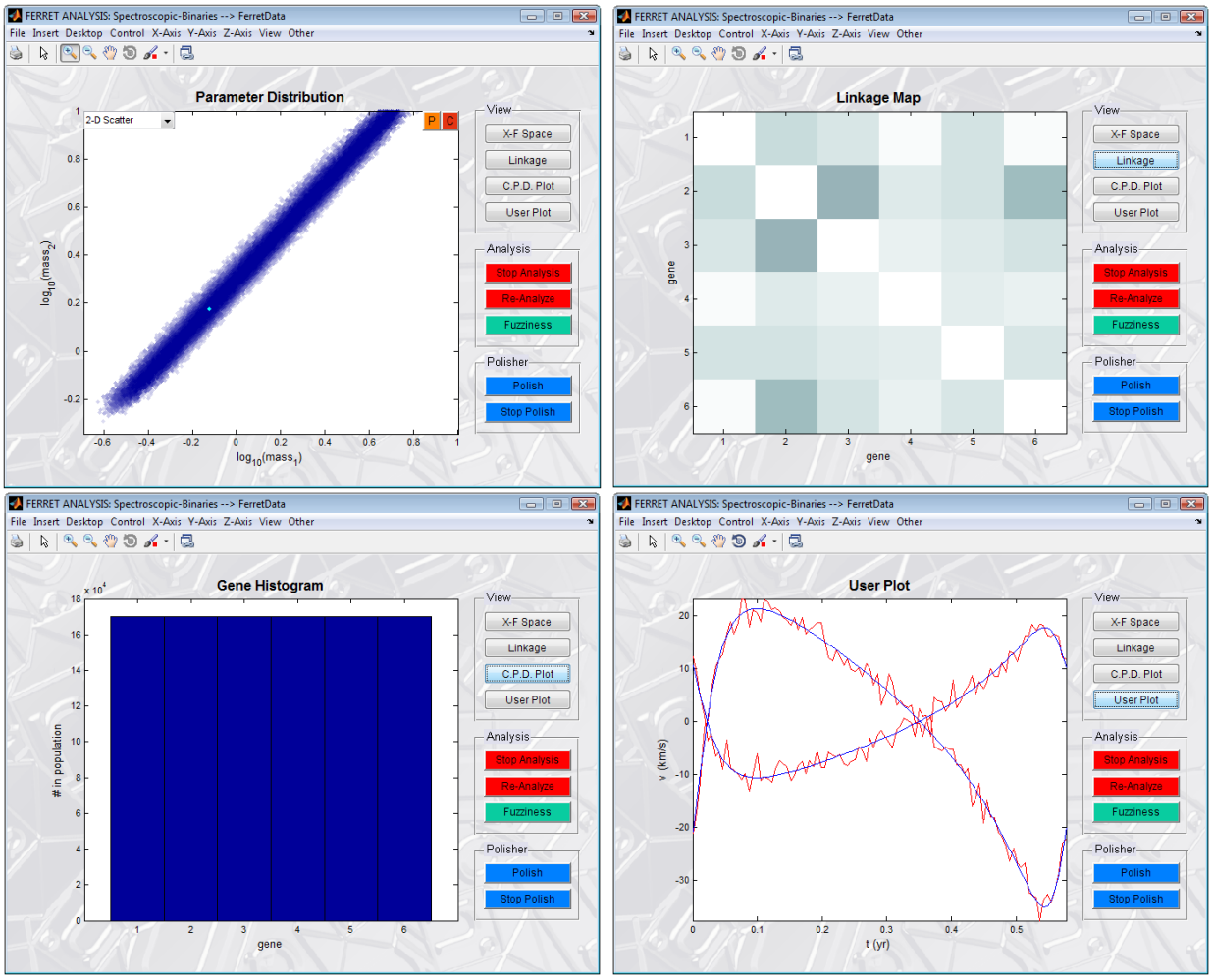

Figure 5.2: The four main views of the Ferret analysis window. Panel a) (top left) 'X-F Space': two and three-dimensional projections of parameter and fitness space. The shading in points indicates the rank of the solution. Better solutions (lower rank) are shaded darker blue than relatively poor solutions (lighter blue). The bright blue dot is the best solution that was discovered. Panel b) (top right) 'Linkage': A view of the cumulative linkage map, averaged over the solutions in the optimal set. Panel c) (lower left) 'C.P.D. Plot' A histogram showing the number of fully specified values present for each parameter (gene) in the optimal set of solutions. This can be thought of qualitatively as the relative importance of each parameter. Panel d) (lower right) 'User Plot': An optional plot that can be drawn by the user if `par.interface.myPlot` is the name of a valid user-defined m-file containing graphics commands.

- Image: A two-dimensional image plot showing members of the optimal set. The Contour/Image Plot Options control panel also pops up when this option is selected.

- 2-D Combo #1: A combination of the contour and 2-D scatter plot.

- 2-D Combo #2: A combination of the contour and 2-D image plot.

- 3-D Combo: A combination of a 3-D scatter plot, with a scatter plot shown in the z=0 plane. three-dimensional plotting should be avoided on computers that do not have a lot of memory if the optimal set is very large. The user is warned of this whenever more than 1000 points are about to be plotted.

Figure 5.3 shows a few examples of these plot options. The two-dimensional plots shown here contain about 170,000 points and can be displayed very nicely as image and contour plots. However, this is far too many to display as a three-dimensional plot, so I stopped the analysis after loading only the final History file generated by the run (Note that History files load in reverse order during analysis, starting with the best solutions from near the end of the run), which resulted in about 9,100 points. This still exceeds the recommended maximum number for plotting, but I was able to render the image and manipulate it with some patience. Rotating the figure interactively is not practical for this many points, so I recommend using MATLAB's 'view' command to rotate it from the command line. For example, the default view was not very good for the plot shown in panel d) of Figure 5.3, so I rotated it manually using the command

```
view(37.5,30)
```

I also scaled the axes manually using the command

```
set(gca,'XLim',[-0.65,1],'YLim',[-0.3,1],'ZLim',[0.4,1])
```

After I analyze a run, I usually go to the 'X-F Space' projection and begin flipping through various two-dimensional projections looking for interesting relationships between the parameters and the objective functions. I generally prefer to use the scatter plot or image plot options for problems with many points in the optimal set. Figure 5.4 shows several two-dimensional projections of the optimal set for the spectroscopic binary modeling problem.

Panel a) of Figure 5.2 and panels b) through d) of Figure 5.4 indicate that something quite interesting is going on with this model. It is clear that the distribution of points fills out an essentially *linear* region of parameter space, which *contains* the real solution, along with many other models. This is an example of *degeneracy* in a data-modeling problem, which arises because we are attempting to determine model parameters from incomplete information. Recall that only the component of the motion along the line of sight toward or away from the observer can be observed in a spectroscopic binary, while motion in the plane of the sky cannot be resolved. It turns out that this does not provide quite enough information to fully constrain the physical parameters of our model. It can, in fact, be shown that there really does exist a one parameter family of spectroscopic binary models that look exactly alike, and Ferret has discovered this relationship numerically!

There are two approaches that you can take if your plots show evidence of mathematical degeneracy. If the relationship seems simple, you can try to discover it by plotting projections and various functions of the parameters, trying to find a relationship that collapses a degenerate extended structure into a well defined cluster of points. This can be time-consuming and requires some intuition, but power law scaling relationships are common in nature and quite easy to find by this method. Alternatively, once you *know* that mathematical degeneracy exists in your problem, it is usually not too difficult to find the source by examining your equations carefully. In our example, one of the relationships is obvious from panel a) of Figure 5.2, which shows points distributed about a straight line with a slope of one:

$$\log_{10} m_2 = \log_{10} m_1 + k \tag{5.2}$$

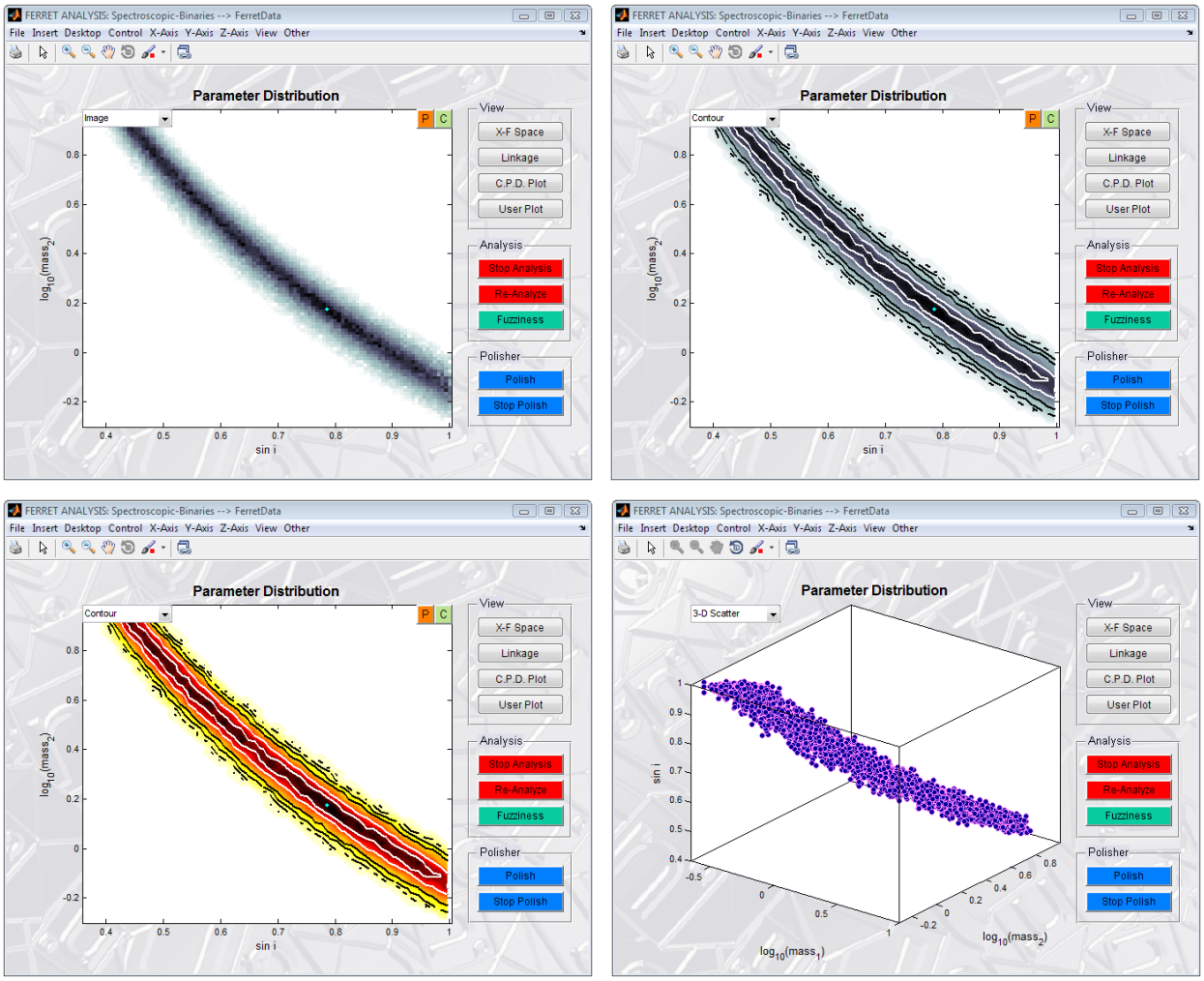

Figure 5.3: Examples of several types of plots that are possible with the Ferret analysis interface. Panel a) (top left): An image plot with a logarithmic stretch, using the default 'bone' colour map. Panel b) (top right): A contour plot with a logarithmic stretch. The black and white contours are selected using the 'Contour/Image Plot Options' interface discussed in Section 5.4 below. Panel c) (lower left): Same as panel b), except using the 'hot' colour map. Most of the standard MATLAB colour maps are available in the plot options interface. Panel d) (lower right): A three-dimensional plot.

for some constant k. Equivalently, this relationship can be expressed as

$$q_1 = \frac{m_2}{m_1} = constant. \tag{5.3}$$

Thus, the data can uniquely determine the mass ratio m_2/m_1, even though the masses cannot be determined independently. There are two other relationships that can be found with somewhat greater effort:

$$q_2 = \frac{a^3}{m_1 + m_2} \tag{5.4}$$

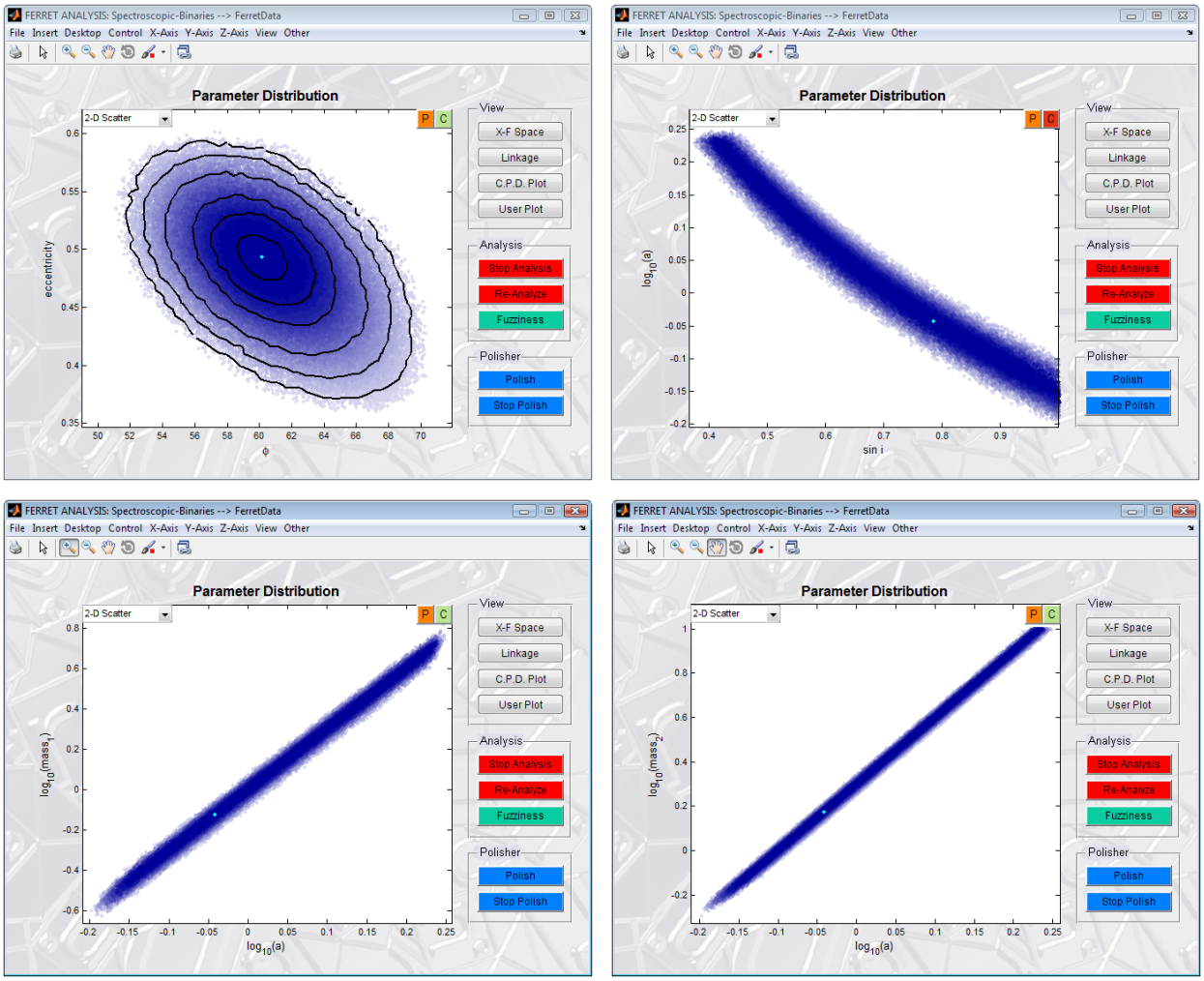

Figure 5.4: Several two-dimensional projections of the optimal set for the spectroscopic binary problem using the image plot option. All solutions shown are within the 'observational errors' of the noisy artificial data in the sense that they are within 1 of the reduced $\chi^2_{reduced}$ minimum value: $\chi^2_{reduced} \leq \chi^2_{min,reduced} + 1$. Contours can be drawn optionally on any two-dimensional plot, as shown in panel a). Panel a) of Figure 5.2 shows another projection.

$$q_3 = \sqrt{\frac{m_1 + m_2}{a}} \sin i, \tag{5.5}$$

where q_1, q_2, and q_3 are well defined quantities that can be determined from the data. Figure 5.5 shows these q functions plotted against each other to show that they really do collapse our extended solution set into a compact cluster of points.

The relationships given above can also be thought of as a set of similarity transformations that map one set of parameters onto a new set that produces *exactly* the same velocity curves. For some arbitrary scaling

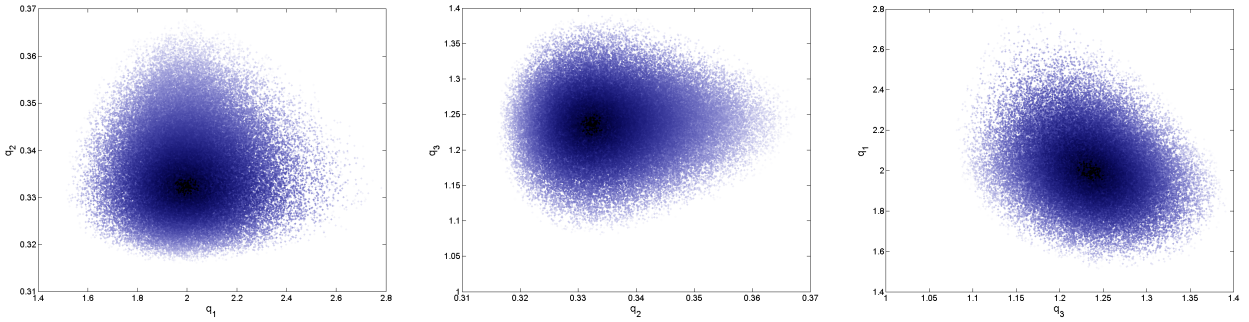

Figure 5.5: Three different projections of the 'q' functions defined by equations 5.3 to 5.5. As our analysis predicts, all three functions are well-determined by the model, and therefore collapse the extended distribution of points found by Ferret into a compact cluster of points.

parameter $S > 0$, equations 5.3 to 5.5 are equivalent to the following one-parameter scaling relationship:

$$\begin{aligned} a &\rightarrow Sa \\ m_1 &\rightarrow S^3 m_1 \\ m_2 &\rightarrow S^3 m_2 \\ \sin i &\rightarrow S^{-1} \sin i \end{aligned} \tag{5.6}$$

This explains why we see extended linear structures in our optimal set, because a line, or a more general curve, is a set of points that can be described by a single parameter. The Advanced/Spectroscopic-Binaries demo included with Qubist contains a function called testSimilarity.m, which allows you to test this similarity relationship for different orbit parameters and values of S.

Ferret, plus some *a posteriori* analysis of the solution set, has determined the mathematical nature of this problem's degeneracy, which tells us what can and cannot be learned about binary stars from spectroscopic data alone. I get excited when I see evidence of a mathematical degeneracy, because it means that there is an interesting relationship or symmetry in my problem, and I begin to search for it in my equations. As it turns out, equation 5.3 comes from the basic fact that the stars orbit about a common centre of mass, which is at the origin by construction. Thus,

$$m_1 \mathbf{x}_1 = -m_2 \mathbf{x}_2 \tag{5.7}$$

and

$$m_1 \mathbf{v}_1 = -m_2 \mathbf{v}_2, \tag{5.8}$$

where $\mathbf{x}$ and $\mathbf{v}$ are the position and velocity vectors respectively. It follows that

$$\frac{m_2}{m_1} = \frac{max\{|v_{1,obs}|\}}{max\{|v_{2,obs}|\}}, \tag{5.9}$$

where $max\{|v_{1,obs}|\}$ and $max\{|v_{2,obs}|\}$ are respectively the maximum magnitudes of the observed Doppler velocities as the stars go about their orbits. Equation 5.4 comes from Keppler's Law, which tells us that the period of the orbits is given by

$$P = \sqrt{\frac{4\pi a^3}{G(m_1 + m_2)}}. \tag{5.10}$$

Since we can observationally measure the period, the data must determine $a^3/(m_1 + m_2)$ accurately, and hence q_2 must be well-determined by the model. Finally, the model tells us that the orbit velocity scales like

$$v \propto \sqrt{\frac{G(m_1 + m_2)}{a}}, \tag{5.11}$$

but the observed Doppler velocity is given by $v_{obs} = v \sin i$. Thus, we expect that the amplitude of the velocity curves should accurately determine the q_3 function given by equation 5.5.

Degeneracy is common in many real-world problems and the ability to identify degeneracy and determine its mathematical structure, even approximately, is key to understanding how a physical system works. The example given in this section illustrates how Ferret can be used to probe the mathematical structure of models in a complicated multi-dimensional parameter space, in the presence of noisy data. In addition to being an extremely powerful optimizer, Ferret's ability to map and visually explore entire classes of optimal solutions in a multi-dimensional parameter space is one of the features that makes it an extremely powerful research tool. These features can point out real degeneracies in a model, and inform the user what is and what is not *knowable* about a model from a given data set. The information learned from such an exploratory approach may result in new insights about the mathematical structure of the model, as they did for our simple example, or they may point the way to an improved non-degenerate model with fewer parameters.

5.3 Parameter Space Projections

The X-Axis, Y-Axis, and Z-Axis menus can be used to select any two or three-dimensional projection of the parameter space in order to explore the structure of a multi-dimensional optimal set. These menus are active in both the console and analysis windows, but they are especially important for studying the structure of the optimal set displayed in the analysis window. Whenever I analyze a new problem, the first thing I do is to set the analysis window to a two-dimensional projection and start examining different axis projections. This often leads to important insights into the structure of the optimal set. For example, you can quickly discover which parameters are well defined (i.e. have a narrow distribution in the parameter space), and which are poorly defined (i.e. have a broad distribution). Perhaps even more importantly, one often discovers relationships between parameters, which can lead to important physical insights into the mathematical structure of the problem's underlying model, like the degeneracy that we discovered in Section 5.2. These types of degeneracies (or partial degeneracies) appear in data fitting problems quite frequently and are often characterized by long, linear features that can be seen in two-dimensional projections of the $\chi^2_{reduced}$ surface. Whenever you see a feature like that in the analysis window, it is a good idea to step back and try to understand *why* it exists, because it may indicate an important symmetry in the equations, which can tell you a great deal about how your model works.

It is sometimes useful to write a simple code to load the OptimalSolutions.mat file and flip through projections automatically. You can do this sequentially, but I often find it just as useful to examine random projections. The following is a skeleton of a code that does exactly this:

```
function showProjections(OptimalSolutions)
% Cycle through random projections of the OptimalSolutions structure.
%
NGenes=size(OptimalSolutions.X,1);
figure(1);
%
% Note this is an infinite loop!  Users must send control-C to exit.
while true
        % Choose axes randomly, but make sure they are different.
        xAxis=ceil(rand*NGenes);
        yAxis=xAxis;
        while yAxis == xAxis
                yAxis=ceil(rand*NGenes);
        end
        %
        % Make plots.
        plot(OptimalSolutions.X(xAxis,:), OptimalSolutions.X(yAxis,:),'.');
        %
        % Label axes.  Axis names use the labels in par.general.XLabels if they exist.
        % Otherwise, labels are generated from the parameter number.
        if isempty(OptimalSolutions.par.general.XLabels)
                xlabel(['X_', num2str(xAxis)]);
                ylabel(['X_', num2str(yAxis)]);
        else
                xlabel(OptimalSolutions.par.general.XLabels{xAxis});
                ylabel(OptimalSolutions.par.general.XLabels{yAxis});
        end
        %
        % Stop to allow users to view the figure.  Press any key to move on.
        pause
end
```

5.4 The 'Contour/Image Plot Options' Interface

The optimal set generated by Ferret can be thought of as scattered data that is represented by a set of points in an $(N_{par} + N_{obj})$-dimensional space, where N_{par} is the number of parameters and N_{obj} is the number of objectives. These points are viewed as a scatter plot by default, but Ferret can also represent them as an image or contour plot, which smoothes out the data and sometimes makes the distribution of points easier to see. This is especially useful when your solution set contains a large number of points, since scatter plots can become confusing when the individual points start to overlap.

To create an image or contour plot from the scattered point data in an OptimalSolutions structure, Qubist constructs a two-dimensional projection of the points, and then bins them onto a regular grid. The 'Contour/Image Plot Options' interface, shown in Figure 5.6 is a set of controls that can be used to control this binning and modify the appearance of two-dimensional image and contour plots that are displayed in the analysis window. Ferret's Z-Axis menu controls which parameter or objective is represented by the colour or shading level of an image plot, or by contours for a contour plot.

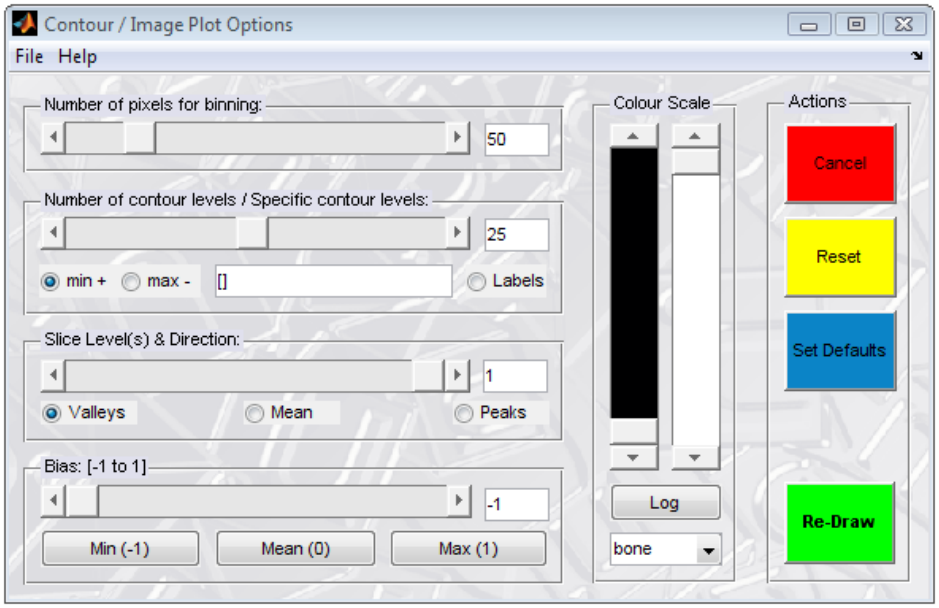

Figure 5.6: The 'Contour/Image Plot Options' control panel shown here pops up whenever a plot option is selected that requires a two-dimensional image or contour plot.

The image options interface may seem quite complicated at first, but it will seem simpler once you have used it a few times. It provides a great deal of flexibility in how your images are displayed, which helps to tease information out of your optimal set or to prepare publication quality figures.

The slider and corresponding text box titled 'Number of pixels for binning' controls the number of pixels used for image and contour plots. If the slider value is set to a number N_{pix}, then image and contour plots will be based on an $N_{pix} \times N_{pix}$ grid. This control is probably the most important one on the contour/image plot interface, and the ability to see features in the optimal set hinges on choosing it well. The idea is to make the grid coarse enough (sufficiently small NPix) that the image or contour plot looks smooth and well filled in, but fine enough (sufficiently large N_{pix}) to minimize information loss due to over-smoothing the image. Generally, larger values of N_{pix} are appropriate for large optimal sets, while smaller values are required to fill in gaps between points when the optimal set is small. Usually, I just try various settings until I get the visual effect that I'm looking for. Panel a) of Figure 5.3 shows an image plot with points shaded according to the logarithm of $\chi^2_{reduced}$ for our binary star modeling example. Logarithmic scaling does a better job of highlighting the 'spine' of the curve than linear scaling or scatter plots.

For contour plots, the number of contours can be specified using the slider or textbox labelled 'Number of contour levels / Specific contour levels'. A small number of levels makes a filled contour plot look like it has steps between levels, while a larger number tends to smooth out the plot. This control has no effect on image plots. It is also possible to specify specific levels, which are plotted as heavy black or white lines. These contour levels appear on *any* two-dimensional view, including scatter plots (See the top left panel of Figure 5.4 for example), image plots, and the 2-D Combo plots. The contours are labelled if the 'Labels' radio button is selected. If either of the buttons labelled as '$min+$' or '$max-$' are selected, then the

contour levels chosen are regarded as relative to either the minimum or maximum z value in the current projection of the optimal set. For example, if a contour level 0.1 is chosen and $min+$ is chosen, then the level actually contoured will be the minimum z value plus 0.1. If $max-$ is chosen, then the maximum z value minus 0.1 will be the level that is contoured. Panels b) and c) of Figure 5.3 show two contour plots of the same projection that are displayed using different colour maps.

An image or contour plot bins and flattens the X-Y-Z values into two dimensions, and displays the Z-coordinate using colours, shades, or contours. However, there is some choice in how these data are flattened. For example, you could choose to represent the minimum value in each square bin or pixel, the maximum value, or the average z value of all points that fall within that pixel. Other choices are also possible. For example, you could slice through the points that fall within a particular pixel, and choose to average only the values below the slice, or above the slice, or in both directions some distance on either side of the slice. My point is that real choices exist in how to display these data, and your choice can strongly affect how well you can see features in the optimal set.

Qubist uses a combination of data slicing and averaging to convert scattered three-dimensional data into a two-dimensional image or contour plot. The third slider on the contour/image interface sets the value of a control parameter called *sliceLevel*, which determines how the data are sliced. There are three radio buttons below this slider, labelled 'Valleys', 'Mean', and 'Peaks', which determine how the data slicing is applied. The procedure used in each case is discussed below:

- Valleys: Determine the extent of the data in the z-direction: $\Delta z_0 = max(z) - min(z)$. Apply a slice at certain fraction 'sliceLevel' above the minimum: $z_{slice} = z_{0,min} + \Delta z_0 \times sliceLevel$. Throw away all points above the cut, and keep the remaining points to display. All of the data points are included when $sliceLevel = 1$.
- Peaks: Determine the extent of the data in the z-direction: $\Delta z_0 = max(z) - min(z)$. Apply a slice at certain fraction 'sliceLevel' below the maximum: $z_{slice} = z_{0,max} - \Delta z_0 \times sliceLevel$. Throw away all points below the cut, and keep the remaining points to display. All of the data points are included when $sliceLevel = 0$.
- Mean: Find the average z value of points in the optimal set. Cut both above and below the mean level, using the 'peaks' and 'valleys' method discussed above. In this case, sliceLevel determines the width of the band about the mean. All of the data points are included when $sliceLevel = 1$.

The points that remain in each pixel after slicing must still be combined to determine an overall value for the pixel. Once again, there is some choice in how this can be done. This is done by choosing a control parameter called the *bias*. The most important consideration about the bias is that it is applied on a *pixel-by-pixel* basis. This is in contrast to the slicing operation, which cuts through the data set as a whole. The three choices for the bias are as follows:

- bias = 0: Return the mean value for each pixel.
- bias < 0: For each pixel separately, determine the range Δz of values contributing to the pixel: $\Delta z = max(z) - min(z)$. Place a cut at some height a fraction equal to *bias* down from max(z): $z_{cut} = max(z) - |bias| \times \Delta z$. Return the mean value of points below the cut.
- bias > 0: For each pixel separately, determine the range Δz of values contributing to the pixel: $\Delta z = max(z) - min(z)$. Place a cut at some height a fraction equal to *bias* up from min(z): $z_{cut} = max(z) - bias \times \Delta z$. Return the mean value of points above the cut.

The following are a few common cases that are worth considering:

- If a user wants to plot the minimum value in each pixel, then 'Valleys' should be chosen for data slicing, sliceLevel should be set to 0, and bias should be set to -1.
- If a user wants to plot the maximum value in each pixel, then 'Peaks' should be chosen for data slicing, sliceLevel should be set to 0, and bias should be set to 1.
- If a user wants to plot the mean value in each pixel, then 'Mean' should be chosen for data slicing, sliceLevel should be set to 0, and the bias should be set to 0.

Other choices for the bias can be used to effectively smooth the data prior to image or contour plotting in order to make figures look less noisy. For example, if we are viewing a plot in 'Valleys' mode and set $bias = -1$, then the image is shaded by using only the minimum z value of the points within each pixel. The resulting image can look quite noisy, since only a single value (the minimum) is actually used to compute the overall value for each pixel. On the other hand, if we set the *bias* to a slightly higher value, say -0.9, then multiple points near the minimum value will be averaged to compute the displayed shade of each pixel. Averaging tends to smooth out noise, and as a result, the image will appear smoother in most cases.

The panel labeled as 'Colour Scale' controls the colour map for the image. There are two vertical sliders that control the colors assigned to the minimum and maximum z values of pixels. These can be used to adjust the contrast of the image, or even to reverse the colour scheme if the slider on the left is placed higher than the slider on the right. There is a toggle button labeled as 'Log' below the sliders. When this is pushed in, a logarithmic stretch is applied to the data before it is shaded. Logarithmic stretches are useful when the z values of the pixels have a large dynamic range. This panel also contains a pop-up menu that allows you to select between most of the colour maps that are available in MATLAB. It is set to MATLAB's bluish-gray 'bone' colour map by default, which I think looks good with the Qubist interface, but feel free to experiment.

Note that changes made using the plot options interface do not take effect until the user presses the 'Re-Draw' button in the 'Actions' panel. The reason for this delayed action is to avoid unnecessary and sometimes time-consuming re-binning and re-rendering of plots until all changes have been made. The only exception is for changes to the colour map, which take effect immediately because no re-binning of data is required. Note that this interface keeps track of default settings. At any time, the user can revert to defaults or set new ones, using the 'Reset' and 'Set Defaults' buttons. The 'Cancel' button closes the plot options interface without applying any changes.

5.5 The 'Painted Points' Interface

We now return to the binary star modeling example of Section 5.1 to illustrate a powerful way to explore the models presented by the analysis window. There are two small buttons labelled 'P', and 'C' in the upper right hand side of the plot axes in Figure 5.7 (left figure), which stand for 'Paint' and 'Colour' respectively. Clicking on the 'C' button chooses a new random RGB (red/green/blue) active colour for painting solutions, and the 'C' button will also change to the active colour. These colours do not cycle, and you should just keep clicking until you see a color that you like. This is implemented in this way because I wanted a very simple interface with a large number of possible colours.

If you click the 'P' button, you should see a set of crosshairs on the analysis window, which you can use to select a single solution or multiple solutions. To select a single solution, just click on the axis and the

nearest point is selected. To select a circular region, right-click on the centre, release the mouse button, and drag the cursor outward to define the radius. You should see a red circle, which shows its size, and all points within the circle will be selected. Regions more complicated than a circle can be built up from several circular regions selected with the same colour.

A second effect of selecting points, or clicking on the 'C' button, is that the 'Painted Points' control will pop up, as shown on the right hand side of Figure 5.7. This interface contains three text boxes that display information about the selected solutions, plus several other controls to work with your point list. The text box on the left is simply a numerical list of points, indicating their number in the saved OptimalSolutions.mat file. For example, solution 82033 (the selected solution in the text box on Figure 5.7) corresponds to `OptimalSolutions.X(:,82033)`, `OptimalSolutions.F(:,82033)`, and `OptimalSolutions.auxOutput{82033}`, if your fitness file returns *auxOutput* (see Section 4.8). The colour of this text box matches the colour of the selected point. This colour will change as you click on different solution numbers in the text box, and a faint blue-green circle will appear on the analysis window axis, centred on the selected solution. The upper right text box shows the parameter list corresponding to the selected solution (i.e. `OptimalSolutions.X(:,82033)`), and the lower right text box shows the corresponding fitness value (i.e. `OptimalSolutions.F(:,82033)`). *It is very important to note that the order of the solutions in OptimalSolutions will change if the run is re-analyzed, unless OptimalSolutions.mat contains only a single solution. If you re-analyze your run, any points that were painted previously may not have the same numbers, unless OptimalSolutions only contains a single solution.*

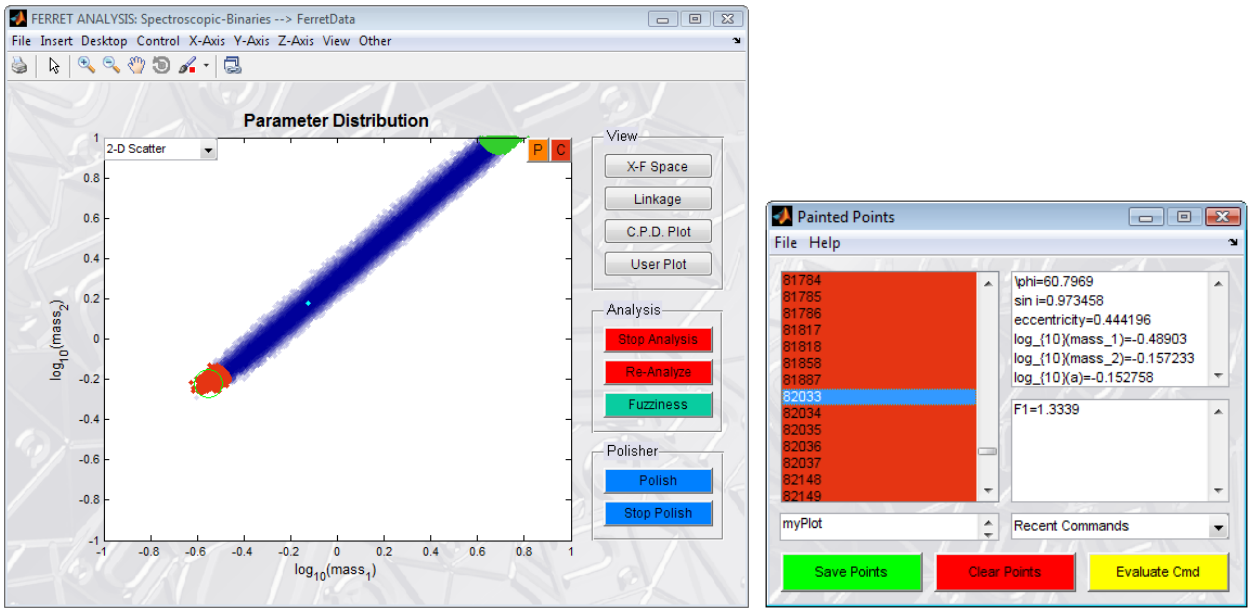

Figure 5.7: Painting points for further analysis: Panel a) (left): It is possible to select points by pressing the small 'P' button in the upper right hand corner of the analysis graphics panel and either clicking on individual points or selecting circular regions. Panel b) (right) The selected points are listed in the 'Painted Points' interface, and shown in the figure on the right.

You can change the projection that is displayed, and any previously painted points move to their new projected positions, as shown in Figure 5.8. For example, Figure 5.8 shows that the solutions with the most massive stars, painted in green, have the lowest inclination of the orbital plane (i.e. their orbits are

closer to being in the plane of the sky) and the largest semi-major axes, which is verified by equations 5.4 and 5.5. Conversely, the solutions with the lowest mass stars have the smallest semi-major axes and the largest inclination angles. Note that some solutions have $\sin i \approx 1$, which indicates that the orbital plane is almost exactly perpendicular to the line of sight. The story is quite different when we view the painted points from the projection showing the orbital phase angle (ϕ) and eccentricity. Here we see that the least massive and most massive solutions form a compact distribution that is thoroughly mixed, which indicates that the phase angle and eccentricity can be obtained independently from the mass.

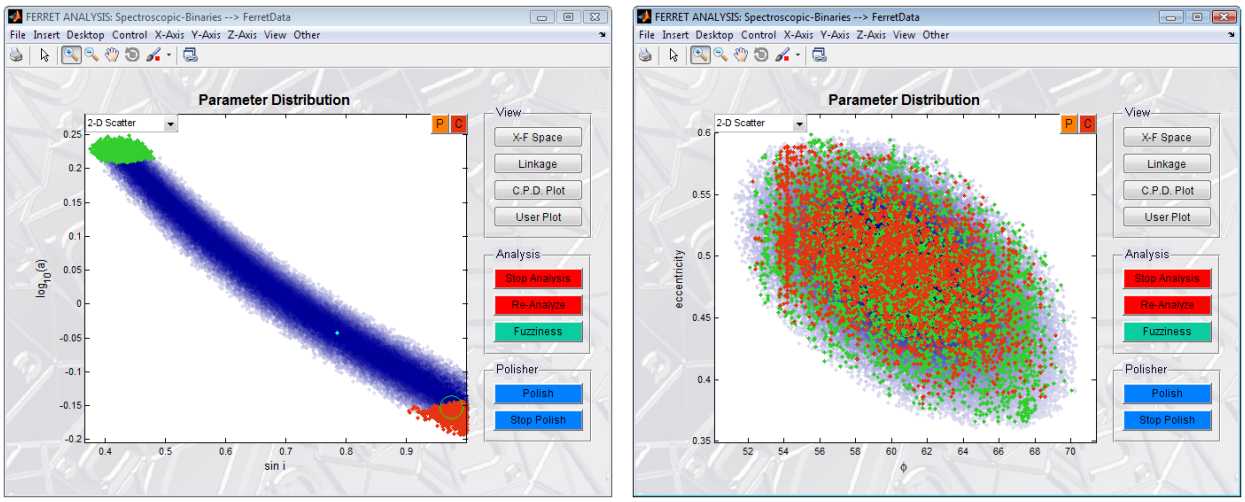

Figure 5.8: Panel a) (left): Different projections, showing the same points selected in Figure 5.7. Panel b) (right) A plot showing the result of executing the 'myPlot' function on a set of parameters selected in the Painted Points interface.

It is also possible to visualize individual solutions using a customized plotting function if your project has a myPlot function defined (see Section 4.10). This is done by selecting a solution from the list of painted points, entering the name of your myPlot command in the text box above the 'Save Points' button shown in Panel b) of Figure 5.7, and clicking the 'Evaluate Command' button. This technique can evaluate *any* MATLAB command that is entered in the textbox, as long as it has the same function signature as a myPlot function. The text in this box is processed using MATLAB's 'eval' command without much validation, so please be careful when using this feature.

It is possible to save your painted points in the OptimalSolutions.mat file by clicking the button labelled 'Save Points'. This will add a new field `OptimalSolutions.paintedPoints` to the *OptimalSolutions* data structure and re-save the OptimalSolutions.mat file. The paintedPoints structure is organized as follows:

```
paintedPoints  % top-level painted points structure
              |
              .index      % Cell array containing indices for each colour.
              |
              .X          % Cell array of matrices containing X values for each colour.
              |
              .F          % Cell array of matrices containing F values for each colour.
              |
              .rank       % Cell array of vectors containing the rank of each solution,
              |           % for each colour.
              |
              .colour     % Array of colours: Each row is the RGB colour corresponding
              |           % to each field in the above cell arrays.
              |
              .cmdResult  % The result returned by any command executed by the CMD
              |           % button.
              |
              .cmdText    % The text of the current command in the command box.
```

5.6 Fuzziness

Fuzziness is turned on by setting `par.selection.FAbsTol > 0`, `par.selection.FRelTol > 0`, or `par.selection.FRankTol > 0` in the setup file. The $FAbsTol$ and $FRelTol$ control parameters can be set interactively using the 'Fuzziness' button in the Analysis button group on the right side of the Analysis window (i.e. see Figures 5.7 and 5.8), which launches the 'Analysis Tolerances' interface shown in Figure 5.9.

Figure 5.9: The Analysis Tolerances window allows the user to interactively set tolerances (`par.selection.FAbsTol` and `par.selection.FRelTol`) to be applied to the analysis window. This interface is launched from the 'Fuzziness' button on the right side of the Analysis window.

The Analysis Tolerances interface is sometimes useful when exploring an optimal set because it allows you to dynamically change the fuzzy selection criteria used to select solutions for the optimal set. For example, if a tolerance is specified in the setup file that turns out to be too 'loose' when the run is analyzed (i.e. too high of a value for `par.selection.FAbsTol` or `par.selection.FRelTol`), then the solution set may look

like a very large blob that does not adequately exclude poor regions of the parameter space. To tighten your tolerances, you would decrease $FAbsTol$ to obtain a more localized optimal set. The fuzziness interface is useful because it effectively allows you to change your mind about $FAbsTol$ and $FRelTol$ *after the run is complete* to refine your selection criterion.

To use the Analysis Tolerances interface, simply change either $FAbsTol$ or $FRelTol$ and click the 'Set Tolerances' button to commit the change, and re-analyze the run by pressing the 'Re-Analyze' button in the Analysis window. This will bring up a dialog box asking whether to use the current $OptimalSolutions$ structure or re-load the History files. It is faster to use the current $OptimalSolutions$, but this must be done with care as discussed below. After choosing one of these options, the run will be re-analyzed, and you will be presented with a dialogue box indicating that the new $OptimalSolutions$ structure has not yet been saved to disk, but is available by loading a global variable called $OptimalSolutions2$ on the MATLAB command line. The dialogue box asks whether to save the new $OptimalSolutions$ structure as OptimalSolutions.mat. This must also be done with care, as discussed below.

The Analysis Tolerances interface has an important subtlety that users must be aware of when saving solutions after changing $FAbsTol$ or $FRelTol$. The subtlety is that *only solutions that are currently in $OptimalSolutions$ will be subjected to the new criterion.* This means that if you have previously made the analysis more restrictive by decreasing $FAbsTol$ or $FRelTol$ and saved the changes, and then select 'Use OptimalSolutins' rather than re-loading the History files, then *only solutions in the current $OptimalSolutions$ will be included in the analysis.* This can cause significant confusion when users notice that the optimal set fails to increase to its initial size, which looks like solutions have been somehow lost. As a general rule, you can select 'Use OptimalSolutions' if you have not saved changes previously, or if you are making the selection criteria more restrictive. If you have previously saved changes and are now relaxing the selection criterion (increasing $FAbsTol$ or $FRelTol$), then the History files must be re-loaded.

Chapter 6

Parallel Computing with Qubist

'There are 3 rules to follow when parallelizing large codes. Unfortunately, no one knows what these rules are.'

-W. Somerset Maugham & Gary Montry

Both Ferret and Locust are capable of parallel runs, which make use of a file system-based parallel computing system that I developed for Qubist. This does not rely on MATLAB's add-on parallel computing toolbox or any other third-party toolboxes. Everything you need for parallel runs is included with Qubist. Parallel computing with Qubist is as easy as clicking a few buttons on the Qubist node manager interface, but there are a few subtleties that you should be aware of. This chapter will introduce you to Qubist's parallel computing features and show you how to make full use of multi-CPU computers and clusters for your runs.

6.1 Introduction and History

Parallel computing with Qubist has had a somewhat convoluted history throughout the development of the code, which goes back to 2002-2003, when Ferret was the only optimizer in the package. Ferret-1 (circa 2002-2003) was in fact was a parallel code, which made use of Java RMI (Remote Method Invocation). This was a natural choice because:

- Communication overhead for a GA is very light, so Java's sometimes less than stellar performance is not really an issue.
- Any Java program is already a MATLAB program. There's no messing around with the external interface library. You just compile your Java and it acts just like any other MATLAB function.
- Java is (more or less) platform-independent. I needed a parallel computing solution that worked transparently in a heterogeneous computing environment, with Linux, Windows, and MacIntosh machines distributed across the network, and this made a Java-based solution an attractive option.

This early system was very simple, but neither robust nor user-friendly. The user was required to use multiple populations, and the parallel computing system was designed to send each population, via the RMI protocol, to a separate core for processing. When done, RMI sent the results back to the Ferret console node for processing. This approach was *very* limited because without any load balancing, the rate of processing was dominated by the slowest worker node that contributed to the calculation. Really, it was only useful when all of the processing nodes were equally fast and under no other significant load. This early code was also not very easy to use, had very little error recovery to deal with unresponsive (or crashed) nodes, and had no facility to disconnect or add new nodes once the calculation had started. I only built this rather fragile system for my own use to take advantage of a multi-core machine that I was using at the time, and also because I was interested in RMI. Nevertheless, it served my purposes at the time rather well.

I dropped the RMI approach in Ferret-2 because I was more interested in core GA development (specifically the linkage-learning problem, as well as a few others) and Ferret was evolving rapidly. The development of my simple RMI system simply did not keep pace with the rest of Ferret, and the package lost its parallel computing capabilities for about the next four years.

Recently, I've brought back parallel computing to Ferret-4 using a much more robust file-system based approach, and I have also ported the system to work with Locust. A similar approach is employed by two commonly used (and free) parallel computing MATLAB toolboxes called pMATLAB (http://www.ll.mit.edu/mission/isr/pMATLAB/pMATLAB.html) and MATLABMPI (http://www.ll.mit.edu/mission/isr/MATLABmpi/MATLABmpi.html). However, the parallel computing system is built in to Qubist and is not based on either of these packages, and no extra software is required to run parallel jobs with Qubist. Building Qubist's parallel code as a customized system embedded in the code means that it could be built as a relatively simple, light-weight system with a user interface specifically designed for Qubist. It does not require the generality of these other packages because it only manages parallel computing for Ferret and Locust. This file system-based parallel computing code performs very well due to the low communication overhead of GAs and particle swarm optimizers.

Qubist's new parallel computing system is analogous to a 'feeding frenzy': each node reads a work file that indicates what parameter sets (drawn from the population) are remaining to evaluate, reserves a chunk of the remaining work, and then it goes off to do its portion of the evaluation. When done, it writes information back to the work file to indicate which solutions it evaluated, and the actual solutions are saved in a scratch directory. I liken this system to a 'feeding frenzy' because it's completely asynchronous - worker nodes compete to reserve work chunks on a 'first come first serve' basis. Contrast this with the orderly RMI-based system that I described above. The new system never sits waiting idle for a slow node. If there is no work left that has not been reserved, then nodes that are finished jump in and start doing work that has been reserved, but not completed. The first node to finish all of the work sends a message to the others to indicate that the current set of solutions is fully evaluated, and that all other nodes should stop what they're doing and wait for the next batch of work. There is also a system in place that disconnects any node that has become unresponsive, and sends a signal to the unresponsive node to shut down if it is running.

The main challenge with this chaotic feeding frenzy approach is to very carefully control access to the scratch directory where the results are being accumulated. This is done using a lock file as a semaphore: i.e. a single file that indicates whether or not the scratch directory is currently being written to by another node. If the directory is locked, then all other nodes must wait until the node doing the writing indicates that it is done. This can be tricky, and I had plenty of problems with node lock-ups during development. These problems have all been solved, and the current system is very robust.

Qubist's parallel computing system is especially useful for problems with fitness functions that require more than a few tenths of a second per evaluation (i.e. for a single parameter set). A thorough parameter search and optimization may require many CPU-hours or even CPU-days of computation for such problems. This is not at all uncommon for scientific modeling because the underlying mathematical systems are often very complex. Some of my own applications in astrophysics require up to a CPU-month of computing, although a CPU-day or two is more typical. I develop another code called GalAPAGOS (Galaxy Astrophysical Parameter Acquisition by Genetic Optimization Software) at the University of Manitoba, which uses Ferret to fit astrophysical models to the rotating neutral hydrogen disks present in spiral galaxies. Figure 6.1 shows the results from a careful benchmarking experiment of Ferret's parallel computing system running GalAPAGOS models. The code was run eight times, using one to eight cores on an eight core server. The experiment was done a total of five times, corresponding to the five lines on the figure. The code ran for 100 generations each time, which was not long enough for the code to converge, but was sufficient to obtain accurate timing information. The left panel of the figure shows the results on a linear scale, and the right panel shows the same data on a logarithmic scale. The dashed line on the right panel shows a -0.8 power law for comparison. Note that a perfect parallelization scheme with no overhead would obey a -1 power law.

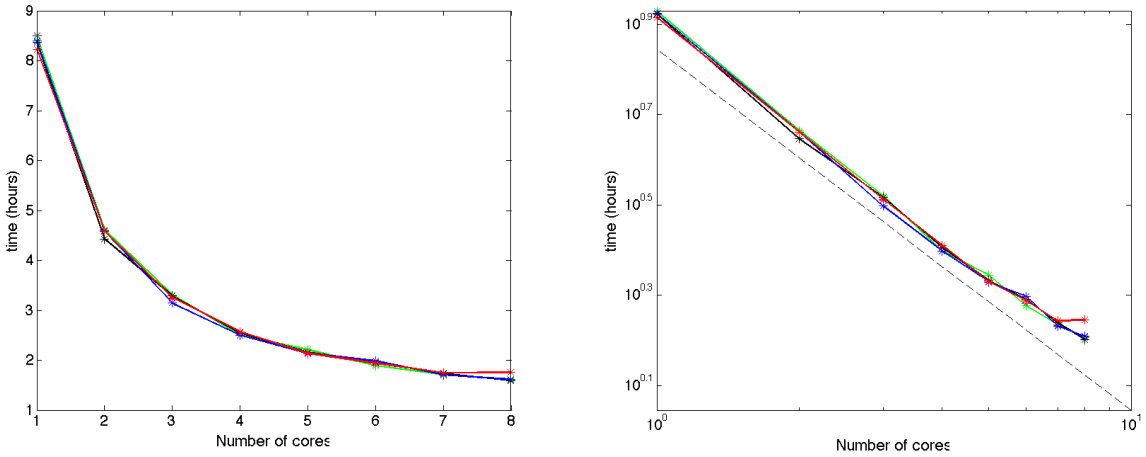

Figure 6.1: Results of a benchmarking experiment of Ferret's parallel computing system, using the GalAPAGOS astrophysical modeling code as the fitness function. The code was run eight times, using one to eight cores on an eight core server. The experiment was done a total of five times, corresponding to the five lines on the figure. Both panels show the same data, on a linear scale (left) and a logarithmic scale (right). The right panel shows a -0.8 power law for comparison.

The parallel computing system is *not* very useful for projects with fitness function that evaluate very quickly (i.e. an entire Ferret generation or Locust time step finishes in a few seconds), because in such cases the optimizer's internal calculations become the computational bottleneck. These types of problems finish quickly without parallelization in any case. If you are unsure about whether parallel computing is suitable for your problem, just try it. You do not have to modify your code, although one tiny addition discussed in Section 6.4.4 can help. If you notice a speed-up, leave Ferret or Locust in parallel mode, or fall back to single processor mode if not. Note that this can be done through a simple user interface, even while the code is running!

6.2 Starting a Parallel Run from Ferret/Locust

The easiest way to start a parallel run on multi-core machines is by starting the 'node manager' from the Control menu in the Ferret or Locust console window. This can be done after a project has been initialized, or at any time during a run. Starting the node manager will pop up a window like the one shown in Figure 6.2. This is a very simple interface, but it contains everything that you need to manage parallel runs.

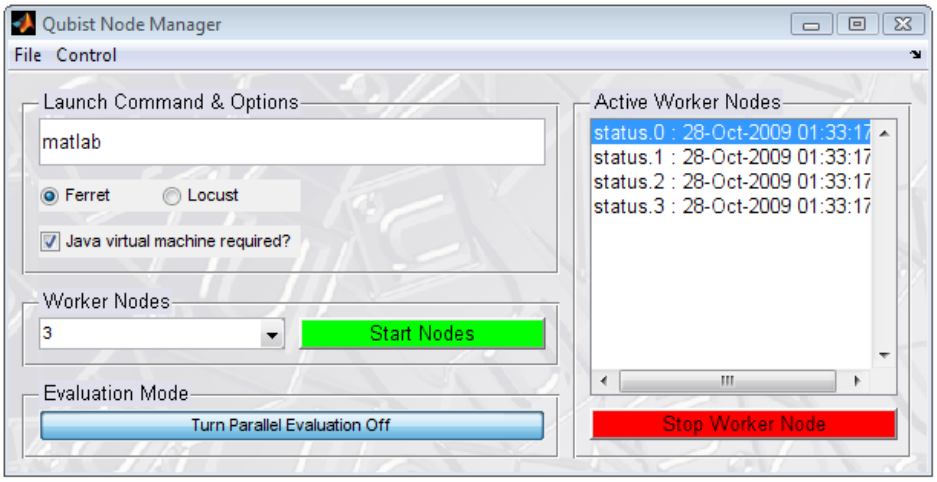

Figure 6.2: Qubist's node manager running on a Windows quad-core computer. The 'status.0' line refers to the console node, and the three other status lines refer to the three worker nodes. Note that each status line is followed by a time, indicating when each node last wrote to its status file to indicate that it is still alive.

If you plan to run on multiple computers, rather than a single multi-core machine, the nodes that are intended to run on the same machine as the Ferret or Locust console can be started directly from Ferret or Locust, but the nodes on other machines must be started from the Qubist Node Manager, which can be launched from the component selector using the `launchQubist` command. Runs involving a cluster of machines are discussed in Section 6.3.

Before you begin to experiment with parallel runs, I recommend turning off MATLAB's built-in multi-threading by unchecking the box labelled 'Enable multithreaded computation' in the MATLAB Preferences menu.[1] This option can be accessed from the MATLAB desktop from the File → Preferences menu, and looking for 'Multithreading' under the tab labelled 'General'. Parallel processing in Qubist *will* still work with multithreading turned on, but this is less efficient, especially if you plan to 'max out' your machine by using the node manager to match the number of MATLAB processes to the number of cores on your system (see below).

[1] The 'Enable multithreaded computation' option is not present in the R2009b version of MATLAB. Don't worry too much about this if the option is not available. It's somewhat better to run parallel Qubist jobs with the multi-threaded option disabled if you can, but the difference is not *that* great, and it is certainly not going to make or break your calculation. It is possible to start MATLAB in single-thread mode using the '-singleCompThread' startup option, but I have not experimented enough with this option to recommend it.

6.2.1 The Launch Command and Options Panel

Each *worker node* contributing to the calculation runs in a separate MATLAB session. These extra MATLAB processes run in the background under the MacIntosh and Linux operating systems, but you will see extra windows appear under Windows. The node manager must be given the command that starts MATLAB on your system from a command line (Windows), Terminal window (MacIntosh), or shell (Linux).

Examples of Launch Commands

The 'Launch Command & Options' panel is near the top of the node manager GUI. Here are some examples of the commands that I use to start MATLAB on some of the systems that I have used:

- On my Windows computer, the default command 'matlab' works.
- On Linux, the default command 'matlab' also works, if the command is on my path. However, I sometimes use machines where it is not directly on my path. On these machines, I qualify the 'matlab' command with the full path: '/user/local/matlab/bin/matlab' for example. Of course, the path will vary from one system to another, so you should consult your system administrator to obtain your full MATLAB path if you don't know it.
- On my MacIntosh computer, the default command 'matlab' works with the R2009 MATLAB versions. However, I required a more elaborate command for the R2008a 64-bit beta version: 'export MACI64=1; /Applications/MATLAB_R2008a/bin/matlab'. The 'export' part of the command is required to set a system variable that is required to launch MATLAB from the command line with this version.[2] Starting older MATLAB versions would require a similar command, except that the 'export' part would not be required. Note that the exact command will vary from one computer to another and possibly even from one MATLAB version to another. A full path to MATLAB's location within your Applications folder may be required for pre-R2009 versions.

The basic rule is that the launch command box should contain whatever command you use to start MATLAB from your shell in Linux, Terminal in Mac OS X, or the command window (CMD) in Windows. If you are able to launch MATLAB from a command line by typing this command, then it should work with the Qubist node manager.

Is Java Required?

Java is tightly integrated with the MATLAB language and some projects might be written partly in Java. The node manager contains a check box labelled as 'Java virtual machine required?'. If your project contains any explicit calls to Java, you should check this box. If not, then you can uncheck this box to save memory, although leaving it checked will do no harm.

[2] If you are a Mac user and haven't tried the new 64 bit versions, you should. In my experience, they are much faster than the older 32-bit versions.

6.2.2 The Worker Nodes Panel

The 'Worker Nodes' panel contains a drop-down menu where you can select the number of worker nodes to launch. *Note that the numbers here refer to worker nodes only, and do not include the main MATLAB session running the Ferret or Locust console, which will also be engaged in evaluating your fitness function.* Usually, the number of nodes selected should not exceed the number of cores on your system. Therefore a single worker node is reasonable for a dual-core system, up to three workers for a quad-core system, seven workers for an eight-core system, etc. Chances are good that you will crash your computer if you try to launch 127 workers on your dual core laptop, so beware. This panel contains a button labelled 'Start Nodes', whose purpose is self-explanatory; worker nodes will not actually be started until this button is clicked.

Your decision regarding the number of nodes is not final. You can launch more at any time using the method outlined above, shut them down using the 'Stop Worker Node' button, as discussed below, or pause parallel processing at any time during your run without re-starting.

6.2.3 The Evaluation Mode Panel

The 'Evaluation Mode' panel contains a single button that allows you to flip between parallel and single-process evaluation. Turning parallel evaluation off here will not shut down your worker nodes. However, they will go into an idle mode where they use minimal CPU resources. Another click of the same button will turn parallel evaluation back on. A good use for this button is if you need to do something else briefly on a system that is very busy with your Qubist run. However, it is usually better to just shut down a node or two if you need your system for other purposes for more than a few minutes. Note that you can always restart nodes later.

6.2.4 The Active Nodes Panel

The node manager contains a panel labelled 'Active Worker Nodes', which lists the nodes that are currently available to evaluate your fitness function. Every node is assigned a node number when it is started, and each node frequently 'touches' its status file in the project's scratch directory to updates its time stamp and indicate that it is alive. The status files present, along with their timestamps, are shown in this panel. Note that 'status.1' refers to the status file for worker node 1, 'status.2' to the status file for worker node 2, etc. The process running the Ferret or Locust console is not counted as a worker node. However, it is assigned node number 0, and shows up in this window as 'status.0'.

If a node runs into a problem and crashes or hangs, then the Qubist node manager will normally recover by disconnecting it and continuing on with the remaining nodes. There is a FerretSetup (or LocustSetup) line `par.parallel.timeout=n`, which specifies the time n in seconds to wait for an unresponsive node before disconnecting it and attempting to kill the process. Don't be too greedy with this setting. You can afford to wait fifteen seconds or a minute for an unresponsive node to recover if your run requires hours or days to complete. Setting this low (i.e. less than a few seconds) is fine if there are no other processes running on your computer and it runs MATLAB perfectly without ever swapping to disk or temporarily hanging. Perhaps your system is that well-tuned but mine certainly isn't, and I find it useful to set the timeout parameter to at least fifteen seconds to avoid unnecessary disconnections.

You can stop a node manually at any time by selecting it in the window and clicking the 'Stop Worker Node' button. You should briefly see a line 'node.n.killed', where n is the target node. The .killed file is a

signal from the GUI to shut down the target node. The target status file, as well as the corresponding .killed file should disappear if the node shuts down cleanly. If an unforeseen problem occurs and the node does not shut down cleanly, then these files may not disappear from the list. Your run will not be affected by this, but it is possible (though unlikely) that you may still have an idle MATLAB process running. You can kill this process from the operating system, or you can just leave it alone, since it probably isn't using many system resources anyway.

Finally, please note that node numbers are not recycled. If you start node number 1 and shut it down, then the next node that you start will be number 2. I have received bug reports regarding this behaviour, but it is actually by design. It is simply too risky to recycle node numbers because a serious error would result if a node failed to shut down properly and a new one was started with the same number - two active nodes with the same identification number would wreak havoc on Qubist's parallel computing system and almost certainly cause it to hang. It is much better to always use a fresh node number because then a node that fails to shut down after being sent a '.killed' signal, for whatever reason, can do no harm.

6.3 Parallel Runs on Multiple Machines

It is easy to set up parallel runs on multiple machines, but the procedure is slightly different than described above. There can only be one Qubist console (node-0) for the run, so you should choose the fastest computer in your cluster and start Ferret or Locust on that machine only. You should start one MATLAB session only on each other machine and start Qubist using your launchQubist.m file. You should select the 'Qubist Node Manager', which will bring up the same node manager interface that you get when you start it from Ferret or Locust, except that it will include menus to load demos and projects. By selecting a project, you are essentially telling the node manager what project you would like it to join. Once you have done this, you can use the node manager in the same way discussed in Section 6.2.

One small complication can occur when using multiple machines on a shared file system if there is significant latency when writing files. If you start seeing messages in your MATLAB window like 'Unknown error calculating fitness function. Re-evaluating N solutions.', then you should tell Qubist to pause briefly each time it writes to the scratch directory by adding the following line to your setup file: `par.parallel.latency=n`, where n is the pause duration in seconds. These errors are harmless and should not crash your run. However, if you see it happening frequently, then you should try to avoid it by increasing the latency parameter because re-evaluating solutions is costly.

6.4 Common Parallelization Gotchas

There are a couple of problems that come up occasionally with parallel runs. You need to be aware of these issues you plan to run in parallel mode, especially if you are a Linux user, or if your project requires a lot of memory.

6.4.1 Use a Local Disk When Possible

Reading and writing files on a local disk is normally *much* faster than accessing files on a shared disk over a network. Qubist's file system based parallelization method will therefore perform best if your run's data directory is on a local disk connected to the machine running the Ferret console. If your run involves

worker nodes running on multiple machines, then you can minimize the parallelization communication overhead by placing your data directory on a disk connected to the machine hosting the most worker nodes, and also running the Ferret console from the same machine.

6.4.2 'Nicing' Down Runs

The Linux and MacIntosh operating systems allow users to 'nice down' runs so that other jobs take priority on the CPU. This is especially useful if other users are also trying to use the machines that are contributing to your parallel calculation. In this case, you should be aware that all nodes that you start using the node manager will take on the same priority as the node that you used to start the MATLAB process running the node manager. You should never include a 'nice' command in the node manager's command line. For example 'nice +19 matlab' won't work. If you want to nice down your nodes, nice down the MATLAB process running the node manager before you start nodes, or 'renice' them from the Linux command line afterward.

6.4.3 Memory Problems

If your project requires a lot of memory, or your machine is very memory deficient, then you should be aware that nodes do not share memory, so your memory requirements increase in proportion to the number of nodes. This is unfortunate, but the problem is that every node runs in a separate MATLAB process, and I'm not aware of any technique that allows multiple MATLAB processes to share memory. If you have this problem, you will probably notice one of the following behaviours:

- Nodes take a very long time to start, or may never start.
- Ferret or Locust slows right down. Your entire system may respond poorly, and you may even hear your disk drive making a lot of noise. This is symptomatic of running out of memory, and MATLAB trying to continue by using the swap file. This is not good, and you'd might as well stop the run because progress will be very slow.

To remedy this situation you can try to decrease your project's memory usage, install more memory, or distribute your run over multiple machines.

6.4.4 The 'isAbortEval' Directive

Qubist's parallel computing system works by assigning work chunks to worker nodes, and assembling the set of solutions returned by the workers as they finish. Workers are never allowed to sit idle when work still needs to be done; if a worker finished early, it is assigned a new work chunk, that may have also been assigned to a slower worker that has not yet finished. Therefore, it is a good idea to halt any remaining evaluations that worker nodes might still be executing, once Ferret or Locust has received the results from all of the fitness evaluations requested.

This is done by monitoring for the 'abortEval' signal, and placing a break point in the fitness function that is triggered if `isAbortEval(extPar.status)` returns *true*:

```
function [F,auxOutput,saveData,XMod]=fitness(X,extPar)
% Qubist demo fitness function.

NIndiv=size(X,2);
for i=NIndiv:-1:1
    if isAbortEval(extPar.status)
        F=[]; auxOutput={}; saveData={}; XMod=[];
        break
    end
    [F(i),auxOutput{i},saveData{i},XMod(:.i)]=calcFitness(X(:,i),extPar);
end
```

If `isAbortEval(extPar.status)` returns *true*, then this means that Ferret or Locust no longer needs the remaining fitness values that are still being computed, because another worker node finished evaluating them first. Therefore, you should just return empty variables of the appropriate type for F, $auxOutput$, $saveData$, and $XMod$. Of course, you can omit any that your fitness function does not use.

Chapter 7

Self-Optimization of Ferret's Strategy Parameters: Auto-Adaptation and Modifying Values Manually

'...Ferret is a complicated machine with a lot of moving parts, and modifying parameters by hand while it's running is a bit like replacing some of your car's engine components while you're driving down the highway - if you touch the wrong moving part, disaster is the likely outcome.'

Section 4.7 briefly discussed the Ferret setup file, and the settings of each strategy parameter will be covered exhaustively in Chapter 8 below. If you have explored some of the demos, then you probably have a good idea by now that there are a lot of settings that can be changed. I have tried to provide good default values for the parameters and to ensure that Ferret is robust against failure over a wide range of 'sensible' parameter choices. The net effect is that the precise value of *most* parameters is not usually all that important, and you should not have to spend a lot of time fine-tuning your setup file. However, there are a few parameters whose settings can be quite sensitive, but Ferret does not burden users with fine-tuning them. Rather, Ferret's auto-adaptation system is designed to vary potentially sensitive parameters, collect real-time performance data as they are varied, and use the data acquired to fine-tune and optimize them. Ferret does this by building an internal probabilistic model of its own performance and maximizing it over the parameters that are allowed to auto-adapt.

This chapter discusses Ferret's auto-adaptation interface, which allows users to monitor and interact with the auto-adaptation system. Note that only a few important parameters are handled by auto-adaptation at this time. However, I also show how *any* control parameter can be modified manually during a run, without stopping it, using Ferret's 'Modify FerretSetup' interface.

7.1 Auto-Adaptation

The purpose of this section is to outline the use of Ferret's auto-adaptation system. This section is short due to the 'auto' in 'auto-adaptation' - other than the initial setup of the auto-adaptation options in the

setup file (discussed in Section 8.4), there is really not much for the user to do with the auto-adaptation interface, other than observe how Ferret has optimized your settings. Ferret's auto-adaptation system is designed to learn from experience and to maximize the probability that a strategy parameter will have a positive effect on the run's development. Figure 7.1 shows the auto-adaptation interface in action. The text panel on the left gives the current *mean* values for all strategy parameters that can be controlled adaptively, and whether they are currently under adaptive control or are 'frozen' (set to a fixed value). The initial state of each parameter is given by the settings in `par.strategy`; see Section 8.4 for details.

The dropbox under the text panel allows you to select a strategy parameter. The current parameter in this case is the mutation scale, which is the average RMS (root mean square) size of mutations, as discussed in Section 8.7. The graphs on the right hand side and the buttons on the bottom show the current probability distribution function and history of the selected parameter over the course of the run. The top figure on the right is a histogram showing the probability of that a particular value will be chosen; this graph and the text box to its left show that values chosen for `par.mutation.scale` follow a distribution centred on a value of 0.0161 at present, and that the probability falls to zero for larger values. The bottom right figure shows the history of the strategy parameter throughout the run. We see the mutation scale started with an average value of 0.25 (from the user's setup file) but dropped to its present value by about generation 400. An overall decrease in the mutation scale makes sense for most problems because large mutations are often less useful late in the run after the population has mostly converged. Note that the gradual drop in mutation scale is punctuated with occasional and very abrupt jumps, when for some reason, a few relatively large mutations 'get lucky' and find good solutions. This surprising behaviour is commonly seen when the mutation scale is under adaptive control, and is actually quite useful because it very effectively stirs up the population once in a while, even after it has mostly converged. This helps to continue exploration even late in the run, and combat against genetic drift, a common problem of genetic algorithms that I discussed in Section 2.2.3.

The distribution is similar to the positive side of a Gaussian centred on zero, except for a sharp dip at the origin, which is useful because mutations of size nearly equal to zero are pointless. Note that Ferret suppresses very small mutations, even when the distribution is forced to be a Gaussian in the auto-adaptation interface, as discussed below.

The buttons below the graphs can be used to toggle the current strategy parameter, shown in the drop box, between full *non-parametric* adaptation (using the probability distribution shown in the top right figure), freezing the parameter at it's present value, or allowing it to adapt, but forcing the distribution to be Gaussian. Generally, I prefer fully non-parametric, auto-adaptation. It is also possible to 'tweak' the distribution using the arrows in the upper corners of the top right figure. These arrows allow you to move the boundaries of the figure to effectively shift the distribution function to higher or lower values. I don't generally do this, and I am not aware of any users who do this regularly, but the option is there if you feel that that auto-adaptation has chosen a distribution centred on values that are too high or too low for your problem. Note that dropping the mutation scale, crossover dispersion, or crossover strength to very low values by hand could result in premature convergence. It is generally safe to increase the values for these parameters, but decreasing them should be done with caution. Please refer to Sections 8.7 and 8.8 for advice on setting these parameters.

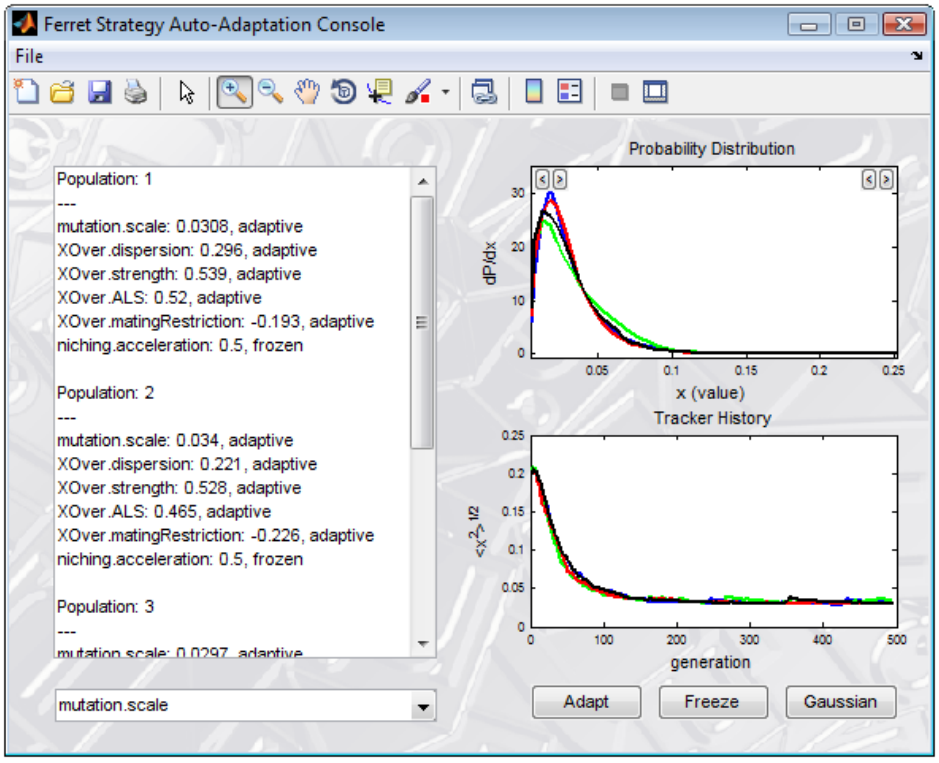

Figure 7.1: Ferret's strategy auto-adaptation interface allows users to enable or disable the adaptation of parameters, watch how they are optimized, and tweak the distributions used to choose values.

7.2 Modifying Parameters During a Run, and Ferret's Polish Mode

Ferret contains a feature for advanced users, which allows them to change *any* strategy parameter loaded from the setup file, and inject the new value into the run, *while Ferret is running*. Frankly, this scares the willies out of me if I'm doing an important run, and Ferret will present any user attempting this with a stern warning that is meant to instill a certain amount of trepidation. The problem is that Ferret is a complicated machine with a lot of moving parts, and modifying parameters by hand while it's running is a bit like replacing some of your car's engine components while you're driving down the highway - if you touch the wrong moving part, disaster is the likely outcome.

Some parameter changes are known to be safe, like the changes made by the special 'Polish Mode' discussed in Section 7.3. For other parameters, I have included enough validation to prevent changes to parameters that I *know* will crash a run, but users often break things in ways that surprise me, and I have no doubt that this could happen now and then when modifying parameters on the fly. Please consider this feature to be experimental for now, and use it with caution, or not at all. Some common sense is warranted here; for example, changing the maximum number of generations, mutation scale, selection pressure, or niching parameters should be OK, but changing the project directory would be a crazy thing to attempt.

Figure 7.2 shows the 'Modify FerretSetup' interface and the warning presented to users when using this feature. Here, I show the maximum number of generations (`par.general.NGen`) being increased to 1000, which is an entirely safe change to make. After typing the new value in the textbox, the normal procedure is to press the 'Validate Value' button to run the change through the same validation function that Ferret uses to check setup files when starting a run. This button saves changes but does not send the new value to Ferret, so further changes can be made by repeating this procedure. Once all necessary changes have been made, you can either press the 'Discard Changes' button to cancel your modifications, or click the 'Validate and Save' button to validate once more and inject the modified setup parameters into your Ferret run.

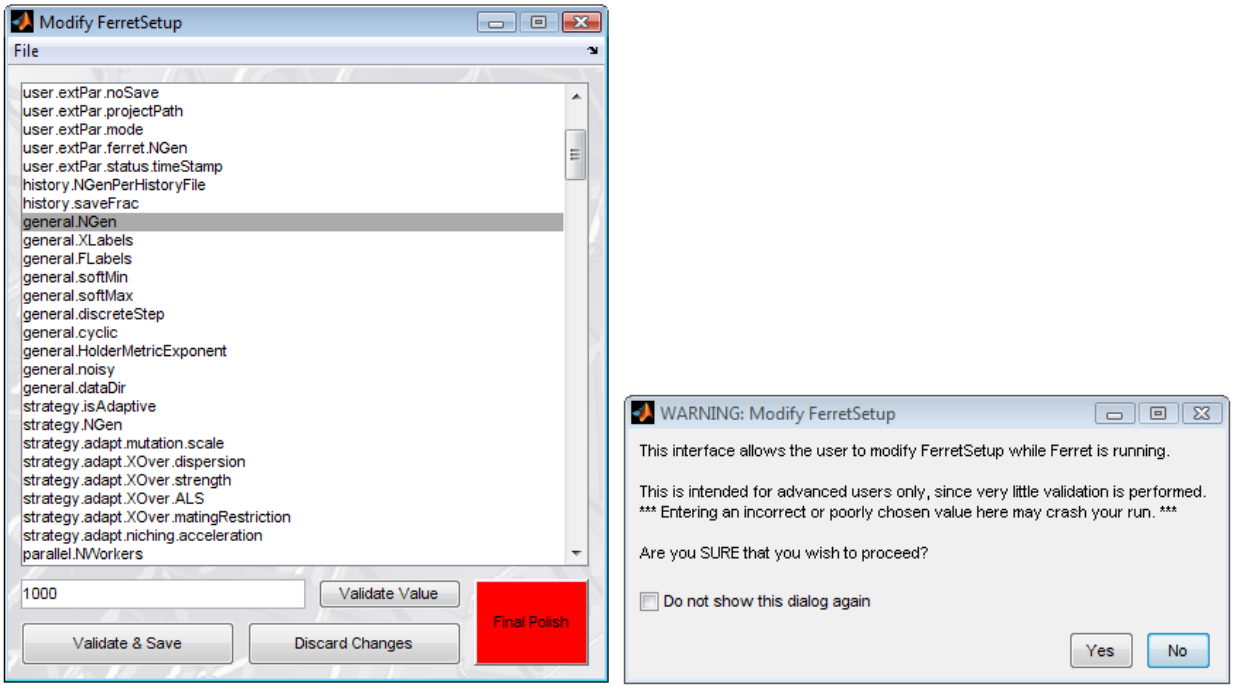

Figure 7.2: The 'Modify FerretSetup' interface (left) and a stern warning presented to users when using the 'Modify FerretSetup' interface (right).

7.3 Ferret's Polish Mode: Using Ferret as its Own Polisher

The 'Modify FerretSetup' interface also contains a big red button labelled 'Final Polish'. This is an interesting feature that you can use near the end of the run, which turns off most of Ferret's parameter exploration features and puts the code into a very aggressive mode that zeros in on the best solution or solutions in the parameter space. Qubist contains other optimizers that can be used to polish Ferret's optimal solutions (Anvil, SAMOSA, and SemiGloSS), but I think this is quite an elegant feature that effectively turns Ferret into its own polisher. Specifically, the following changes are made to the strategy parameters:

```
par.strategy.isAdaptive=false;
par.selection.PCompete=1;
par.selection.pressure=1;
par.selection.BBPressure=1;
par.selection.FAbsTol=0;
par.selection.FRelTol=0;
par.selection.FRankTol=0;
par.selection.shareFitness=0;
par.XOver.dispersion=0;
par.link.PLink=0;
par.link.FAbsTol=0;
par.link.FRelTol=0;
par.link.FRankTol=0;
par.link.PRandomizeNewBBs=0;
```

These changes shut down Ferret's auto-adaptation and linkage-learning system, turn off all fuzzy tolerance parameters ($FAbsTol$, $FRelTol$, and $FRankTol$), set the competition probability to 100%, and turn the selection pressure to maximum. If you are running a single-objective parameter-space mapping data-modeling problem like the 'Data-Modeling' demos, you will see the cluster of points mapping the one sigma χ^2 surface rapidly collapse to a single point near its centre, as Ferret goes hunting for the single lowest χ^2 in the parameter space.

Note that the polish mode modifications given above are in the Qubist user directory and are therefore modifiable by users who want to customize their Polish Mode settings. The relevant file is [Qubist_Home]/user/Ferret/skins/[mySkin]/setup/modifyFerretSetup.m, where [mySkin] is 'Mac', 'Linux', 'Windows' or 'Minimalist'.

Despite my warnings in Section 7.2, I regard Polish Mode to be safe. Use it whenever you wish.

Chapter 8

FerretSetup Parameters

> *'One of the universal rules of happiness is: always be wary of any helpful item that weighs less than its operating manual.'*
>
> -Terry Pratchett, *Jingo*

Writing a software user's guide is a lot less fun than writing the software in the first place, and more than any other, this is the weighty chapter that I have dreaded to write. This is the chapter where I will undertake to explain the entire list of Ferret's control parameters in more or less the same order that they appear in the defaultFerretSetup.m and FerretSetup_fullTemplate.m files, which respectively reside in the [QubistHome_]/user/Ferret/defaults and [QubistHome_]/user/Ferret/templates directory.

Ferret has evolved tremendously over the years and I often find myself offering advice on the use of the code that blatantly contradicts advice that I would have offered even couple of years ago for an earlier version of the code. A quote from my favourite author Douglas Adams, from the introduction to the complete edition of the 'Hitchhiker's Guide to the Galaxy' that permanently resides in my office, seems particularly appropriate:

> *'...the publication of this omnibus edition seemed like a good opportunity to set the record straight – or at least firmly crooked. Anything that is set down wrong here is, as far as I'm concerned, wrong for good.'*

This user's guide is perhaps not quite *that* final, but I do hope to offer advice in this chapter that will remain valid for at least the next couple of major versions of my code. So without further procrastination, let us begin.

8.1 par.user

This is the first set of parameters in the defaultFerretSetup file, and this is where the user tells Ferret the name of their fitness function and the optional output function:

```
% User
par.user.fitnessFcn='fitness'; % [string]: Name of the fitness function.
par.user.output=''; % [string]: Name of optional user-defined output function called ↘
        ↪ each generation.
```

Both of these control parameters should be strings, and *not* function handles. The setup and output files are discussed in Sections 4.7 and 4.9 respectively. The fitness function is mandatory, unless your project uses the default file name 'fitness.m'. The output function is optional, and the default is an empty string: `par.user.output=''`, which indicates that no output function is specified.

Note that the project's init function is called before the setup file, so the name of the init function is not given in `par.user`. The name of the init function should be included in your launchQubist.m file, as discussed in Section 4.4.1.

8.2 par.history

This group of parameters specifies the location of the data directory for your run and controls how Ferret writes History files. History files are written in the 'History' subdirectory of the run's data directory, and contain all of the information necessary to analyze or resume a run, or to play a movie showing how the run progressed.

```
% History
par.history.dataDir='FerretData'; % [string]: Directory for Ferret History files, etc.
par.history.NGenPerHistoryFile=25; % [integer >= 1]: How many generations per ↘
        ↪ History file?
par.history.saveFrac=0; % [0 - 1]: Save only the optimals --> [0], ↘
        ↪ or a fraction [0 - 1].  See also par.analysis.keepFrac.
```

Ferret's data directory is set by the string specified in `par.history.dataDir`. This can be a local path (as shown here), in which case the data directory will be saved in the project directory. It is also possible to specify a global path by including the full path in the *dataDir* string. I often use a global path and set it to a run directory outside of my project directory, because this keeps my run data separate from my program files. This is especially useful when a project is under rapid development, because it is then possible to archive snapshots of the program directory for backup purposes without including a lot of run data.

Each History file is numbered History-1.mat, History-2.mat, etc., and each one contains a cell array of length `par.history.NGenPerHistoryFile`. Each cell represents a single generation of the run and contains the same fields as Ferret's top-level *world* data structure, as outlined in Section 4.9.1, as well as other information used internally by Ferret. The full structure of a History file is given in Section 4.16.4.

The generation number saved in `History{`n`}` is given by `History{`n`}.par.bookKeeping.gen`. Normally, the user does not need to access the history files directly, but a great deal of information is saved in these files if you need it. As a result, Ferret's data directories can become quite large. However, the 'noSave' feature described in Section 4.6 provides a useful technique for reducing the amount of information that is stored.

The `par.history.saveFrac` parameter controls how many solutions are saved in each History cell. By default, `par.history.saveFrac=0`, which causes Ferret to save only optimal solutions plus elites. This is the most conservative setting in terms of disk usage, but sufficient information is saved to construct the optimal set at the end of the run and play a reasonably detailed movie showing its progress. If the

saveFrac parameter is set to 1, then *all* solutions are saved each generation, so it is possible to play a movie of the run or otherwise analyze it in perfect detail. This uses a lot more disk space and is usually not necessary, but it is sometimes useful to diagnose problems or to construct a nice movie for a presentation. Note that *saveFrac* can also be set to intermediate values between 0 and 1, in which case the optimals and elites are saved, plus other solutions with low rank number[1], up to the specified fraction of the population.

8.3 par.general

The `par.general` branch of the setup file contains the most basic information about the mathematical structure of your problem and how you want to control the run.

```
% General
par.general.NPop=1; % [integer >= 1]: Number of populations.
par.general.popSize=500; % [integer >= 1]: Size of each population.
par.general.NGen=100; % [integer >= 1]: Maximum number of generations to run for.
par.general.XLabels=; % [Cell array of strings]: ↘
        ↪ Give names to some or all parameters: 'A','B',...
par.general.FLabels=; % [Cell array of strings]: ↘
        ↪ Give names to some or all fitness values: 'FA','FB',...
par.general.min=[0, 0]; % [real vector]: Minimum values of all parameters.
par.general.max=[1, 1]; % [real vector]: Maximum values of all parameters.
par.general.softMin=[]; % [integer vector > 1]: Which parameters have soft minimum ↘
        ↪ bounds?
par.general.softMax=[]; % [integer vector > 1]: Which parameters have soft maximum ↘
        ↪ bounds?
par.general.discreteStep=[]; % [real vector >= 0]: Discrete stepsizes of some ↘
        ↪  parameters?  0's or [] --> continuous.
par.general.cyclic=[]; % [integer vector > 1]: Which parameters are cyclic?
par.general.HolderMetricExponent=2; % [real]: Holder metric for determining distances.
par.general.noisy=0; % [0 or 1]: Does the fitness function contain noisy fluctuations?
par.general.useMex=false; % Use mex to speed up execution?
```

`par.general.NPop` specifies the number of populations in Ferret's *world*, and `par.general.popSize` specifies the size of each population. For most problems, I use only a single population, but it is sometimes advantageous to run multiple populations. Note that multiple populations do not interact via crossovers and they do not compete with each other. They only interact via Ferret's 'immigration' operator, which is controlled by the probability specified by `par.immigration.PImmigrate`, as discussed in Section 8.11. Multiple populations can sometimes help to explore the parameter space of very difficult problems by effectively decreasing the selection pressure; when an improved solution is found, it tends to spread rapidly within its own population via selection and crossover, but its propagation to other populations will be delayed if the immigration probability is low. This sets up an effective barrier between populations, which gives them extra time to explore on their own before committing to an improved solution.

There is a secondary advantage to using multiple populations in that it reduces Ferret's memory requirements slightly and improves the speed of its internal distance metric calculations. Because Ferret uses a niching technique to spread solutions over the optimal region of the parameter space, it must

[1] Recall that solutions with 'low rank' are the best solutions in the population. The optimal set has $rank = 1$.

maintain an internal array for each population, which contains the distances between all solutions belonging to that population in the parameter space. The size of each of the $NPop$ arrays is given by $popSize^2$. Thus, $NPop \times popSize^2$ distances must be stored for a total size $N = NPop \times popSize$. Therefore, N individuals require $N^2/NPop$ distance array elements, which shows that distributing the total number of individuals over $NPop$ populations effectively reduces the memory requirements of the distance metric by a factor of $NPop$. For most problems, this is not a major consideration unless you are using a very large population or running on a computer with very modest memory. Of greater importance is that fact that the time required to compute distance metrics inside Ferret increases in proportion to the number of array elements. Therefore, you can often decrease both the memory requirements and the computational overhead of Ferret's internal operations by decreasing the population size and using multiple populations.

Normally, I start with a single population with a size in the range of 250-500 and modify this as necessary. Ferret is not especially sensitive to the population size, as long as it is 'big enough'. You will not be able to explore the parameter space very thoroughly if it is set too low; below about 100 is dangerous for most problems, because the parameter space will probably not be explored adequately before the population converges. However, very small populations are often very useful for testing because you can cycle through many generations quickly. *As a general rule, you should increase the population size or the number of populations if you start seeing inconsistent results between runs.*

`par.general.NGen` specifies the maximum number of generations to run until completion. Of course, runs can be stopped manually by pressing the 'Stop Run' button on the Ferret console window at any time, by setting stopping tolerances as outlined in Section 4.5, or by specifying a user-defined stopping criterion as discussed in Section 4.9.2. Usually, I just set $NGen$ to some large number like 10000, and stop the run manually once Ferret's graphics show that improvement has stopped, or that a stable population has formed and the results displayed in the analysis window look sufficiently well filled in.

`par.general.XLabels` and `par.general.FLabels` are cell arrays of strings that represent the names of the parameters and objectives of your problem, and these strings are used to label the graphs generated by Ferret. The order must match the order of the parameters specified by `par.general.min` and `par.general.max`. For example, in the binary star modeling problem discussed in Section 5.1, I used the following:

```
par.general.XLabels={'\phi', 'inclination (degrees)', 'eccentricity',...
     'mass_1 (M_{sol})', 'mass_2 (M_{sol})', 'semi-major axis (AU)'};
```

I did not actually set par.general.FLabels, but a reasonable choice would have been:

```
par.general.FLabels={'\chi^2_{reduced}'}.
```

Note that these labels are sent directly to MATLAB's xlabel and ylabel commands when the figures are generated. LaTeX symbols are allowed and should be processed correctly when MATLAB renders the figure.

If the length of the $XLabels$ cell array exceeds the actual number of parameters, as determined by `par.general.min` and `par.general.max` (see below), or the length of the $FLabels$ cell array exceeds the number of objectives returned by the fitness function, then the extra labels are ignored. Parameter names not specified by $XLabels$ and $Flabels$ are filled in with default values like $X1$, $X2$, etc. for parameters, and $F1$, $F2$, etc. for objectives.

`par.general.min` and `par.general.max` are row vectors that specify the minimum and maximum values of each parameter of your problem. These vectors must be the same length, and each entry of `par.general.min` must be *strictly* less than the corresponding value of `par.general.max`. It is an error if any value in max is less than or equal to the corresponding value in min. Ferret uses the length of these

vectors to determine the number of parameters in your problem.

`par.general.softMin` and `par.general.softMax` are vectors of integer indices that specify which parameters (if any) are allowed to have 'soft' boundaries. For example, `par.general.softMin=[1,3,5]; par.general.softMax=[5,7]` tells Ferret that parameters 1, 3, and 5 have soft lower bounds, while parameters 5 and 7 have soft upper bounds. Populations are always initialized with values in the ranges specified by `par.general.min` and `par.general.max`, but if a boundary is declared to be soft, then Ferret is allowed to explore outside of these ranges to find improved solutions. The $softMin$ and $softMax$ parameters are empty sets by default, so that the ranges specified by `par.general.min` and `par.general.max` are strict. Note that soft boundaries are meaningless for cyclic variables (see `par.general.cyclic` below) and are ignored.

`par.general.discreteStep` is a vector of real numbers greater than or equal to zero that can be used to force parameters to take on discrete values. Each number in the vector specifies the step size of the corresponding parameter in `par.general.min` and `par.general.max`. Values of zero or missing elements in the vector indicate continuous variables. Parameters can take on values equal to `par.general.min` and increasing in steps equal to the values in discreteStep. For example, if `par.general.min=[0, 0.5, 1]`, `par.general.max=[3, 3, 3]`, and `par.general.discreteStep=[0,1]`, then the first and third parameters are treated as continuous, and the second parameter is allowed to take on values of 0.5, 1.5, and 2.5 only. Values in discreteStep should always be greater than or equal to zero, but the absolute value is used if any values are negative.

`par.general.cyclic` is a vector of integers, specifying the indices of parameters that are considered to be cyclic or periodic, such that the maximum value wraps back onto the minimum value. Such variables usually represent angles, where 2π radians (360 degrees) is the same as 0 radians (0 degrees). For example, if the second and third parameters of a three-parameter problem represent angles in radians, then one might indicate this with the following lines:

```
par.general.min=[0, 0, 0];
par.general.min=[1, 2*pi, 2*pi];
par.general.cyclic=[2, 3];
```

It is not usually a disaster if you forget to include an angle variable in `par.general.cyclic` if the solution turns out to be well away from 0 and 2π. However, if the solution is near 0 or 2π, problems can occur if the variable is not designated as cyclic. The problem is that if the run begins to converge on the wrong side of the $0/2\pi$ boundary, it won't be able to drift over to the other side if the variable is not cyclic, and will probably end up stuck on or near the boundary.

Another significant problem occurs if the optimal solution set straddles the $0/2\pi$ boundary for a parameter space mapping problem, if the parameter has not been designated as cyclic. This is very inefficient because crossovers will occur between the two sides of the parameter space, and many poor quality solutions ('lethals') will result when offspring solutions are formed at intermediate angles well away from $0/2\pi$. This is easily fixed by designating the parameter as cyclic, because then crossovers will take the 'short way' across the $0/2\pi$ cut. Very few lethals will be generated, and you will obtain a better mapped solution set with fewer evaluations.

I am always suspicious of solutions that are very close to any boundary in my problems. When I see this happening, I ask questions like:

- Am I sure about the *min* and *max* values?
- Did I make a mistake with my units? Note that this could cause values to be off by orders of

magnitude.

- Are any of these parameters cyclic?

`par.general.HolderMetricExponent` is the Holder metric exponent p used to calculate the distance $d_{i,j}$ between vectors $\mathbf{x}^i$ and $\mathbf{x}^j$ for niching:

$$d_{i,j} = \left[\sum_{n=1}^{N} \left(x_n^{(i)} - x_n^{(j)} \right)^p \right]^{1/p} . \tag{8.1}$$

This reduces to the usual Euclidean metric space when $p = 2$, which is the value that I always use. I do not find that this this parameter makes much difference on my problems, but it should be quite harmless to experiment with, as long as values are of order unity. Negative values are allowed, but not recommended, since the geometric meaning is unclear.

`par.general.noisy` is a logical value (true/false, or equivalently 1/0) that tells Ferret whether or not your problem is inherently 'noisy', where a noisy problem is one whose evaluation is stochastic, such that slightly different fitness values are returned each time that it is run on the same parameter set. A good example of a noisy problem is a fitness function that evaluates the goodness of results from a Monte Carlo simulation, like my 'Advanced/EyeVolution' demo, which evolves the lens of a faceted eye by directly simulating the deflection of photons striking random positions on the lens. Here, the error of each evaluation is proportional to the square root of the number of photons, and a different value is obtained every time a given parameter set is evaluated. In cases like this, you should set `par.general.noisy=1`, so that Ferret knows that the fluctuations that it sees might be noise and do not necessarily represent complicated non-linearities in the parameter space.

`par.general.useMex` is a logical value (true/false, or equivalently 1/0) that tells Ferret whether it should use compiled mex files for its most time-consuming internal calculations. You should set this to 'true' unless you have trouble compiling the mex files on your system. Note that they should have been compiled automatically the first time that you launched Qubist, and you would have received an error message and instructions if there was a problem.

8.4 par.strategy

The `par.strategy` section of the setup file includes options that control Ferret's auto-adaptation features, which allow it to model and optimize its own performance on your problem while it runs. Auto-adaptation was outlined briefly in Section 2.2.8 and the features of the auto-adaptation console are discussed more thoroughly in Section 7.1.

```
% Strategy Parameters
par.strategy.isAdaptive=true; % [logical]: Global switch for strategy adaptation.
par.strategy.NGen=10; % [integer > 1]: Number of generations to use for trackers.
%
% Which strategy parameters are allowed to adapt? (true or false)
par.strategy.adapt.mutation.scale=true; % [logical]
par.strategy.adapt.XOver.dispersion=true; % [logical]
par.strategy.adapt.XOver.strength=true; % [logical]
par.strategy.adapt.XOver.ALS=true; % [logical]
par.strategy.adapt.XOver.matingRestriction=true; % [logical]
par.strategy.adapt.niching.acceleration=true; % [logical]
```

`par.strategy.isAdaptive` is a logical value (true/false, or equivalently 1/0) that toggles auto-adaptation on (true/1) or off (false/0). *This parameter overrides the toggles for individual strategy parameters in the* `par.strategy.adapt` *structure discussed below.*

Strategy auto-adaptation builds a model of Ferret's performance based on data gathered while the code runs. However, the model must remain current as the run progresses and encounters different parts of the fitness landscape. Therefore, it needs to weight recent experience more strongly while gradually 'forgetting' generations in the more distant past that are probably no longer relevant. This is modeled as an exponential decay, where `par.strategy.NGen` represents the 'decay time constant' or roughly the number of generations required to forget. I typically use a value of about 10-25 generations, although this is probably not critical.

All of the fields inside `par.strategy.adapt` are logical values (true/false, or equivalently 1/0) that toggle the auto-adaptation of an individual strategy parameter. They are all over-ridden and set to false/0 when `par.strategy.isAdaptive=false`. When `par.strategy.isAdaptive=true`, then the `par.strategy.adapt parameters` can be used to turn individual strategy parameters on or off.

- `par.strategy.adapt.mutation.scale` controls the auto-adaptation of the mutation scale. The mutation scale is set initially to the value in `par.mutation.scale` and is allowed to evolve without any imposed constraints on the upper bound. The lower bound is zero.
- `par.strategy.adapt.XOver.dispersion` controls the auto-adaptation of the dispersion parameter used in crossover. The dispersion is set initially to the value in par.XOver.dispersion and is allowed to evolve without any imposed constraints on the upper bound. The lower bound is zero.
- `par.strategy.adapt.XOver.strength` controls the auto-adaptation of the crossover strength. The crossover strength is set initially to the value in `par.XOver.strength` and is allowed to evolve without any imposed constraints on the upper bound. The lower bound is zero.
- `par.strategy.adapt.XOver.ALS` controls the auto-adaptation of the crossover setting for Advanced Lethal Suppression (ALS). The ALS parameter is set initially to the value in `par.XOver.ALS` and is allowed to evolve within its normal interval of [0, 1]. *This parameter is usually poorly defined by auto-adaptation. It is harmless to allow it to auto-adapt, but the benefit is often minor.*
- `par.strategy.adapt.XOver.matingRestriction` controls the auto-adaptation of the mating restriction parameter. The matingRestriction parameter is set initially to the value in `par.XOver.matingRestriction`, and is constrained to its [-1, 1] interval.
- `par.strategy.adapt.niching.acceleration` controls the auto-adaptation of niche acceleration. This feature allows Ferret to preferentially explore the parameter space in search directions that

coincide with the spatial distribution of the population, as described in Section 8.9.3 below. The acceleration parameter must be in the range of [0,1] and is set initially to the value contained in `par.niching.acceleration`. For most problems, the acceleration parameter will drift to low values in the [0,0.5] range.

I developed the auto-adaptation feature for Ferret-4, and it is therefore a relatively recent addition to the code. However, it has been used quite a lot already and works well. I normally turn auto-adaptation on for all parameters and only turn it off if I suspect a problem based on the appearance of the graphs in the auto-adaptation console (i.e. a strategy parameter heading for a very small or very large value). This happened frequently with early and less reliable versions of the auto-adaptation technique, but I have not needed to do this for any project for quite some time.

8.5 par.parallel

The control parameters in `par.parallel` specify options that control Ferret's parallel computing system, which was the topic of Chapter 6). Parallel computing with Ferret is very easy, and most of the parameters in this section are fairly self explanatory. Note that some can be overridden using the node manager, as discussed below.

```
% Parallel Computing
par.parallel.NWorkers=0; % [integer >= 0]: Number of worker nodes to launch initially.
par.parallel.minChunkSize=10; % [integer]: Minimum number of evaluations in each ↘
        ↪ work chunk.
par.parallel.timeout=15; % [integer >= 0]: Maximum time in seconds before an ↘
        ↪ unresponsive node disconnects.
par.parallel.latency=0; % [real > 0]: Time to pause in seconds after writing to the ↘
        ↪ scratch directory.
par.parallel.useJava=true; % [logical]: Is Java required for worker nodes?
par.parallel.writeLogFiles=true; % [logical]: Are log files required?
```

`par.parallel.NWorkers` is an integer that specifies the number of worker nodes to start automatically when your project starts. Recall that the main MATLAB process running Ferret does not count as a worker node, so that $NWorkers = 3$ corresponds to four MATLAB processes working together on your project. Note that it is not usually necessary to set `par.parallel.NWorkers` in your setup file, unless you *always* want your project to start with the same number of nodes, or you are using a custom launcher for a parallel run that does not start the Ferret console (see Section 4.14.2). Nodes can be started using the node manager interface at any time after initializing the project, as discussed in Chapter 6, which is the preferred method for starting nodes.

`par.parallel.minChunkSize` is the minimum size of an 'evaluation chunk' (the number of parameter sets) that is sent to each worker node for evaluation. Parameter sets are divided between nodes according to the formula

$$\texttt{chunkSize} = \texttt{min}(\texttt{NX}, \texttt{max}(\texttt{par.parallel.minChunkSize}, \texttt{ceil}(\texttt{NX/NNodes/nodeDistributionFactor}))); \tag{8.2}$$

where NX is the number of parameter sets requiring evaluation, $NNodes$ is the number of active nodes including the Ferret console (node-0), and $nodeDistributionFactor$ is a small integer (usually 3-5) that is set inside Ferret. The $minChunkSize$ parameter should always be greater than 1, and preferably at least

10. Low values result in small evaluation chunks, which increase the communication overhead and decrease the efficiency of the code.

`par.parallel.timeout` is the time to wait in seconds for an unresponsive node before disconnecting it and sending a signal to try to shut down the node. In effect, this is the longest duration that Ferret could hang in the unlikely event that a node crashes or becomes unresponsive while holding the lock for the scratch directory used by the parallel computing system. As discussed in Chapter 6, I recommend setting the timeout parameter to at least fifteen seconds to avoid unnecessary node disconnections.

`par.parallel.latency` is the time to pause after a node writes results to the scratch file. This is sometimes useful for networked disks on file systems when multiple computers are contributing to a parallel run, as discussed in Chapter 6. The *latency* parameter should be set to `par.parallel.latency=0` if all of your nodes are running on the same machine. Usually, I set this parameter to 0, unless I start seeing a lot of errors like 'Unknown error calculating fitness function. Re-evaluating N solutions.'. Such errors are actually harmless, but degrade the efficiency of the code. See Section 6.3 for details.

`par.parallel.useJava` is a logical parameter that determines whether Java will be used for nodes that are started automatically when `par.parallel.minChunkSize > 0`. Note that a 'useJava' option is also available in the node manager, which is the normal method for choosing whether or not to use Java. However, once a node has been started, Java cannot be turned on or off without shutting it down and re-starting.

`par.parallel.writeLogFiles` is a logical parameter that determines whether or not log files will be written to disk showing the progress of each node. The log files will be written in the scratch subdirectory within the run's data directory, and will be named 'log.n', where n is the number of the associated worker node. The main Ferret console writes logs to the file 'log.0'. These are simple ascii text files that you can read with the MATLAB editor, or with any text editor.

8.6 par.selection

All genetic algorithms require a selection operator to filter out poor quality solutions in favour of better ones, which eventually dominate the population. Ferret's selection operator uses a binary tournament, and the `par.selection` fields of the setup file control the details of how this operator works.

```
% Selection
par.selection.PCompete=1; % [0 - 1]: Probability that each individual will compete.
par.selection.pressure=1; % [0 - 1]: Selection pressure on overall fitness.
par.selection.BBPressure=1; % [0 - 1]: Selection pressure on BBs.
par.selection.FAbsTol=0; % Absolute fitness range (+/- FAbsTol) to use as a fuzzy ↘
        ↪ fitness band.
par.selection.FRelTol=0; % [0 - 1]: Fraction of fitness range to use as a fuzzy ↘
        ↪ fitness band.
par.selection.FRankTol=0; % [0 - 1]: Fraction of rank range to use as a fuzzy ↘
        ↪ fitness band.
par.selection.shareFitness=0; % [0 - 1]: Fitness sharing.
par.selection.clusterScale=1; % [0 - 1]: Setting clusterScale < 1 promotes clusters ↘
        ↪ on the requested scale.
par.selection.competitionRestriction=0; % [-1:1]: Negative values enhance ↘
        ↪ probability of competitions between nearby individuals.
```

Tournament selection in Ferret is based on a set of prioritized criteria; in case of a tie, Ferret drops to the next most important priority. The selection priorities are as follows:

1. Fitness: Lower values of F are preferred (Recall that Ferret, and all other Qubist optimizers, are minimizers), but ties result when fitness values are equal for single-objective problems, or equal in a Pareto-optimal sense for multi-objective problems. Fuzziness is taken into account when `par.selection.FAbsTol > 0`, `par.selection.FRelTol > 0` and/or `par.selection.FRankTol > 0`, as discussed below.

2. Gene count: Ferret's 'Critical Parameter Detection (CPD)' system is designed such that solutions with fewer specified parameters are preferred over solutions that are more specific. The reason behind this preference is the idea of *parsimony* - that solutions with fewer specified parameters are fundamentally simpler than solutions that are more completely specified. Recall that when a gene is unspecified in Ferret, it is treated as a random number within the current range of that parameter. If random numbers perform as well as a fully specified parameter, then this is an indication that the parameter is unimportant. Solutions with unspecified parameters are more general because they represent an entire class of solutions, where the unspecified parameter can take on any value within the allowed range. Generality increases with the number of unspecified parameters, and hence less specific solutions are regarded as more desirable.

3. Niche count: This is a measure of how many individuals are nearby in the parameter space. Solutions that have fewer near neighbours are in a less explored part of parameter space and are therefore more valuable due to their relative uniqueness. Niching can be done in either the parameter space ('X-niching' in Ferret), objective space ('F-niching'), a combinatorial 'pattern space' ('P-niching'), or any combination of these simultaneously, as discussed in Section 8.9.

Note that if either CPD or niching are turned off, then the corresponding criteria are ignored. The winner of the tournament is chosen randomly if both competing solutions are equally good according to the criteria that are used.

`par.selection.PCompete` is the probability that a given individual will enter a tournament each generation. Normally, I leave this parameter set to 1, which indicates 100% probability. On occasion, I have reduced this value to decrease selection pressure, which places greater emphasis on mutations, crossovers, and especially building block selection (see `par.selection.BBPressure`). This decreases the convergence rate and results in greater emphasis on exploration, which might be useful if you find that you are obtaining very different solutions each time you run your problem. This is sometimes symptomatic of premature convergence causing a problem to get trapped in local minima. If you see this happening, then it is a good idea to decrease the selection pressure to delay convergence until the parameter space has been explored more thoroughly.[2] In such cases, you can turn down the *PCompete* parameter, although it is usually more effective to turn the selection pressure down explicitly by decreasing `par.selection.pressure`.

`par.selection.pressure` offers *explicit* control of the selection pressure. Setting this parameter to a value less than 1 will delay the convergence of your problem, which is often a good thing because the parameter space will be explored more thoroughly. Note that this does not affect the competition probability, but it strongly affects how the winner of each tournament is chosen; the winner will be chosen using the normal priorities discussed above a fraction of time equal to `par.selection.pressure`. The remainder of the time, the actual fitness values will be ignored in the tournament, causing CPD and niching to take top

[2]Note that this problem can also be caused by inadequate parameter space exploration due to a population size that is set too low. See the comments in the discussion of `par.general.popSize` above.

priority. This has a similar effect to decreasing `par.selection.PCompete`, but it is usually more effective to decrease `par.selection.pressure` because it puts greater *explicit* emphasis on niching, which improves the exploration of the parameter space by spreading solutions out as much as possible. My advice is therefore to control selection pressure using `par.selection.pressure` rather than `par.selection.PCompete`.

Ferret contains a second selection operator that works on individual building blocks. This selection operator is processed after building block crossover (see Section 8.8.2 below) and works like a binary tournament between an individual before building block crossover, and an altered form of the same solution after building block crossover. `par.selection.BBPressure` controls the pressure on building block selection in exactly the same way that `par.selection.pressure` controls the pressure of the normal selection operator. I always leave $BBPressure$ set to its default value of 1, which means that fitness always takes priority in a building block selection tournament. This does not tend to cause problems with premature convergence because building block selection involves tournament competitions between solutions that are identical except for a single building block (or a small number of building blocks if `par.link.PMultiLink > 0`), which typically involves only a few parameters rather than the entire parameter set. Therefore building block tournaments normally make less drastic changes to solutions than regular tournaments. Building block crossover and selection are really the heart of Ferret's linkage learning system, and turning down the building block selection pressure partially disables the linkage-learning system. This could result in poor parameter space exploration and solutions getting trapped in local minima. I've left `par.selection.BBPressure` as a control parameter that you can modify, but I don't really recommend decreasing its value below 1.

`par.selection.FAbsTol`, `par.selection.FRelTol` and `par.selection.FRankTol` are tolerances that can be used to add fuzziness to the fitness function. $FAbsTol$ is an absolute tolerance, $FRelTol$ is a relative tolerance, and $FRankTol$ is a tolerance in rank. When a tolerance is set, differences in fitness values less than the tolerance are ignored when determining the winner of a tournament, and the tournament is decided based on gene count and niche count, according to the tournament priorities discussed above. The effect is that solutions spread out within the region allowed by the tolerance. This technique is especially useful to understand the errors in fitting parameters in data-modeling problems. I gave an example of this in the binary star modeling example in Section 5.1, where I set $FAbsTol = 1$ to map out the set of solutions within an absolute tolerance $FAbsTol = 1$ of the $\chi^2_{reduced}$ minimum. Of the three tolerance parameters, $FAbsTol$ is by far the most useful one, $FRelTol$ is sometimes useful, and $FRankTol$ is hardly used at all. $FRelTol$ is a relative tolerance between 0 and 1, defined as a fraction of the range of each fitness value currently in the population. This is sometimes useful for understanding the mathematical degeneracies of a modeling problem when the errors used to calculate χ^2 are not well known. A small non-zero value of $FRelTol$ can also be used to add some thickness to Pareto fronts in multi-objective problems, which helps to fill them in and better elucidate their structure. $FRankTol$ is a fraction of the range of ranks that currently exist in the population. I don't actually find this very useful in practice, but I've included it as an option for completeness.

Fitness sharing is a technique for spreading out solutions over the optimal set (or Pareto front) that is an alternative to niching. In fitness sharing, the rank of a solution is directly modified to take into account the niche count. `par.selection.shareFitness` is a real number between 0 and 1 that determines how much the ranks are modified. Fitness sharing is turned off when $shareFitness = 0$, and is maximal when $shareFitness = 1$. Fitness sharing can operate simultaneously with niching, although I see no point in doing this. Overall, niching seems to be a better technique than fitness sharing for most problems, but fitness sharing is a well known technique and I have therefore included this option in the code. The only time I use it is when I am not happy with the uniformity of the points in an optimal set and I want to try

a different technique for diagnostic purposes.

Solution sets occasionally form distinct clusters or 'islands' in the parameter space. In such cases, it is useful to restrict tournament selection and crossovers so that each individual interacts mainly with members of its own cluster. This serves to promote the formation of clusters and preserve them by limiting their interactions with each other, which could otherwise destroy one cluster in favour of another. It also improves the efficiency of the search on such problems, because fewer lethals are generated as a result of crossovers between distant clusters. The `par.selection.clusterScale` control parameter determines the maximum distance scale for tournament selection and crossover, and hence the scale of clustering. This parameter is set to 1 by default, which indicates that clustering is *not* enhanced because the clustering scale is the same size as the maximum normalized distance possible ($d = 1$) in the normalized parameter space `world.pop{`p`}.indiv.genome.X`, where p is the population number (see Section 4.9.1). Smaller values of *clusterScale* decrease the maximum distance scale of interactions and therefore enhance clustering. The distance scale of interactions is equal to the niche radius when `par.selection.clusterScale=par.niching.X`.

`par.selection.competitionRestriction` is a real number ranging from -1 to 1 that affects how competitors are chosen in a tournament. When $competitionRestriction = -1$, members of the population always choose the nearest available member of the population to compete. The opposite is true when $competitionRestriction = 1$, where individuals always choose the most distant available member of the population. Competition restriction is turned off when the default setting $competitionRestriction = 0$ is used, and all intermediate values are possible. Note that this works much like Ferret's mating restriction parameter, which is discussed in Section 8.8. Negative values might be useful occasionally for the *competitionRestriction* parameter (or at least not harmful), but positive values are probably never useful. I rarely use this option.

8.7 par.mutation

Mutation is one of the most important operators of a genetic algorithm, since it is this operator that allows the code to explore new regions of parameter space by means of random perturbations.

Ferret's mutation operator is quite different from the simple 'bit flip' that a traditional genetic algorithm would use. This is because Ferret does not use binary bit strings, but represents parameters as real numbers, much like an 'evolution strategies' code. This leads to a richer set of possibilities for evolutionary operators than is possible for a traditional genetic algorithm, because real vector spaces admit notions of geometry that would be a conceptual stretch for bit strings. This difference is particularly important for the mutation and crossover operators, which are the topics of this section and Section 8.8.

Ferret includes three mutation-like operators: normal mutation, a second operator called 'supermutation', and building block mutations.

```
% Mutation
par.mutation.PMutate=0.05; % [0 - 1]: Probability of normal mutation.
par.mutation.BBRestricted=false; % [logical]: Restrict mutation to building blocks?
par.mutation.PRandomizeBBs=0.01; % [0 - 1]: Probability of randomizing whole ↘
        ↪ building blocks.
par.mutation.PSuperMutate=0.01; % [0 - 1]: Probability of superMutation.
par.mutation.scale=0.25; % [0 - 1]: Min & max scale of mutation.
par.mutation.selectiveMutation=0; % [-1:1]: Negative values mutate highly clustered ↘
        ↪ individuals selectively.
```

Mutations in Ferret are applied to the normalized parameter space `world.pop{`p`}.indiv.genome.X`, in which all parameters are scaled to the interval ranging from 0 to 1, and p is the population number. Mutations are implemented as Gaussian random perturbations in a random direction, with a distance whose standard deviation is equal to `par.mutation.scale` in this scaled parameter space. `par.mutation.PMutate` is the probability of such a mutation occurring to a given individual in a given generation.

Mutations are restricted to a subset of parameters defined by a single, randomly chosen building block (discovered by linkage learning) when `par.mutation.BBRestricted=true`. In my experience, this only appears to be useful for large, non-parametric problems, with possibly hundreds of parameters (see Section 4.20 or the demo 'Advanced/Non-Parametric-Image-Modeling'). The default setting `par.mutation.BBRestricted=false` is the best setting for most other problems, since it allows Ferret to decide automatically when to restrict mutations to individual building blocks or do unrestricted mutations over all parameters.

`par.mutation.selectiveMutation` is a parameter that can be used to influence the formation of clusters in the parameter space. When this parameter is negative, mutations are applied most frequently to individuals residing in highly clustered regions of parameter space, which have a high 'niche count', as defined by equation 8.3. This tends to break up clusters and often results in a more uniformly filled optimal set. The opposite is true when this parameter is positive. In this case, mutations are applied most often to individuals with low niche count, and clusters are left relatively undisturbed to develop and converge. The $selectiveMutation$ parameter must range from -1 to 1, and is turned off when $selectiveMutation = 0$, much like the `par.selection.competitionRestriction` parameter discussed in Section 8.6. In my experience, negative values are useful for many problems, but positive values usually have a negative effect on the quality of solutions.

Ferret's second mutation operator is called 'supermutation' and is controlled by a single parameter `par.mutation.PSuperMutate`. Supermutation is an idea that arose from my observation that well-converged runs would sometimes enjoy a sudden improvement after resuming them following a computer crash, using the resume features discussed in Sections 4.17 and 12.1. This was puzzling initially, but the explanation is reasonable and quite interesting. By default, History files only contain optimal solutions and elites, so there is not enough state information saved to resume a run *exactly* where it was before a crash. All non-optimal and non-elite solutions are randomized, but this is OK because the optimals and elites dominate the population within a few generations anyway. The observation of an improvement indicates that occasional population-wide randomization events can be *beneficial* to the progress of the run. I cannot resist likening this effect to the explosions of diversity and rapid evolution observed in Earth's fossil record following cataclysmic global extinction events. The analogy is not perfect, but the general principle is the same; an extinction event that wipes out many dominant species leaves the door open for evolution to experiment with new forms. Diversity blossoms throughout the ecosystem, and this injection of new ideas sometimes results in overall improvement.

Ferret's 'supermutation' operator is triggered with a probability equal to `par.mutation.PSuperMutate`. When a supermutation 'cataclysm' occurs, a single building block is perturbed for all non-optimal and non-elite solutions, in all populations. The perturbation occurs in the scaled parameter space discussed above, and has a magnitude drawn from a Gaussian random distribution whose standard deviation is equal to `par.mutation.scale`. Supermutation is a very disruptive event, and clearly you do not want this to happen very often, so the $PSuperMutate$ probability should be low: $PSuperMutate = 0.01$ is a good, safe value, and values up to about 5-10% are probably OK.

`par.mutation.PRandomizeBBs` is a probability that controls Ferret's final mutation-like operator, which selects individuals not currently in the optimal set, with probability $PRandomizeBBs$, and randomizes a single building block within each member of this group. $PRandomizeBBs$ should be no greater than a few percent. This building block mutation operator is not very disruptive because relatively few individuals are affected each generation, and only a single building block is affected for each solution. This is a new feature at the time of this writing, and the value of doing this is not quite certain, but it is unlikely to hurt the search, as long as $PRandomizeBBs$ is small. However, it might help the search by injecting new diversity into the population.

Note that the mutation scale `par.mutation.scale` is one of the parameters controlled by Ferret's auto-adaptation system. When this parameter is allowed to auto-adapt, Ferret will try to optimize it according to a mathematical model for the diffusion of new mutations throughout a population. This works well in practice, and I normally allow this parameter to auto-adapt in my own runs.

8.8 par.XOver and par.XOverBB

Crossover combines two individuals by mixing their parameters, which results in a new individual that shares some traits with each parent. This is a straightforward operation in a standard binary genetic algorithm, where strings of zeros and ones are simply cut, recombined, and spliced back together. However, Ferret's parameter representation admits a more elaborate crossover operator because it is based on real vector spaces, and vectors naturally lead to notions of geometry. Ferret's most important crossover operator, which I refer to as 'X-Type' crossover, is a geometry-based operator that is in some ways analogous to the bit string operator found in traditional binary encoded GAs. X-Type crossover is discussed in Section 8.8.1.

Ferret also has a second type of crossover operator, which is at the heart of its linkage-learning system, and has no analogy in traditional genetic algorithms. I refer to this second type of crossover as 'building block crossover', and it is discussed in Section 8.8.2.

8.8.1 par.XOver

X-Type crossover is essentially an averaging operation, which draws a line between the parameter space coordinates of two individuals and selects a point between the individuals on that line. There are some useful modifications to this process, which I discuss below, but that is the basic geometric idea. The options for Ferret's 'X-Type' crossover are as follows:

```
par.XOver.PXOver=1;  % [0 - 1]: Probability of X-Type crossovers.
par.XOver.BBRestricted=false; % [logical]: Restrict crossover to building blocks?
par.XOver.strength=0.25; % [0 - 1]: Scale of crossover.
par.XOver.maxScale=1.1; % [~1, or slightly larger]: maximum XOver perturbation allowed.
par.XOver.antiXOverProb=0; % [0 - 1]: Probability of anti-crossovers.
par.XOver.ALS=0; % [0 - 1]: ALS: Advanced Lethal Suppression.
% ----------------------------------------
% Which dispersion technique should be used?
par.XOver.dispersionTechnique='cylindrical'; % [string]: Hollow cylinder.
% par.XOver.dispersionTechnique='conic'; % [string]: Scaled hollow cone.
% par.XOver.dispersionTechnique='biconic'; % [string]: Scaled hollow double-cone.
% ----------------------------------------
par.XOver.dispersion=0.25; % [real < 1]: Determines major axis standard deviation ↘
        ↪ of dispersion structure.
par.XOver.matingRestriction=0; % [-1:1]: Negative values enhance probability of ↘
        ↪ matings between relatively nearby individuals.  Positive values are ↘
        ↪ probably not useful.
```

`par.XOver.PXOver` is the probability that an individual will engage in crossover with another individual in any given generation. Crossover is crucial for mixing the parameter space and propagating new solutions throughout the population. I recommend setting `par.XOver.PXover=1` for most problems.

Crossovers are restricted to a subset of parameters defined by a single, randomly chosen building block (discovered by linkage learning) when `par.XOver.BBRestricted =true`. In my experience, this only appears to be useful for large, non-parametric problems, with possibly hundreds of parameters (see Section 4.20 or the demo 'Advanced/Non-Parametric-Image-Modeling'). The default setting `par.XOver.BBRestricted =false` is the best setting for most other problems, since it allows Ferret to decide automatically when to restrict crossovers to individual building blocks or do unrestricted crossovers over all parameters.

`par.XOver.strength` influences the magnitude of the perturbations caused by crossover events. Normally, the magnitude is given by the absolute value of a Gaussian random variable, whose standard deviation is equal to `par.XOver.strength`. The crossover is scaled such that the newly recombined solution is halfway between the parents when a magnitude of 0.5 is drawn from the Gaussian distribution. A magnitude of 0 corresponds to no crossover at all, while a magnitude of 1 moves all the way over to the opposite parent to form the child solution, which also means that there is effectively no crossover.

A Gaussian random variable has no upper limit, but a crossover magnitude greater than 1 overshoots the opposite parent. Therefore, the Gaussian distribution of crossover magnitudes is truncated at a value equal to `par.XOver.maxScale`. When a magnitude is drawn that is larger than this value, it is replaced with a uniform random number between 0 and `par.XOver.maxScale`. It is actually beneficial to set this value slightly higher than 1 because an occasional *slight* overshoot helps to fill out linear features in the optimal set. This subtlety works much like the 'anti-crossover' operator discussed below.

`par.XOver.antiXOverProb` is the probability of an 'anti-crossover' event. Basically, anti-crossover just reverses the direction of a normal crossover event. Instead of moving a point closer to its mate, it moves further away. I would be the first person to agree that this is a counter-intuitive idea, but I have observed many times that this usually leads to better mapping of parameter space, especially when there are long linear features present in the optimal set. This option has been available since Ferret-1 and I used it on an

astrophysical problem as early as 2002 (published 2004; see Fiege et al. [2004]). While useful, this should not happen too often. The antiXOverProb probability is set to zero by default, and probably should not exceed about 10%.

Lethals are poor solutions that result from the crossover of two good parent solutions. Ferret contains an 'Advanced Lethal Suppression' (ALS) algorithm, which is designed to decrease the number of lethals by avoiding poor regions of parameter space during crossover. ALS is controlled by `par.XOver.ALS`, which should be a real number between 0 (ALS off) and 1 (maximum ALS). The ALS system works quite well, and the effect can be observed by turning it on and off in the Ring/Ring-2D demonstration found in the Demo menu. Note that ALS is one of the parameters that is controlled by Ferret's strategy auto-adaptation system. I usually leave this on, but it rarely chooses a well-defined value for this parameter. I would be inclined to turn ALS on manually for problems if I suspect that it is really necessary.

Ferret's crossover operator includes a 'dispersion' step, which plays an important role in helping solutions to expand throughout the optimal set and fill it out uniformly. Dispersion is essentially an extra mutation step in which the magnitude of the mutation scales with the magnitude of the crossover event. Without dispersion, crossovers follow a line joining the two parameter sets that are involved. However, when dispersion is included, this line is broadened to fill a multi-dimensional region centred on the line, which is why crossover plus dispersion helps to fill the optimal set more completely than crossover alone. Ferret contains three distinct dispersion techniques.

- cylindrical: A perturbation is selected with a magnitude drawn from a Gaussian random distribution with standard deviation equal to $par.XOver.dispersion \times \Delta\mathbf{x}$, where $\Delta\mathbf{x}$ is the distance between the two points involved in the crossover. This is added to the crossover point. A large number of crossovers and dispersions would map out a multi-dimensional cylinder with rounded ends centred on the two parents involved in the crossover.
- conic: A perturbation is selected with a magnitude drawn from a Gaussian random distribution with standard deviation equal to $par.XOver.dispersion \times \Delta\mathbf{x}$, where $\Delta\mathbf{x}$ is the crossover magnitude, as defined above. This is added to the crossover point. A large number of crossovers and dispersions would map out a multi-dimensional cone with an apex on one of the two parents, and a rounded end centred on the other parent.
- biconic: identical to the conic option, except that the region mapped is biconic, with an apex on both parent solutions.

`par.XOver.matingRestriction` is an option that affects how mates are chosen prior to crossover. It works much like the `par.selection.competitionRestriction` parameter, but is much more useful. The *matingRestriction* parameter can range from -1 to 1 and is set to 0 by default. Individuals engage in crossover preferentially with individuals that are nearby in parameter space when $matingRestriction < 0$, and with individuals that are distant when $matingRestriction > 0$. Mating restriction is turned off when this parameter is set to zero. Negative values are often beneficial because crossovers between two nearby good solutions are likely to result in another high quality solution, while crossovers between more distant individuals are more likely to produce a poor solution. Negative values have a similar effect to ALS, and also to setting `par.selection.clusterScale > 0`; all of these techniques can be used simultaneously to suppress lethals. I usually set the *matingRestriction* parameter to about -0.5 initially for my problems. Negative values are often useful and I have never observed a case where a moderate negative mating restriction was actually harmful. Positive values are usually damaging because of the increased probability of lethals, and are not recommended.

Note that `par.XOver.matingRestriction` is one of the parameters that can be controlled by Ferret's auto-adaptation system, although it is usually not very tightly controlled. Nevertheless, auto-adaptation will usually prefer values centred on zero, or negative values for this parameter, which is consistent with the comments above.

8.8.2 par.XOverBB

Building block crossover involves swapping an entire building block in its entirety between two individuals. This very simple operation is fundamental to the improvements that result from Ferret's linkage learning system. The building block crossover operator is solely responsible for how building blocks are recombined with other building blocks, and is therefore an essential part of Ferret's linkage-learning system. The operator has no effect when linkage-learning is turned off (`par.link.PLink=0`).

```
% Building Block Crossover
par.XOverBB.PXOver=1; % [0 - 1]: Probability of Building Block crossovers.
par.XOverBB.multiBB=0; % [0 or 1]: Single or multiple BB XOver?
par.XOverBB.NPass=1; % [integer >= 0]: Number of passes through BB selection per cycle.
```

`par.XOverBB.PXOver` is a real number between 0 and 1 that represents the probability that an individual will engage in building-block crossover at a given generation. Normally, the thorough mixing of building blocks is a good thing, and I leave this parameter set equal to its default value of 1 to maximize the effects of linkage-learning. I have never seen a case where it was clear that a lower setting was beneficial.

Ferret actually contains two different (and mutually exclusive) methods for recombining and performing selection on building blocks. By default, `par.XOverBB.multiBB=0` (false) so that building block crossover and selection operate only on a single building block for each mating pair of solutions. However, it is possible to process multiple building blocks simultaneously by setting `par.XOverBB.multiBB=1` (true). This may result in faster convergence of the linkage map, and therefore faster convergence of the population to the optimal set. Both options work well, although I regard single building block crossover to be somewhat more conservative, and this is my usual choice.

The mixing and selection of building blocks is so important in Ferret that I sometimes like to go through several passes of these operations before proceeding to other operations in the generational cycle. Therefore, I have included an option `par.XOverBB.NPass`, which is an integer that controls the number of building block crossover/selection passes per generation. This is set equal to 1 by default, but I have seen problems where a higher setting can result in a faster and more reliable solution due to greater mixing of building blocks. Note, however, that a greater number of solutions will need to be processed each generation, and generations will take longer. More processing will be done each generation though, so Ferret's performance as a whole might not be degraded.

8.9 par.niching

Ferret includes a sophisticated niching mechanism to help distribute solutions evenly over the optimal region, or trade-off surface for multi-objective problems. This section explains the types of niching options available in Ferret.

```
% Niching
par.niching.priority='PXF'; % [string: re-order 3 letters, 'P', 'X', or 'F']: ↘
        ↪ Priority of niching. If empty, use Pareto niching.
par.niching.P=0;  % [0 - 1]: Pattern niching: Typically ~0.5 is about right.
par.niching.X=0;  % [0 - 1]: X-Niching: Typically ~0.25 is about right.
par.niching.XPar=[]; % [integer vector > 1]: List of parameters used for X-niching. ↘
        ↪ If empty, use all.
par.niching.F=0;  % [0 - 1]: F-Niching: Typically ~0.25 is about right.
par.niching.method='sigmaShare'; % [string: 'sigmaShare' or 'powerLaw']: ↘
        ↪ Specify a niching method.
par.niching.exponent=2; % [real: usually > 0 & <~ 2]: Used in the niche function.
par.niching.powers=[4]; % [real vector: usually > 0]: Used only for power-law ↘
        ↪ niching - not sigmaShare.
par.niching.acceleration=0.5; % [0 - 1]: Acceleration parameter.
%
% ----------------------------------
% Pattern Niching Specifics
par.niching.patternMethod='sigmaShare'; % [string: 'sigmaShare' or 'powerLaw']: ↘
        ↪ Method to use for pattern niching.
par.niching.patternExponent=2; % [real: usually > 0 & <~ 2]: ↘
        ↪ exponent for pattern niching.
par.niching.patternPowers=[2]; % [real vector, usually > 0]: ↘
        ↪ Used only for power-law niching.
```

Niching can be done in parameter space ('X-niching'), objective space ('F-niching'), or 'pattern (combinatorial) space' ('P-niching'). The first two options have been discussed in the literature (see Fonseca & Fleming [1993] for example) and are by far the most commonly used, so I will address these first. Niching in X (`par.niching.X > 0`) and F (`par.niching.F > 0`) differ only by whether niching is done over the set of parameters or over the fitness values determined by your objective function. Obviously, you should normally choose to niche in X for single-objective problems; although niching in F is not technically an error, Ferret will try to spread out the fitness values evenly over the optimal region, which may just be a point for non-fuzzy problems with a single objective. If a single-objective fitness function is fuzzy, then F-niching would try to spread out the population over all possible fitness values that are within the fuzzy tolerance of the minimum value. Even so, this probably won't do a very good job of mapping the parameter space, and it is probably not what you want to do. Niching in X makes much more sense for single-objective problems - even fuzzy ones.

`par.niching.XPar` is a vector that may contain a subset of parameters to be used for X-niching. For example, setting `par.niching.XPar=[1:5,7]` causes parameters 1 through 5, as well as parameter 7 to be used when calculating the distance metric used for niching, while all other parameters are ignored. `par.niching.XPar` is empty by default, which indicates that all parameters should be used for niching.

For multi-objective problems, niching in either X or F is equally valid and the best choice depends on the goals of the problem. If you are interested mainly in the trade-off surface and want to show well-populated graphs of one fitness function versus another, then you probably want to niche in F. On the other hand, if you are more interested in mapping out the parameters of your optimal set, then you should probably choose to niche in X. Ferret will emphasize whichever option that you choose as your priority, at the expense of the other. For most of my problems, I choose to niche in X because my problems usually involve modeling data, and this choice is best for mapping the range of model parameters allowed by the

data. I have also occasionally done the same run twice using X-niching in one run and F-niching in the other, and then combined the results using Qubist's 'Merge OptimalSolutions' tool. This approach offers the best of both worlds.

Niching comes into play when solutions are equal in a Pareto-optimal sense, or within some fuzzy tolerance, or when `par.selection.pressure < 1`. Niching is simply a strategy that prefers solutions with fewer near neighbours over solutions with a greater number of neighbours. The logic behind this preference is simple: solutions in a less populated region of parameter space are more unique, and therefore more valuable to the exploration of the space. The problem of niching therefore reduces to the final round of tournament selection - after fitness and CPD selection (see Section 8.6) - that prefers solutions with a low 'niche count' (see Section 8.6), which (roughly speaking) determines the number of near neighbours that a solution has within a certain 'niche radius'.

Pattern niching ('P-niching') is an unusual type of niching that strives to produce the greatest possible variety in the *combinations* of specified and unspecified parameters. I designed this a number of years ago when I was building a simple demonstration that would download a few hundred stock histories from the internet and combine them to construct optimally-performing portfolios that simultaneously maximize historical performance, while minimizing risk. Any portfolio managers reading this should relax; this was intended only as a simple demonstration and not as a serious attempt at portfolio optimization. I am very well-aware that optimizing portfolios based on historical data alone is a dangerously simplistic approach, but even this over-simplified problem is interesting from the perspective of an optimization problem. I decided that I wanted to allow portfolios that did not always contain the full set of stocks, and moreover I wanted to find the most diverse set of Pareto-optimal portfolios possible. I set up this problem so that Ferret's CPD system would send NaN values instead of random numbers to the fitness function if a parameter was not defined (`par.CPD.NaN2Random=0`), and these NaN values would be interpreted as 'this stock is not present'. I designed pattern niching to prefer unique *combinations* of stocks based on the patterns of NaN values present in the portfolio representation, which very effectively spread out the solutions in 'pattern space' to find a maximally diverse set of portfolio options. I have not really used this option for any other problem, but there may be other problems that I have not considered that could benefit from this approach.

Two solutions are considered to be near neighbours if they reside within the 'niche radius' in X, F or P space; solutions that are outside do not contribute to the niche count. The X and F niche radii are determined by `par.niching.X` and `par.niching.F` respectively, which reflect normalized distances in the parameter and objective spaces. Therefore, they are real numbers that are normalized to be in the range of [0,1]. Likewise, pattern niching works in a normalized pattern space, and `par.niching.P` is also normalized to the range [0,1]. These strategy parameters are used to scale the niche radius dynamically such that all other solutions in the population reside within the niche radius when the parameter is set equal to 1, and only a few other solutions are within the niche radius when the parameter is small. The normalization ensures that you never need to worry about the actual range of your parameters or fitness values. The same niche radius will work even as the population zeros in on the optimal set or zooms in (see Section 8.14), and the range of values represented in in the population decreases.

Ferret's niching system evolved over an extended period of time over all four generations of the code; the system is robust, and will just work with any 'sane' value for the niching radius. Niching is skipped for niche radii that are set equal to 0. Normally, I set `par.niching.X` or `par.niching.F` to somewhere in the range [0.2,0.35], and I do not find that runs are particularly sensitive to this value. It is often useful to set `par.niching.P` slightly higher, and I would recommend a value of about 0.5 if you are using pattern niching.

I normally only turn on one type of niching for any given run, but you can turn on more than one type if you wish. In this case, `par.niching.priority` sets the priority for the type of niching. `par.niching.priority` is a string of up to three characters comprised of the letters 'P', 'X' and 'F', which specifies the priority of niching if more than one type of niching is defined. Ferret will do niche selection in the order specified, with 'ties' proceeding to the next round of niche selection. For example, `par.niching.priority='PXF'` will cause Ferret to niche in pattern space. Ties, which are likely to be common for pattern niching, will proceed to niching in parameter space ('X-niching'), and any remaining ties will compete based on their niche count in objective space ('F-niching'). If `niching.priority` is set to an empty string, then Ferret will try to minimize all three niche counts simultaneously, in a Pareto-optimal sense, without preferring one over the other. Note, however, that the corresponding type of niching is skipped when `par.niching.X=0`, `par.niching.F=0`, or `par.niching.P=0`. Generally, it it better to use Pareto niching (`niching.priority=''`) if you have enabled both X-niching and F-niching, because niche count ties are unlikely in X and F.

You can select between two different types of niching by setting `par.niching.method` to either 'sigmaShare' or 'powerLaw'. The sigmaShare option is the default, and the better choice in my opinion. I have spent considerably more time and effort testing the sigmaShare option, and I have not yet seen a case where powerLaw niching is demonstrably superior.

8.9.1 'SigmaShare' Niching

'SigmaShare' niching is the standard method for niching in Ferret, which is selected by specifying `par.niching.method='sigmaShare'`. A niche radius σ_{share} is chosen, which is equal to either `par.niching.X` or `par.niching.F`, and solution i is assigned a niche count ϕ_i based on how many other solutions are within the niche radius:

$$\phi_i = \sum_{j:j\neq i} \left(1 - min\left\{1, \frac{d_{i,j}}{\sigma_{share}}\right\}\right)^p, \tag{8.3}$$

where the exponent $p > 0$ is given by `par.niching.exponent`. Basically, this equation says that the niche count for solution i is obtained by summing the expression over all other solutions j where the normalized distance $d_{i,j}$ between solutions i and j is less than the niche radius σ_{share}, giving more weight to nearby solutions than those near the boundary of the niche radius. Ferret attempts to minimize ϕ_i as one of the tournament selection criteria discussed in Section 8.6. The exponent `p=par.niching.exponent` controls the 'niche shape', as illustrated in Figure 8.1. This parameter does not seem to be very critical and I usually use a value in the range of 1 to 2. Higher values cause extra weight to be given to very close solutions, while solutions at a distance close to σ_{share} contribute much less to the niche count. Lower values result in more even weighting of solutions within the niche radius. In the limiting case of `par.niching.exponent=0`, the niche count is a simple, unweighted count of the solutions within the niche radius.

8.9.2 'PowerLaw' Niching

Power law niching is an alternative to the sigmaShare approach discussed above, which is enabled by choosing `par.niching.method='powerLaw'`. The overall idea is similar, but power law niching puts a 'soft edge' on the niching radius. The contribution to the niche count ϕ_i is exactly zero for all solutions outside the niche radius when sigmaShare niching is enabled, but the contribution falls off gradually as a power law in radius when the powerLaw option is chosen; all solutions will contribute to the niche count, but ϕ_i is dominated by solutions that are nearby.

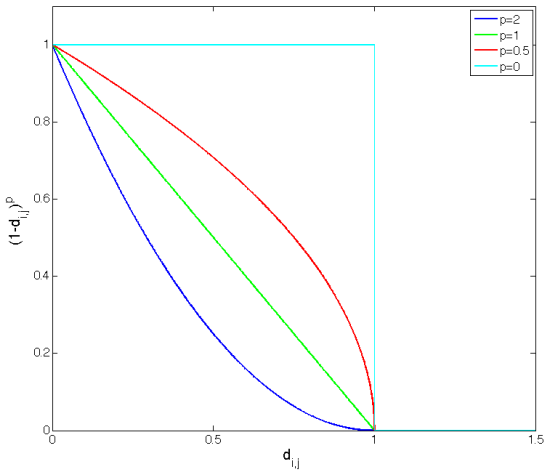

Figure 8.1: Illustration of niche shapes, as used in equation 8.3.

When power law niching is selected, Ferret's niching mechanism constructs a niche count for each individual based on power laws of the following form:

$$\phi_i = \sum_{n=1}^{N} \sum_{j:j\neq i} \left(1 + \frac{d_{i,j}}{\sigma_{share}}\right)^{-p_n}, \tag{8.4}$$

where p_n is the nth power (normally positive) in the list of N powers contained in real vector `par.niching.powers`, and σ_{share} (equal to `par.niching.X` or `par.niching.F`) is used as a *core distance* to soften the power law. Softening is required because equation 8.4 would be singular where $d_{i,j} = 0$ if we were to leave out this term and use a simple power of $d_{i,j}$. By choosing appropriate positive powers, the user is able to determine how rapidly contributions to the niche count fall off with distance. In most cases, a single power is used, but I have included the option to allow multiple powers for the purpose of experimentation. The contribution to the niche count is shown for several values of p_n in Figure 8.2. Note that `par.niching.powers` and `par.niching.core` have no effect when sigmaShare niching is selected.

8.9.3 Acceleration

When `par.niching.acceleration > 0`, Ferret examines the spatial distribution of solutions within the niche radius σ_{share}, and uses this information to influence the distribution of new points chosen for mutation or crossover dispersion. The idea is that the local distribution of solutions reflects search directions that are known to be productive. For example, if the population has settled into an optimal region delineated by a long, slender valley, then it makes sense to expend greater effort searching for new solutions that follow the valley, rather than using an isotropic distribution, where most points would be poor quality solutions that land on the valley walls well above the minimum. Ferret uses no acceleration when `par.niching.acceleration = 0`, and full acceleration when `par.niching.acceleration = 1`. In my experience, full acceleration is too much for most problems because it limits the search directions too strongly. I usually use an initial value of about 0.5.

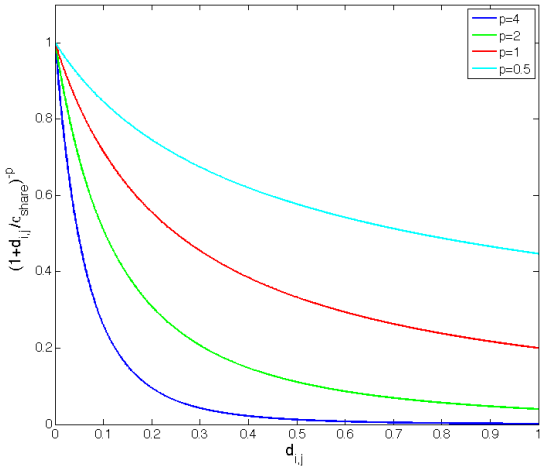

Figure 8.2: The effect of varying the power law p_n in powerLaw niching. A value of $\sigma_{share} = 0.25$ was chosen. Refer to equation 8.4 for details.

Note that the acceleration can be controlled by Ferret's auto-adaptation system, as discussed in Section 8.4. My advice is to set this parameter to about 0.5 and turn on the auto-adaptation: `par.strategy.isAdaptive=true`; `par.strategy.adapt.niching.acceleration=true`. You will probably find that auto-adaptation prefers lower values in the range of [0,0.5] for most problems.

Acceleration requires most of the machinery necessary to do niching in X, specifically a distance metric computed over each population. For most problems, it is usually best to turn on niching in X (`par.niching.X > 0`*) when acceleration is enabled to take advantage of this information, which is calculated for acceleration anyway.*

There is one type of problem where it is *critical* to turn off niching acceleration. This is necessary for very large-scale problems with hundreds of parameters, which are usually non-parametric problems as discussed in Section 4.20. Niching acceleration requires mathematical operations that do not scale well to such large problems, and it may even dominate the total run time of your code. In such cases, niching acceleration is simply too expensive, and should be turned off. Note that you must also turn off strategy auto-adaptation for this parameter: `par.strategy.adapt.niching.acceleration=false`.

8.9.4 Pattern Niching

The parameters that control pattern niching are `par.niching.patternMethod`, `par.niching.patternExponent`, and `par.niching.patternPowers`. These work exactly like the corresponding parameters for the more common types of niching (X-niching and F-niching).

8.10 par.CPD

The par.CPD options control Ferret's critical parameter detection (CPD) algorithm. The purpose of the CPD system is to 'weed out' model parameters that have little effect on the quality of solutions, which effectively reduces the dimensionality of the problem. This works by occasionally 'deactivating' a gene, which is signaled by setting its internal value to NaN in world.pop{p}.indiv.genome.X. Note that world.pop{p}.indiv.genome.XPhys contains the *actual* parameter values that are sent to the fitness function, usually with NaN values replaced by real-valued parameter values, as discussed below. Please refer to Section 4.9.1 for a discussion of these fields.

```
% Critical Parameter Detection
par.CPD.PDeactivateGenes=0; % [0 - 1]: Probability of gene deactivation.
par.CPD.PReactivateGenes=0; % [0 - 1]: Probability of gene reactivation.
par.CPD.useTemplates=false; % [logical]: Use templates to replace NaN's in ↘
        ↪ evaluation when possible
par.CPD.NaN2Random=true; %  [logical, usually true]: Replace undefined genes (NaN's) ↘
        ↪ with random numbers?
```

par.CPD.PDeactivateGenes is the probability that a gene will be deactivated each generation. Thus, we would expect approximately par.CPD.PDeactivateGenes $\times$ $NGenes \times N$ deactivations in total each generation, for a problem with $NGenes$ parameters and a population size of N. Clearly, par.CPD.PDeactivateGenes should be very small. I usually use a value on the order of 0.001, although the exact value makes little difference for the reasons discussed below. *Note that the entire CPD system is deactivated when* par.CPD.PDeactivateGenes=0. There is also a reciprocal 'reactivation' operator, which converts NaN values (deactivated genes) to a random value within the parameter range. Reactivation occurs with probability par.CPD.PReactivateGenes, which should be similar in size to the deactivation probability.

The par.CPD.NaN2Random and par.CPD.useTemplates parameters determine whether or not and how underspecified solutions containing NaN values are converted to fully specified parameter sets prior to evaluation by the user's fitness function. When $NaN2Random$ evaluates to *true*, NaN values are converted to a random number within the current parameter range before they are sent to the fitness function. When $useTemplates$ is *true*, Ferret replaces NaN values in underspecified solutions with the corresponding genes from a template solution, which is chosen randomly from the set of fittest individuals in the population. In both cases, it is important to realize that NaN values are maintained internally by world.pop{p}.indiv.genome.X, so the degree to which a given solution is specified internally remains unchanged. Note that the $NaN2Random$ setting overrides the $useTemplate$ setting when both are *true*; in this case, Ferret ignores the $useTemplates$ directive and replaces NaN values with random numbers prior to evaluation of the fitness function. When $useTemplates$ and $NaN2Random$ are both set to $false$, NaN values are sent directly to the fitness function, and it is up to the user to determine what this means and handle their evaluation. I recommend using the $NaN2Random$ setting for most problems, and I hardly ever use templates. Table 8.1 shows the effect of the different combinations of $NaN2Random$ and $useTemplates$ that are possible.

The role of the par.CPD.DeactivateGenes probability is to seed occasional NaN values into the population. However, whether a given gene becomes unspecified or not throughout the population is determined almost entirely by how well these seeds proliferate via Ferret's selection operator - the amount of initial seeding barely matters at all. It is for this reason that the values of $PDeactivateGenes$ and $PReactivateGenes$ are not critical, as long as they are small.

NaN2Random	useTemplate	received by fitness function
false	false	NaN
false	true	template values
true	false	random values
true	true	random values

Table 8.1: Conversion of NaN values in `world.pop{`p`}.indiv.genome.X` when `world.pop{`p`}.indiv.genome.XFullySpecified` and `world.pop{`p`}.indiv.genome.XPhys` are generated prior to evaluation of the fitness function for population p. NaN values are used as placeholders by Ferret's Critical Parameter Detection (CPD) system to indicate parameters that are not specified. A parameter has little impact on the quality of solutions when most of its values are unspecified in `world.pop{`p`}.indiv.genome.X`.

8.11 par.immigration

Ferret has the ability to run multiple populations simultaneously, where the number of populations is controlled by `par.general.NPop`. Section 8.3 briefly discussed the motivations for running multiple populations. The most important of these is that populations operate nearly independently of each other, and for very difficult problems, this redundancy serves as an 'insurance policy' that can help to find the global solution. However, multiple populations are most useful when the firewall between them is not quite perfect, so that they can communicate and cooperate to some small extent.

```
% Immigration
par.immigration.PImmigrate=0.05; % [0 - 1, usually << 1]: Probability of immigration ↘
        ↪ between populations.
```

Ferret controls the interaction between populations through a single 'immigration' probability called `par.immigration.PImmigrate`. When this parameter greater than zero, there is a finite probability, equal to $PImmigrate$, that individual solutions will be swapped at random between populations. Thus, when a superior region of parameter space is discovered by one population, this information will eventually diffuse to other populations when high performing individuals immigrate to those populations and multiply. The exact value of $PImmigrate$ is not usually critical; however, it should be set to a small, but non-zero, probability that should never exceed a few percent. The key idea here is that if you are running multiple populations, then individual populations should never mix via immigration faster than they explore the parameter space and develop on their own. Setting $PImmigrate$ too high would inevitably result in too much mixing between populations, which would eliminate any benefit of running multiple populations, except for slightly decreased memory and CPU requirements associated with computing the distance metrics, as discussed in Section 8.3. I usually set this parameter to a value between 0.01 and 0.05. Of course, `par.immigration.PImmigrate` has no effect when only a single population is used (`par.general.NPop=1`), which is the default setting.

8.12 par.elitism

Elitism is commonly used in genetic algorithms to ensure that the best solutions present in each generation survive the random perturbations due to mutation and the parameter space mixing of crossover events. The basic idea is that a small fraction of the very best solutions are selected each generation and preserved

for the next. Thus, the fitness of the best individual in the population should always improve monotonically for single-objective problems when elitism is used. Likewise, the trade-off surface as a whole should improve monotonically for multi-objective problems, although this is not necessarily true of the individual objectives.

```
% Elitism
par.elitism.mode='normal'; % ['normal' or 'boundary']: Elitism mode.
par.elitism.frac=0.05; % [0 - 1]: Fraction of population that are designated elite.
```

Ferret's elitism operator is very effective, and turning this option on has resulted in improved performance for every problem that I have ever encountered. I therefore strongly encourage you to use this feature, and I recommend setting `par.elitism.frac`, the fraction of the population saved in elites each generation, to at least a few percent. Elitism is turned off when `par.elitism.frac=0`. If you only want a single elite per population, just set `par.elitism.frac` to some arbitrarily small number that is greater than zero; for example, `par.elitism.frac=1e-6`. It is sometimes useful to turn `par.elitism.frac` up as high as about 20% for parameter space-mapping problems, which may be either multi-objective problems, or single-objective problems where `par.selection.FAbsTol`, `par.selection.FRelTol`, or `par.selection.FRankTol` are set to a value greater than zero (see Section 8.6). Such high values of the elitism fraction would probably raise eyebrows for most users experienced with other genetic algorithms. However, the implementation of elitism in Ferret is quite different, and such high values are occasionally beneficial.

The initial round of selection of elites is based on the fitness parameter, for single-objective problems, or on Pareto fitness rank for multi-objective problems. However, parameter space-mapping problems (single or multi-objective) are characterized by the property that they have an optimal region that does not reduce to a single solution. Thus, it is sensible to include a second round of elite selection based on the niching principle, using the same tournament priorities discussed in Section 8.9. For such problems, the most valuable solutions are in the optimal region *and* have a low niche count (see Section 8.9). These solutions are the ones that are preferentially chosen as elites.

You are likely to notice an interesting phenomenon if you carefully observe the behaviour of elites on parameter space-mapping problems, when X-niching is selected. A good example of this type of problem is the demonstration labelled as 'Multi-Objective/Triangle-Subspace-X-Niching', which can be accessed from Ferret's 'Demo' menu. What you should see is that the elites will tend to cluster around the boundaries of the optimal set and effectively 'shepherd' the solutions residing in the interior region. The shepherding behaviour is due to the fact that elite selection is based, in part, on niching; solutions that are on the boundary of the optimal region tend to have fewer near neighbours, and are therefore selected preferentially by the niching operator. Shepherding is extremely helpful in parameter space-mapping problems because these elites form a relatively stable perimeter around the edge of the optimal region, while the interior solutions continue to explore. Their density will be especially high near convex corners of an optimal set, such as the corners of the triangle in the example mentioned above. The overall effect after analyzing the runs is that the interior part of the optimal region will be well-explored, but there will be a somewhat higher density of solutions near the edge; this non-uniform coverage is a good thing, because it helps to discriminate between the interior optimal region, and the non-optimal region on the outside. This is also why it is acceptable to use a relatively high value for `par.elitism.frac` on such problems. Note that this is not likely to cause problems with premature convergence, which could result from a high values of the elitism fraction in some genetic algorithms. Ferret includes many safeguards that explicitly prevent elite solutions from dominating the population too early in the run, when the emphasis should be on exploration rather than the exploitation of the optimal region.

Ferret's elitism operator contains a second parameter called `par.elitism.mode`, which should be set to either 'normal' or 'boundary'. The boundary option places somewhat greater emphasis on shepherding by explicitly selecting solutions that are near the boundary between the optimal and non-optimal region. In my experience, normal elitism provides adequate shepherding and I rarely use the boundary elitism mode in practice. Nevertheless, I think this is an interesting feature that might be useful to some users. I have therefore left it as an option in the code, which might become more prevalent in future versions of the software.

8.13 par.link

Ferret's linkage learning system is perhaps its most innovative feature, and certainly a feature that sets it apart from other GAs. The general philosophy of linkage learning is described in Section 2.2.7, but this section will provide a few more details, and most importantly, show you how to set up linkage learning for your problems. At first sight, the list of options in `par.link` looks somewhat daunting, but the default settings are fine for most problems and the linkage learning algorithm is not particularly sensitive to these settings. For most problems, you can expect to modify perhaps one of these control parameters, or at most a few.

```
% Linkage Learning
par.link.PLink=1; % [0 - 1]: Probability that an individual will be chosen for ↘
        ↪ linkage learning.
par.link.FAbsTol=1e-6; % Absolute fitness range (+/- FAbsTol) to use as a fuzzy ↘
        ↪ band.
par.link.FRelTol=1e-10; % [0 - 1]: Fraction of fitness range to use as a fuzzy ↘
        ↪ fitness band.
par.link.FRankTol=0; % [0 - 1]: Fraction of rank range to use as a fuzzy fitness band.
par.link.maxStrongLinkFrac=0.33; % [0 - 1]
par.link.NLinkMax=100; % [integer >= 1]: Maximum number of linkage attempts per ↘
        ↪ generation.
par.link.lifetime=100; % [integer >= 1]: Lifetime of linkages.
par.link.useOptimalDonors=false; % [logical]: Should a member of the Optimal set be ↘
        ↪ used for the donor?
par.link.acceleration=2; % [real >= 1]: Accelerate decay of weak links.
par.link.failureCost=1; % [real ~1]: Cost associated with higher-order links.
par.link.initialLinkage=0.5; % [0 - 1]: How linked are the parameters initially?
par.link.maxPass=20; % [integer >= 1]: Max number of passes through the linkage queue.
par.link.searchThreshold=0.9; % [0 - 1]: Stop looking for linkages when links are ↘
        ↪ stronger than this value.
par.link.PMultiLink=0.5; % [0 - 1]: Linkage strategy.  0 --> simple links only. ↘
        ↪ 1 --> general, higher order links possible.
par.link.PXOverDonor=1;  % [0 - 1]: Probability to engage in XOver with the donor.
par.link.strategy=0.5; % [0 - 1]: 0 --> opportunistic, 1 --> aggressive.
par.link.PRandomizeNewBBs=1;  % [0 - 1]: Randomize BBs when they form?
par.link.mixDim=0; % [usually 0 - 1, possibly higher]: Greater than 0 if mixing is ↘
        ↪ desired.
par.link.BB=; % [linkage matrix (0-1), or cell array of integer vectors]: Known ↘
        ↪ building blocks. Empty list if none are known.
```

Generally speaking, finding that a problem has fewer linkages is good news because it means that the problem is intrinsically easier. However, the division between linked and non-linked is not quite as cut and dried as one might like. In general, the linkage map is contextual in the sense that what seems linked early on in the evolution may not be linked after many generations, since Ferret will likely be sampling only a small part of the problem's initial parameter space near the optimal region late in the optimization process. The opposite is sometimes also true; links that are not seen early on in a run may become apparent later on when Ferret zooms in on the optimal region. The point is then that the linkage map should never be viewed as a static feature of a problem. Rather, it is more productive to view it as a dynamic map of the parameter space that changes and evolves as Ferret zeros in the optimal region. Ferret incorporates dynamic linkages by setting an exponential lifetime `par.link.lifetime` on all newly discovered links. When a new link is discovered, the corresponding cell in the linkage map increased abruptly, but then gradually decays over the exponential lifetime unless it is rediscovered.

All linkages have a value ranging from 0 (totally unlinked, black in the display) to 1 (totally linked, white in the display). At the start of the run, the linkage between all parameters is set to a value of `par.link.initialLinkage`, which is 0.5 by default. This shows up on the linkage map display as a white diagonal on a half-tone grey background, which indicated that every parameter is 100% linked to itself, and assumed to be liked with 50% probability to all others. Why not set the initial linkage to zero and only add linkages as they are discovered? This is usually a *very* bad idea because it tells Ferret to behave as if all parameters are independent. Ferret will try to take advantage of this information, which is probably false, and there is a good chance that it will converge rapidly to the wrong solution. Setting the initial linkage to 0.5 seems to be a good choice for most problems. A truly cautious user might prefer to set this control parameter to 100%, which causes Ferret to assume complete linkage initially, and and links gradually decay over the linkage lifetime for those that fail to be discovered. However, this is overly cautious for every optimization problem that I have ever tried. An initial linkage of 50% is adequate, and higher decrease the efficiency of the early part of the run, before the initial linkages decay by approximately the generation given by `par.link.lifetime`.

The most important parameter in the par.link structure is the linkage search probability `par.link.PLink`, which should be between 0 and 1 (inclusive). This is the probability that Ferret will examine any individual solution for evidence of linkage, and a judicious choice of this parameter is important to the successful solution of most complex problems. Linkage-learning is turned off when `par.link.PLink=0`. In this case, the linkage map is still determined by `par.link.initialLinkage` at the start of the run, but it does not evolve as the run progresses. This can still be useful, because it effectively determines how many parameters are swapped during an evaluation of the building block crossover operator, as discussed in Section 8.8.2. When linkage learning is enabled, values of `par.link.PLink` between 0.25 and 1 are usually appropriate. Values on the higher end will cause Ferret to spend more time evaluating the fitness function while hunting for linkages, but the linkage map may be more accurate. Try changing *PLink* to a higher setting if your results seem inconsistent and the linkage map varies *drastically* between runs.[3] Note that the linkage probability can be modified by `par.link.NLinkMax`, which sets the maximum number of linkages tested per generation.

Ferret's linkage learning system is designed to dramatically improve the efficiency of the parameter search by dividing large problems into smaller, easier problems. However, the search for parameter space linkage is computationally expensive because it requires extra evaluations of the fitness function. Therefore, the choice of a value for *PLink* represents a trade-off between gains made by a more intelligent search strategy,

[3]Recall that the linkage map changes *dynamically* in response to the fitness function, and different parts of the parameter space may be linked differently. Therefore, the linkage map is not likely to be exactly the same on two different runs because it is discovered stochastically and varies with the path that the population took while searching for the optimal set. It usually has a similar general appearance between runs, however.

and the cost of extra evaluations. Part of the extra cost of linkage-learning is offset because the extra evaluations are recycled and used directly in the optimization. For example, new optimal solutions are often discovered during linkage learning, and these new optimals are never wasted. For most problems that I have encountered in astrophysics, the gains made by setting $PLink$ close to (or often equal to) 100% (`par.link.PLink=1`) outweigh the cost quite dramatically, but this may not be true for other problems. I would therefore recommend that you start by setting `par.link.PLink=1`, and gradually decrease it as you gain experience with your problem. At some point you may find that your parameter space search becomes unreliable. This is a clear sign that you have set $PLink$ too low.

Ferret's linkage-learning system employs two different strategies, which are selected by the probability `par.link.strategy`, which should be set to a value between 0 and 1. Setting `par.link.strategy=0` causes Ferret to always choose an 'opportunistic' strategy, while `par.link.strategy=1` results in an 'aggressive' strategy with 100% probability. Intermediate values are treated as the probability of choosing the aggressive mode. In aggressive mode, a link is considered to be detected anytime that Ferret notices the type of non-linearity that it associates with an indivisible sub-problem in the parameter space. In opportunistic mode, a link is only counted if this type of non-linearity exists, *and* the linkage test directly resulted in an improved solution. Generally, the aggressive mode finds many more linkages than the opportunistic mode, but it may be argued that many links found in aggressive mode are are not useful, because the population has already discovered a region of parameter space where these linkages no longer matter for the improvement of the fitness. The opposing argument is that the mathematical non-linearity has still been shown to exist, and could still be important; just because it did not result in improvement during the single linkage test that found it, does not mean that it is incapable of doing so in future tests. I can see the validity of both arguments and I have tested Ferret using both strategies. Usually, this parameter does not make or break a run, and I usually just use the default value of `par.link.strategy=0.5`.

The `par.link.FAbsTol` and `par.link.FRelTol` parameters are absolute and relative fitness tolerances that are used by Ferret during linkage detection. Differences in fitness are ignored when they are smaller than these values. This is useful for functions whose evaluation is inherently noisy. Linkage can be viewed as a particular type of non-linearity, which can be mimicked by noise in some circumstances. It is therefore advisable to specify a tolerance if your function is noisy and you have characterized the error on each evaluation. If spurious linkages are detected, this could decrease the efficiency of the parameter search because more linkages mean that the problem is not divided into as many smaller sub-problems, and the sub-problems are larger. Note that Ferret usually does a good job at filtering out spurious linkage signals. Non-zero linkage tolerances are really only recommended when you have significant computational errors in your evaluation (i.e. stochastic fitness functions or a lot of roundoff error) and you know them well. `par.link.FRankTol` is a relative tolerance in rank, ranging from 0 to 1, which works like `par.selection.FRankTol`. I almost always leave this value set to zero.

Searching for linkage is a computationally expensive process, and the user can use `par.link.searchThreshold` to limit the effort spent to enhance the strength of linkages that are already quite strong. The searchThreshold parameter is a threshold between 0 and 1. Ferret will not search for new linkages between any pair of parameters when the associated linkage is already greater than $searchThreshold$.

Ferret's linkage learning system tests for linkage by comparing two individuals: the primary individual being tested, and a 'donor'. The control parameter `par.link.useOptimalDonors` determines whether the donor is chosen randomly, or if an optimal donor is chosen from the best solutions found to date. By default, `par.link.useOptimalDonors=false`, which emphasizes the most widespread search for linkages. Setting `par.link.useOptimalDonors=true` may not find as many linkages, but may result in faster

convergence on some problems.

One part of the linkage detection algorithm is mathematically similar to a building block crossover between the primary individual and the donor. Ferret allows the user to insert solutions formed as a result of this crossover operator into the population if they are superior to both parents. This insertion occurs with probability equal to `par.link.PXOverDonor`, which is set to 100% by default. Note that these crossovers must be done anyway when searching for linkage. Keeping any superior solutions that result can be viewed as an opportunistic side-effect of linkage-learning, and I normally leave this parameter set to 100%. The only reason to decrease it is if linkages are discovered *very* frequently, so that this operation becomes the dominant source of an undesirably high selection pressure. I have never actually seen this happen, but you may consider decreasing $PXOverDonor$ if you suspect that this is occurring.

As discussed above, finding relatively few linkages is 'good news' because it means that a problem can effectively be broken down more effectively into smaller sub-problems. How well a problem can be broken into sub-problems depends entirely on the nature of the problem, so it is not usually a good idea to tinker with the linkage map. Nevertheless, Ferret does allow a certain amount of tinkering via the `par.link.maxStrongLinkFrac` and `par.link.BB` control parameters. The maxStrongLinkFrac parameter is a fraction from 0 to 1 that limits the number of 'strong' linkages in the problem, where a strong linkage is defined as one whose value is greater than $par.link.searchThreshold/2$ (0.5 by default). This is accomplished by accelerating the rate that linkages decay whenever the fraction of strong linkages exceeds $maxStrongLinkFrac$. I have never found this feature very useful because most problems aren't really *that* strongly linked. However, the option is there for problems that are very strongly linked, and users who want to try to tune their linkage maps. I would not recommend setting $maxStrongLinkFrac$ less than about 1/3, since this would severely cripple the linkage learning system. Note that it is also possible to accelerate the decay of weak linkages by setting `par.link.acceleration > 1`. Linkages normally have a lifetime equal to `par.link.lifetime`, but the lifetime of the weakest linkages is reduced to $par.link.lifetime/par.link.acceleration$.

Initial known linkages can be set by hand using the `par.link.BB` control parameter. This can be either a cell array of linkages or a linkage matrix. For example, `par.link.BB={[1,2,3],[4,5,6]}` links together parameters 1, 2, and 3, as well as parameters 4, 5, and 6. The same linkages can be established by building the linkage matrix directly:

```
par.link.BB=zeros(6);
par.link.BB(1:3,1:3)=1;
par.link.BB(4:6,4:6)=1;
```

The matrix usage is more flexible because linkage values less than 1 are allowed. Note that this sets the initial linkage values only, and the linkages evolve if linkage learning is turned on. The functionality allowed by `par.link.BB` should be used sparingly, and only when you are certain that you understand the linkage structure of your problem. For most problems, it is best to allow linkage learning to discover the linkage structure automatically without setting values by hand.

Ferret actually contains two distinct (and mutually exclusive) linkage-learning algorithms. The simpler mechanism tries to determine whether a single parameter is linked to some other parameter. The more complex method tries to detect linkage between two *groups* of parameters, which makes it potentially faster and more powerful because each linkage test casts a wider net for linked parameters. However, this 'multi-link' method has a significant drawback. When two groups of parameters are detected as being linked, the code does not really know *which* parameters within the two sets were responsible, or even if the detection represents multiple linkages between the two parameter sets - all it can do is assume that at least one link exists *somewhere* within the sets of parameters that were tested. Ferret contains algorithms to

isolate the linked parameters, but the multi-link method remains somewhat probabilistic in nature. Both methods are powerful, robust, and very well-tested. I have used both the linkage modes successfully on a variety of problems, and I cannot offer universal advice on which technique is better. The multi-link method is certainly more sophisticated and probably more powerful for most problems, but the single linkage mode results in more certain detections. For this reason, I implemented a multi-link probability controlled by `par.link.PMultiLink`, which selects between the single and multi-link methods stochastically. I set this probability to 50% by default, but you can set it higher to more frequently choose the multi-link method, or lower to choose the simple linkage scheme more often. Normally, I hedge my bets by leaving this parameter set to its default value of 0.5.

Since the multi-link method associates a detected linkage with two stochastically chosen groups of parameters, it is inevitable that mathematically non-linked parameters within these groups will occasionally gain linkage strength by virtue of being chosen in the same groups as truly linked parameters. Therefore, the multi-link method contains a slight natural bias towards erroneously linking parameters that have no real linkage. Ferret attempts to isolate the parameters responsible for linkage by recursively passing them through a 'linkage detection queue' for further tests and analysis. This is an expensive process, and the number of passes is limited to `par.link.maxPass`, which is set to 20 by default. Ferret contains effective algorithms to filter out false linkage detections in multi-link mode, but it is also useful to counter the natural bias by directly suppressing them. The essential idea is that mathematically unlinked parameters may occasionally be flagged as linked, but it is more likely that a linkage test involving these parameters will fail. Therefore, we directly punish failed linkages by decreasing the strength of the associated linkages by an amount equal to $par.link.failureCost/par.link.lifetime$. I find it adequate to leave `par.link.failureCost` set to its default value of 1 and rarely play with this parameter. Note that `par.link.failureCost` does not affect the simple (single) linkage method.

After reading this section on `par.link`, you might conclude that linkage detection is a complicated, treacherous business that is full of pitfalls and uncertainty. In some ways, this is true. I have admitted several times in the above paragraphs to not fully understanding which options are best or which values to choose for certain parameters. However, the reason for this uncertainty is often due to the robustness of the linkage-learning system. For example, I can't definitively choose between an aggressive (`par.link.strategy=1`) or opportunistic (`par.link.strategy=0`) linkage strategy, because *both* normally work very well. Likewise, it's difficult to decide between the simple linkage and multi-link algorithms because both work exceptionally well on most problems! In general, you should try to avoid extreme values for the linkage control parameters, and take heart that the default values will normally work just fine. Nevertheless, I have left these options in place for adventurous users.

8.14 par.zoom

Ferret's zooming feature allows a run to automatically zoom in on the optimal set as it is discovered. This is very useful because it effectively rescales Ferret's internal representation of the parameter space, which makes it easier for the niching algorithm to populate the optimal set with solutions, while also increasing the resolution in the console display.

```
% Zooming
par.zoom.NGen=0; % [integer >= 1]: Zoom control.
par.zoom.safety=0.5; % [real, usually <= 1]: Size of buffer zone around optimal set.
par.zoom.allowZoomOut=true; % [logical]: Is zoom-out permitted?
par.zoom.maxPower=1e10; % [real << 1]: Maximum zoom power.
```

Zooming works by determining the maximum and minimum limits of the optimal region of parameter space over the past `par.zoom.NGen` generations. Thus, `par.zoom.NGen` also determines the timescale, in generations, on which zooming responds to changes in the population. No zooming will occur until at least `par.zoom.NGen` generations have passed. Non-optimal solutions may begin to fall outside of the zoomed parameter limits when zooming occurs, and out of range parameters are set to a random value within the zoomed range when this occurs. Thus, it is best to set `par.zoom.NGen` to at least ∼10 generations (`par.zoom.NGen=50` by default), so that regions of parameter space are not lost due to short-term random fluctuations of the optimal set during the search. It is possible for a the zooming feature to zoom out if the boolean parameter `par.zoom.allowZoomOut=true`, but it is much better to avoid losing parts of the optimal region due to over-zealous zooming in the first place. For each parameter, a safety margin or buffer zone is set on either side of the optimal set by `par.zoom.safety`, which is expressed as a fraction of the extent of the optimal region (`par.general.max-par.general.min`). Thus, the default value of 0.5 indicates that a 50% safety margin, relative to the size of the optimal region in each dimension, should be left on either side of the optimal set. `par.zoom.maxPower` sets the maximum zoom power allowed, and is 10^{10} by default.

The zoom feature pays special attention to parameters designated as cyclic. These parameters must remain cyclic, and retain the original mapping from their minimum values to their maximum values. I have taken great care to ensure that cyclic parameters are zoomed correctly, especially when the optimal set straddles the dividing line between the minimum and maximum value (i.e. the $0/2\pi$ boundary for angles). When this occurs, Ferret uses the cyclic property of the parameter in question to re-map the values internally to a continuous set (i.e. not split between the minimum side and the maximum size), approximately centred in the zoomed region. This is very important because it allows Ferret to take advantage of the cyclic property of the variable to make the optimal set continuous, rather than having part of it near 0 and part near 2π. This makes crossovers more likely to succeed, because there will be no lethals formed as a result of crossovers between two distinct islands of solutions near 0 and 2π. It does not matter whether the zoomed region is centred on 0, 2π, or any integer multiple of 2π in Ferret's internal representation of the parameter because it is cyclic.

The console display always re-maps the solutions to the initial range specified by `par.general.min` and `par.general.max` (see Section 8.3), so it may not look like any zooming has actually occurred because values near 0 and 2π are both still displayed. Cyclic variable are certainly zoomed however. Try running the demo called 'Cyclic/Cyclic-Zooming-2D' for 20 generations or so, analyze it, load the History file that is generated, and run the following code on the MATLAB command line to generate a plot:

```
>> index=max(find(~cellfun('isempty', History)));
>> X=History{index}.pop{1}.indiv.genome.XPhys;
>> figure(1); plot(X(1,:), X(2,:), '.')
```

You should see two distinct circular regions filled with solutions, whose bounds do not coincide with the original parameter limits, and the plot will look quite different from the parameter distribution plot shown in the Ferret console display. You can also examine the zoomed limits directly by querying the *par* structure saved with each cell of the History cell array. This is the output that I obtain when I do this:

```
>> History{index}.par.general.min

ans =

    1.9917    1.0456

>> History{index}.par.general.max

ans =

    8.2749    7.3288
```

Clearly, the upper limit has moved outside of the original limit of 2π for both parameters. Solutions are not re-mapped to the original range when History files are saved, but are re-mapped during analysis. The solutions present in the OptimalSolutions file in `OptimalSolutions.X`, are always in the original range specified by `par.general.min` and `par.general.max`.

Re-mapping solutions to the original range is meant to decrease the confusion that can result from cyclic variables. I did not re-map solutions in some earlier versions of Ferret, and this led to a few bug reports that Ferret was not respecting the bounds in `par.general.min` and `par.general.max`. Users were responsible for doing their own re-mapping, which added an additional complication. For this reason, re-mapping is automatic now, and you don't have to worry about re-mapping unless you work with the History files directly.

Zooming always respects discrete parameters. The initial discrete values allowed by `par.general.discreteStep` (see Section 8.3) are unaffected by zooming.

8.15 par.stopping

The most flexible way to stop Ferret is by building a custom stopping criterion, as discussed in Section 4.9.2. However, the `par.stopping` section of the setup file includes simple termination options based on tolerances in X and F.

```
% Stopping criteria
%
% Tolerance on X and F - mainly for single-objective problems.
par.stopping.XTol=0; % [real > 0]: Tolerance on the solution.
par.stopping.FTol=0; % [real > 0]: Tolerance on the energy.
par.stopping.boolean=@or; % [boolean operator]: Should be '@and' or '@or'.
par.stopping.window=25; % [real ~ 1]: Ferret keeps par.tracks.window ↘
        ↪ generations in memory for stopping criteria.
```

This built-in stopping criterion works by accumulating a buffer of X and F values from the optimal solutions of the previous `par.stopping.window` generations. After `par.stopping.window` generations have passed, Ferret starts testing for convergence by comparing the standard deviation of X and F accumulated within the buffer with the tolerances `par.stopping.XTol` and `par.stopping.FTol`:

$$\begin{aligned} \texttt{cond\_X} &= \texttt{all}(\texttt{std}(\texttt{conv.XVec}, 1, 2) <= \texttt{par.stopping.XTol}) \\ \texttt{cond\_F} &= \texttt{all}(\texttt{std}(\texttt{conv.FVec}, 1, 2) <= \texttt{par.stopping.FTol}) \end{aligned} \tag{8.5}$$

These conditions are combined using the boolean function handle `par.stopping.boolean`, which must be set to either @and or @or. When `par.stopping.boolean=@and`, Ferret exits when *both cond_X* and *cond_F* are true. When `par.stopping.boolean=@or`, Ferret exits when *either cond_X* or *cond_F* are true.

8.16 par.analysis

As Ferret runs, it writes a History file to disk every `par.history.NGenPerHistoryFile` generations (see Section 8.2), which contains information about Ferret's state at each generation since the last History file was saved. History files are discussed in detail in Section 4.16.4. Analysis is the process that reloads all of these History files, computes the final set of optimal solutions from the run, and saves this information as an OptimalSolutions.mat file. The analysis process is discussed in Section 4.15, and the structure of an OptimalSolutions.mat file is given in Section 4.18. The `par.analysis` branch of the setup file contains a few options that affect how analysis is done.

```
% Analysis
par.analysis.analyzeWhenDone=false; % [logical]: Do analysis automatically when ↘
        ↪ evolution stops.
par.analysis.conserveMemory=false; % [logical]: Minimize memory usage during ↘
        ↪ analysis? (Analysis will be slower)
par.analysis.maxItNoProgress=100; % [integer >= 1]: Max iterations with no progress ↘
        ↪ before analysis stops.
par.analysis.postProcess=''; % [string]: Name of code to process OptimalSolutions ↘
        ↪ when analysis is done.
```

8.16.1 Manual vs. Automatic Analysis

`par.analysis.analyzeWhenDone` is a logical variable that determines whether or not analysis occurs automatically when the run terminates. This control parameter is set to false by default, which requires you to initiate the analysis manually using Ferret's GUI, or the 'Analyze History' tool, which can be found under the 'Other Tools' heading in the main component selector (see Section 12.2).

Analyzing many solutions manually when they finish can be quite tedious, and I recommend setting `par.analysis.analyzeWhenDone=true` in such situations. Automatic analysis is also useful if you plan to run Ferret from the command line, without graphics, as discussed in Section 4.14.2. Without the Ferret interface, such runs must be analyzed using the Analyze History tool, or automatically by setting `par.analysis.analyzeWhenDone=true`. Chances are good that you want automatic analysis if you have gone to the effort of setting up a batch job with a custom launcher.

8.16.2 The Analysis Algorithm

Analyzing a long run requires Ferret to compare a very large number of solutions loaded from the History files to determine the final optimal set. This is done by means of an iterative algorithm for multi-objective problems, because methods based on directly ranking solutions to obtain the optimal set ($rank = 1$) are too slow to be of practical use for large numbers of solutions. Such methods would require Ferret to compare every individual loaded with every other individual, and hence the time requirements would scale as the square of the number of solutions in the optimal set, and this can be a very large number sometimes.

Note that an iterative algorithm is not required for single-objective problems, which can be analyzed efficiently using a direct approach.

Ferret's iterative analysis algorithm is designed to handle very large runs efficiently, but its iterative nature means that it is technically possible for the algorithm to miss a few solutions when iteratively weeding out sub-optimal solutions from the Pareto surface. Once the History files are loaded, the algorithm works by dividing the solution set in half and applying a binary tournament operator to each pair of solutions, using their saved fitness values, and removing any solution that loses a competition. The tournament priorities discussed in Section 8.6 do not apply; only the fitness matters. Fuzzy tolerances `par.selection.FAbsTol` and `par.selection.FRelTol` are taken into account, but `par.selection.FRankTol` is ignored. This iterative process is repeated until convergence, which is declared when no solution has been removed for `par.analysis.maxItNoProgress` steps.

The convergence criterion does not guarantee that Ferret has eliminated every single non-optimal solution with 100% accuracy, since a chance tournament between two highly optimized solutions on step number $(maxItNoProgress + 1)$ *could*, in principle, detect that one is superior to the other in a Pareto-optimal sense. I have never seen an application where this has been a real problem, but try re-analyzing with an increased value of $maxItNoProgress$ if you notice any suspicious outliers in your optimal set.

8.16.3 Memory Requirements

Loading many History files from a long run can result in very high memory requirements. You should suspect memory problems if you observe any of the following behaviour:

- the analysis is taking much longer than you think it ought to.
- you see the Ferret GUI and other applications becoming very sluggish or completely unresponsive.
- you see memory error messages on the MATLAB console.
- MATLAB is using only a small percentage of the available CPU time, but large amounts of memory.

You can verify how much CPU time and memory MATLAB is using by using the Task Manager on Windows computers, the Activity Monitor on a MacIntosh, or by running 'top' on Linux machines. You have a few options to analyze your run if you notice memory problems.

- The first thing to try is to set `par.analysis.conserveMemory=true` in your code, which tells Ferret's analysis algorithm that memory is a major issue and that it should use the bare minimum required to get the job done.[4] However, the price of running in this mode is that the analysis will take a somewhat longer time to complete.
- If you still can't analyze your run, try clicking on the 'Stop Analysis' button *before* all of your History files load. Note that History files load in reverse order; the last one generated is loaded first, and the first one generated is loaded last. Therefore, the first History files loaded, which were generated near the end of the run, will likely be rich with optimal solutions, while those from the start of the run will contain very few good solutions. You usually won't lose many good solutions by stopping the loading

[4] You can also minimize memory usage by using the Analyze History Tool (see Section 12.2) and choosing 'Minimal Memory' when asked 'Optimize analysis for speed or mimimal memory usage?'. This is exactly the same as setting `par.analysis.conserveMemory=true` in your project.

process before the end. Note that analysis won't *actually* stop (despite the label on the button), but will use whatever has been loaded already to generate the optimal set.

- Finally, if neither of these options work for you, then chances are good that you are attempting a truly massive calculation and you should consider buying more memory or copying your run directory, which includes the History files, to a machine with more memory. You can then analyze your run with the memory-rich machine using the 'Analyze History' tool, which is described in Section 12.2.

8.16.4 Post-Processing of Results

Ferret can automatically run user-defined post-processing code on the optimal solution set when the analysis process is complete. This is done by setting `par.analysis.postProcess` to a string that corresponds to the name of the user's post-processing code, which must be a valid MATLAB function in the project directory or another path loaded by the project's init file (see Section 4.6). By default, `par.analysis.postProcess` is set to an empty string (`par.analysis.postProcess=''`) to indicate that no post-processing code is defined.

Automatic post-processing is discussed in Section 4.19, where the signature of the post-processing function is give. The post-processing code must accept the *OptimalSolutions* structure as an argument, or require no arguments, and no return arguments are allowed.

8.17 par.localOpt

This user's guide has stressed the importance of global techniques for parameter search, optimization, and data-modeling problems in science, engineering, and other technical disciplines. A section discussing Ferret's *local* optimization features may therefore seem surprising and somewhat out of place. However, it turns out that a genetic algorithm like Ferret can benefit tremendously from occasional calls to a local optimizer during a run.

Genetic algorithms are excellent at searching globally, and Ferret contains many features that further improve on these search capabilities. However, they are actually quite poor at zeroing in on a solution to high accuracy once the approximate location of the global solution has been found. Ferret uses sophisticated evolutionary computing techniques to search globally, but can zero in on solutions to high accuracy using a local optimizer. Local optimization is used in two places in Ferret - for polishing at the end of a run, as discussed in Section 4.16 and *during* the optimization procedure, which is the topic of this section. Calling local optimizers once in a while during the run, rather than just at the end, can also sometimes accelerate the convergence of the run substantially.

MATLAB contains a good local optimizer called fminsearch, which is a modified version of the popular Nelder-Mead simplex algorithm. The fminsearch function is Ferret's default local optimizer for single-objective problems, but it cannot be used for multi-objective problems. The Qubist package contains a multi-objective simplex algorithm called SAMOSA (Simple Approach to a Multi-Objective Simplex Algorithm), which Ferret uses for the local optimization step when running multi-objective problems. Neither of these optimizers use any gradient information, which makes them fairly robust on noisy problems or problems where it is difficult to compute the gradient.

```
par.localOpt.startGen=Inf; % [integer]: First generation for local optimization. ↘
        ↪ (Inf or NaN for no set start generation)
par.localOpt.startF=NaN; % [real]: Trigger local optimization when fitness falls ↘
        ↪ below some value. (NaN for no trigger)
par.localOpt.convergeWindow=50; % [integer > 1]: The length of the window (in ↘
        ↪ generations) used to monitor for convergence.
par.localOpt.startFTol=0; % [real]: Trigger local optimization when the change in ↘
        ↪ fitness drops below this value (single-objective only).
par.localOpt.startWhenConverged=false; % [logical]: Trigger local optimization when ↘
        ↪ population appears converged.
par.localOpt.POptimizeOnImprovement=0; % [0 - 1]: Probability of optimization ↘
        ↪ after improvement of best solution.
par.localOpt.POptimizeOnNoImprovement=0; % [0 - 1]: Probability of optimization ↘
        ↪ after no improvement of best solution.
par.localOpt.maxFEval=1000; % [integer]: Maximum number of evaluations allowed for ↘
        ↪ each local optimization.
par.localOpt.tolX=1e-6; % [real > 0, usually small]: Tolerance in X for local ↘
        ↪ optimization.
par.localOpt.tolF=1e-6; % [real > 0, usually small]: Tolerance in F for local ↘
        ↪ optimization.
%
% ------------------------------------
% Selection of optimals to undergo local optimization: The most *restrictive* of ↘
        ↪ these 2 criteria are used: i.e the one that yields
% the fewest optimals.  Each time the local optimizer is called, select a fraction ↘
        ↪ par.localOpt.frac of the solutions in the current
% optimal set (i.e. within the fuzzy tolerance of the best solution for single ↘
        ↪ objective problems, or solutions on the Pareto front
% for multi-objective problems) UP TO a maximum number of optimals equal to ↘
        ↪ par.localOpt.maxNOptimals.
%
par.localOpt.frac=NaN; % [0 - 1]: Fraction of optimals to undergo local ↘
        ↪ optimization. NaN defaults value to a single optimal for single-objective ↘
        ↪  problems, and all optimals on the Pareto front for multi-objective ↘
        ↪  problems.  At least one optimal is chosen.
par.localOpt.maxNOptimals=NaN; % [integer >= 1]: Maximum number of optimals to ↘
        ↪ undergo local optimization.  NaN or Inf means that no additional ↘
        ↪ filtering will be done -- the entire fraction selected ↘
        ↪ (par.localOpt.frac) will be used.
```

Local optimization is not usually useful at the beginning of a run, when the emphasis should be on exploring the space with the genetic algorithm, rather than improving the accuracy of individual solutions. Local optimization is triggered probabilistically, but will not begin until `par.localOpt.startGen`. Setting $startGen = Inf$ or $startGen = NaN$ turns off local optimization.

When local optimization step is triggered, Ferret randomly chooses a fraction of the current optimal set approximately equal to `par.localOpt.frac` for local optimization. `par.localOpt.frac` should be a real number between 0 and 1, or NaN, which indicates that Ferret should always choose a single solution for single-objective problems, or the entire optimal set for multi-objective problems. The number of solutions

chosen will be limited to `par.localOpt.maxNOptimals`, which should be an integer less than or equal to the size of the population (`par.general.popSize`), or either Inf or NaN. Both of these special values signal that there is no additional limit on the absolute number of solutions imposed on top of the optimal solution fraction `par.localOpt.frac`. *At least one solution will be chosen, even if* `par.localOpt.frac=0` *or* `par.localOpt.maxNOptimals=0`*; you should therefore set* `par.localOpt.startGen=Inf` *if you don't want to use local optimization.*

Each of the solutions chosen will be the starting point for local optimization, and will be replaced by the improved solution returned by the local optimizer when it finishes. In my experience, it is usually sufficient to optimize a single solution for single-objective problems, or about 10% of the optimal set for multi-objective problems. My advice is to not be too greedy with the local optimizer. It can be helpful if used in moderation, but remember that Ferret is first and foremost a genetic algorithm, which is vastly more powerful than a local optimizer, and too much local optimization too early on in a run will thwart Ferret's global search. The local optimization step is also quite expensive in terms of the number of function evaluations required. You should therefore use local optimization sparingly and rely on the genetic algorithm to propagate solutions improved by local optimization through the population via crossover and selection. Your setup file should reflect this emphasis on global optimization by the genetic algorithm, rather than locally optimizing a large fraction of the population each generation.

It is extremely important to allow the population time to settle into the rough location of the optimal region before starting local optimization, because the population could converge to a local minimum if the local optimization starts too early in the run. Thus, a robust triggering mechanism is required for local optimization to prevent premature convergence problems. The user can set four different triggers in Ferret. Note that the *first* trigger to activate enables local optimization, which remains enabled for the remainder of the run. Local optimization will start probabilistically from that generation onward. The local optimization triggers are as follows:

1. User-specified start generation: The user can specify by hand when local optimization should start. This is done by setting `par.localOpt.startGen` to the required generation number. Setting *startGen* to NaN or Inf deactivates this trigger.

2. Fitness threshold: Setting `par.localOpt.startF` will cause local optimization to begin when the minimum fitness value reaches this requested value.

3. Fitness convergence: In addition to a fitness threshold, the user can also set a fitness tolerance to trigger the local optimizer. Local optimization will begin when the minimum fitness value has not varied by more than `par.localOpt.startFTol` over the past `par.localOpt.convergeWindow` generations.

4. Population convergence: Section 4.12 offered some insight into how a run converges. The goal is to map out the optimal region of parameter space for multi-objective problems or fuzzy single-objective problems (see Sections 2.1 and 5.1). The number of optimal solutions rises quickly and levels off when this region is found. When `par.localOpt.startWhenConverged=true`, Ferret monitors the population over the past `par.localOpt.convergeWindow` generations and looks for this behaviour as a signal of convergence.

Once enabled, local optimization is controlled by two probabilities: `par.localOpt.POptimizeOnImprovement` and `par.localOpt.POptimizeOnNoImprovement`. The `par.localOpt.POptimizeOnImprovement` control parameter determines the probability that the local optimizer will be called in a given generation when there has been an improvement in the fitness, in a

Pareto-optimal sense, from the previous generation. Likewise, `par.localOpt.POptimizeOnNoImprovement` controls the probability of local optimization when no improvement has been detected. Both of these parameters should be set to a value between 0 and 1. In practice, the main difference between these two control parameters is that calls to the local optimizer triggered by $POptimizeOnImprovement > 0$ will generally occur less frequently (or not at all) late in the run, when improvements to the fitness become rare. This differs from $POptimizeOnNoImprovement > 0$, which will cause the local optimizer to be called at about the same rate throughout the run. I have set both to a value of 0.05 by default, which causes local optimization to occur about every 20 generations for the entire duration of the run, after one of the triggers discussed above have been activated. Local optimization should be used sparingly to avoid premature convergence, and I do not recommend values that are much higher than the defaults.

Ferret allows you to limit the computational cost of each local optimization by specifying the maximum number of fitness function evaluations allowed. This is done by assigning an integer value the control parameter `par.localOpt.maxFEval`, which is set to a default value of 1000. It is possible to do local optimization without evaluation limits by setting `par.localOpt.maxFEval=NaN` or Inf, but I strongly caution against doing this because it is unnecessary, inefficient, and contrary to Ferret's design philosophy, which emphasizes global optimization. To state this more bluntly, why use an advanced genetic algorithm if you are going to spend most of your precious CPU time on local optimization? As I stated above, local optimization should be used *sparingly*. The reason for allowing local optimization during the run, as opposed to just polishing solutions at the end, is to use local information to slightly *nudge* the population toward improvement, but *without overpowering the evolutionary search*. It is also possible to set the desired accuracy for local optimization by setting absolute tolerances for fitness and the parameters by assigning values to `par.localOpt.tolX` and `par.localOpt.tolF`, which are both set to 10^{-6} by default. Each local optimization will stop when the maximum number of evaluations is reached or one of the tolerances is achieved.

8.18 par.polish

Section 8.17 discussed how to call local optimizers during a Ferret run to help improve convergence. However, it is also possible to call local optimizers at the end of a run (after analysis) to *polish* the final set of solutions in the OptimalSolutions.mat file. This is not always necessary, but it can sometimes result in small improvements to the optimal set, especially if local optimization was not used during the run.

```
% *** Polisher Setup ***
%
% par.polish.optimizer='fminsearch';
% par.polish.optimizer='SAMOSA';
% par.polish.optimizer='Anvil';
% par.polish.optimizer='SemiGloSS';
par.polish.optimizer='default'; % [string]: Use defaults.
%
% fminsearch (Single-objective only):
par.polish.fminsearch.NTracks=Inf; % [integer]: Maximum number of tracks. ↘
        ↪ 'Inf' --> all in optimal set.
par.polish.fminsearch.options=[]; % options structure from optimset.
%
% SAMOSA uses its own setup file.  Here, we just add the number of tracks requested.
par.polish.SAMOSA=defaultSAMOSA_setup; % [m-file name]: Name of SAMOSA setup function.
par.polish.SAMOSA.NTracks=Inf; % [integer]: Maximum number of tracks.  'Inf' --> all ↘
        ↪ in optimal set.
%
% Anvil & SemiGloSS uses their own setup files.
par.polish.Anvil=defaultAnvilSetup;  % [m-file name]: Name of Anvil setup function.
par.polish.SemiGloSS=defaultSemiGloSS_setup; % [m-file name]: Name of SemiGloSS ↘
        ↪ setup function.
```

Ferret is capable of using MATLAB's built-in fminsearch optimizer for single-objective problems, or the Qubist optimizers SAMOSA, Anvil, or SemiGloSS for both single and multi-objective problems. The optimizer is chosen by setting the option `par.polish.optimizer` to the name of the chosen optimizer, or to 'default' to use default settings. Acceptable choices for `par.polish.optimizer` are 'fminsearch', 'SAMOSA', 'Anvil', 'SemiGloSS', or 'default'. The 'default' setting uses fminsearch for single-objective problems and SAMOSA for multi-objective problems. Anvil is also sometimes useful for polishing because it is quite a good global optimizer (though not as global as Ferret), and this effectively allows a second opportunity for global optimization. I do not recommend using SemiGloSS for polishing since this is an experimental optimizer that is not yet as powerful or as easy to use as the others. Locust is a 'front line' global optimizer, like Ferret, and cannot be called as a polisher.

The polisher options depend on which optimizer is used, and the setup file contains a section for each of the allowed optimizers. *Note that Ferret uses only the settings in the section corresponding to the chosen optimizers, and all other sections are ignored.* For example, if you set `par.polish.optimizer='SAMOSA'`, then Ferret uses the `par.polish.SAMOSA` section of the setup file, but ignores `par.polish.fminsearch`, `par.polish.Anvil`, and `par.polish.SemiGloSS`.

8.18.1 par.polish.fminsearch

The setup file contains only two options that must be configured to use fminsearch as a polisher. You must set `par.polish.fminsearch.NTracks` to select the number of *tracks* used for polishing. Ferret will attempt to choose $NTracks$ distinct solutions randomly from the optimal set in $OptimalSolutions$, and use these solutions as the starting points for local optimization. The solutions in OptimalSolutions.mat will be replaced by the locally optimized solutions when polishing is complete. Note that setting `par.polish.fminsearch.NTracks=Inf` or NaN tells Ferret to select *all* distinct solutions within the

optimal set that have the best *true* rank, where by 'true rank' I mean that fuzziness introduced by `par.selection.FAbsTol`, `par.selection.FAbsTol`, and `par.selection.FAbsTol` (see Section 8.6) is ignored. This usually results in a single solution being chosen for single-objective problems, but normally results in many solutions being chosen for multi-objective problems.

The remaining options for fminsearch are contained in the `par.polish.fminsearch.options` field, which should be either an empty set (`par.polish.fminsearch.options=[]`) to use the fminsearch default settings, or an fminsearch options structure generated by using MATLAB's optimset command. Please refer to MATLAB's documentation for details on optimset and fminsearch.

8.18.2 par.polish.SAMOSA

SAMOSA requires its own setup file, which is specified by setting `par.polish.SAMOSA` to the name of a valid SAMOSA setup file. *The SAMOSA setup file is called directly as an m-file from Ferret's setup file. It is therefore important that the name is not in quotes. For example,* `par.polish.SAMOSA=SAMOSA_setup` *is OK, but* `par.polish.SAMOSA='SAMOSA_setup'` *will fail.* By default, `par.polish.SAMOSA` is set to SAMOSA's default polisher setup file: `par.polish.SAMOSA=defaultSAMOSA_setup`.

It is important to note that Ferret ignores the default file that SAMOSA uses when called as a standalone optimizer: [Qubist_Home]/user/SAMOSA/defaults. The defaults for using SAMOSA as a polisher are located in [Qubist_Home]/user/Ferret/defaults. Please pay close attention to the warning in the [Qubist_Home]/user/Ferret/defaults/defaultSAMOSA_setup.m:

```
% !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
% WARNING: THIS FILE IS ONLY CALLED WHEN USING SAMOSA AS A POLISHER.
% SAMOSA'S STANDALONE DEFAULT OPTIONS ARE GIVEN IN
% Qubist/user/SAMOSA/defaults.  SOME FIELDS ARE COMMENTED OUT IN THIS FILE
% BECAUSE FERRET OVERWRITES THEM USING VALUES FROM THE FERRETSETUP FILE.
% !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Ferret contains an m-file called translateFerretToSAMOSA.m that automatically fills in missing information from SAMOSA's setup file using appropriate values from Ferret's own setup file. A template *SAMOSA_setup* file is located in [Qubist_Home]/user/Ferret/templates/polisherTemplates/SAMOSA_setup_template.m, which should be identical to the defaultSAMOSA.m polisher file. To customize your SAMOSA setup, you should copy *SAMOSA_setup_template.m* to your project directory, rename it to *SAMOSA_setup.m*, or another name of your choosing, and set `par.polish.SAMOSA` to the name of this m-file in your Ferret setup file. When modifying your SAMOSA setup file, it does no harm to fill in the commented out fields, but these values will be over-written when the polisher is called.

Like fminsearch, using SAMOSA as a polisher requires that you specify the number of tracks for polishing. This is done by setting `par.polish.SAMOSA.NTracks`, which is interpreted by Ferret in the same way as `par.polish.fminsearch.NTracks` discussed in Section 8.18.1. Note that the `par.polish.SAMOSA.NTracks` line simple adds a field to the `par.polish.SAMOSA` structure and therefore *must* be after the `par.polish.SAMOSA` line.

8.18.3 par.polish.Anvil & par.polish.SemiGloSS

Like SAMOSA, these polishers require their own setup files. Templates can be found in [Qubist_Home]/user/Ferret/templates/polisherTemplates/AnvilSetup_template.m and [Qubist_Home]/user/Ferret/templates/polisherTemplates/SemiGloSS_setup_template.m, and the corresponding default setup files are in [Qubist_Home]/user/Ferret/defaults. Note that these default files are distinct from the default files in [Qubist_Home]/user/Anvil/defaults/defaultAnvilSetup.m and [Qubist_Home]/user/SemiGloSS/defaults/defaultSemiGloSS_setup.m, which are called when Anvil and SemiGloSS are used as standalone optimizers.

```
par.polish.Anvil=AnvilSetup;
par.polish.SemiGloSS=SemiGloSS_setup;
```

`par.polish.Anvil` and `par.polish.SemiGloSS` should be set to call valid setup files if you plan to use these optimizers. The same warning applies that I gave in Section 8.18.2: these options must not be set to strings or function handles. Rather, these lines are intended to be actual function calls like

```
par.polish.SAMOSA=defaultSAMOSA_setup
```

Note that the setup files for Anvil and SemiGloSS specify the number of tracks internally, and you therefore do need to add an 'NTracks' field to your Ferret setup file to use these optimizers as polishers. Once again, I caution that SemiGloSS is experimental and frankly just not as good as the other Qubist optimizers yet. I do not recommend its use as a polisher for important runs.

8.19 par.movie

Ferret is able to generate movies of the console window from a run, which is especially useful for diagnosing setup problems because it allows you to study how the population evolves and converges in real time.

```
% Movies
par.movie.makeMovie=false; % [logical]: Generate a movie of the run?
par.movie.frameRate=3; % [integer >= 1]: Movie frame rate.
```

Movies are normally generated from the console by selecting the 'Show Movie' button or by using the Qubist Movie Tool (see Section 12.6), but Ferret can also be configured to generate them automatically at the end of a run by setting `par.movie.makeMovie=true`.

Whether a movie is generated automatically or from the Ferret console, the frame rate is controlled by `par.movie.frameRate`, which is in frames per second. The Movie Tool can be used to modify the frame rate interactively.

8.20 par.interface

The setup file contains several options in `par.interface` that affect the appearance of the console and analysis window, but have no effect on the actual results.

```
% Graphics & messages
par.interface.graphics=1; % [integer: 1, 0, or -1]: Full graphics (1), ↘
        ↪ low graphics (0), or no graphics (-1)?
par.interface.myColorMap='bone'; % [string: colormap name] User choice for colormap.
par.interface.myPlot=''; % [string]: Name of custom plot function.
par.interface.titleFontSize=12; % [integer >= 1]: Self-explanatory.
par.interface.labelFontSize=10; % [integer >= 1]: Self-explanatory.
par.interface.axisFontSize=8; % [integer >= 1]: Self-explanatory.
par.interface.xAxis.type='X'; % ['X' or 'F']: Default X-axis type
par.interface.xAxis.value=1; % [1, NPar] or [1,NObj]: Default X-axis variable
par.interface.yAxis.type='X'; % ['X' or 'F']: Default Y-axis type
par.interface.yAxis.value=2; % [1, NPar] or [1,NObj]: Default Y-axis variable
par.interface.zAxis.type='F'; % ['X' or 'F']: Default Z-axis type
par.interface.zAxis.value=1; % [1, NPar] or [1,NObj]: Default Z-axis variable
par.interface.NPix=25; % [integer >= 1]: Size of the grid for contour and mesh plots.
par.interface.NContours=10; % [integer >= 1]: Number of contour levels for contour ↘
        ↪ plots.
```

Setting `par.interface.graphics=1` causes Ferret to start with the default full graphics console seen in Figure 4.2. Setting this flag to 0 causes Ferret to start with the minimal graphics window seen in Figure 4.3. The minimal graphics console is identical to the interface that results from pressing the 'Graphics Off' button in the console. Recall that closing the low graphics window causes Ferret to truly run with no graphics. Note that instructions are given in Section 4.14.2 to start Ferret without any user interface whatsoever, which corresponds to setting `par.interface.graphics=-1`.

`par.interface.myColorMap` is a string that corresponds to one of MATLAB's colormaps. The default is `par.interface.myColorMap='bone'`, which is a slightly bluish black and white scheme that I think looks good with Ferret's silvery background. Other available colormaps can be found by typing 'help graph3d' in the console window.

`par.interface.myPlot` is the name of the user's optional m-file for custom plots that appear in the 'User Plot' axes of the console and analysis windows. Please refer to Section 4.10 for instructions on how to use the myPlot feature.

`par.interface.titleFontSize`, `par.interface.labelFontSize`, and `par.interface.axisFontSize` are integers that control the size (in points) of the relevant text in the analysis window. It is likely that the apparent size will differ somewhat depending on your operating system, MATLAB version, and your choice of skin (see Section 2.8).

`par.interface.xAxis.type` should be set to a string 'X' or 'F' to set the type of quantity plotted on the x-axis. `par.interface.xAxis.type='X'` indicates that a parameter is initially displayed on the x-axis, while `par.interface.xAxis.type='F'` selects a fitness value. Of course, this can be changed at any time from the console window.

`par.interface.xAxis.value` selects the parameter or objective number that will be used initially for plotting. Ferret's plotting routines contain a few 'sanity checks' to make sure that the requested parameter or fitness value actually exists, and a sensible value is chosen if not. Similar fields also exist in `par.interface.yAxis` and `par.interface.zAxis` to set defaults for the y and z axes. These are initial values only, and can be changed at any time from the console window.

`par.interface.NPix` is an integer value that controls how many pixels are used to bin two-dimensional

projections of solutions to produce image or contour plots. `par.interface.NContours` controls the default number of contours to use for contour plots. These values can also be controlled interactively using the 'Contour/Image plot Options' interface discussed in Section 5.4. Please refer to that section for further information.

Chapter 9

SAMOSA

'Pound for pound, the amoeba is the most vicious animal on Earth.'

-Anonymous

9.1 Introduction

SAMOSA is a simple but surprisingly powerful optimizer based on the well-known Nelder-Mead simplex method, whose name is an acronym that stands for 'Simple Approach to a Multi-Objective Simplex Algorithm'. If you have used MATLAB's built-in fminsearch function, then you have already used a simplex optimizer and SAMOSA's behaviour will seem familiar, at least on single-objective problems. If you are curious about how the simplex method works, I recommend reading the chapter on optimization methods in the one of the Numerical Recipes books, some of which are available online free of charge at http://www.nrbook.com. As you may have guessed (or read earlier in this user's guide), the main difference between SAMOSA and a standard simplex algorithm is that SAMOSA is designed for multi-objective optimization problems, which are not possible with the standard implementation of the Nelder-Mead simplex algorithm. I developed this tool because I needed a lightweight multi-objective local optimizer and polisher for Ferret, and to a lesser extent, Locust. SAMOSA suits this purpose extremely well, and I am very pleased with this little program. The Anvil simulated annealing code and the SemiGloSS 'solution spray' optimizer can also be used as polishers at the end of a run (see Section 8.18), but SAMOSA is the default for both Ferret's local optimization step (see Section 8.17) and polisher for multi-objective runs, and my usual choice for both of these purposes. Note that it can also be used as a standalone multi-objective local optimizer.

SAMOSA is very simple to configure, and section 9.10 will run through its configuration options. However, you should realize that SAMOSA can read the setup files used to control Ferret or Locust. If you have put together a project that already works with one of these optimizers, then it's already set up to work with SAMOSA! There are functions in the Qubist distribution called 'translateFerretToSAMOSA.m' and 'translateLocustToSAMOSA.m' that read Ferret or Locust setup files and make sensible choices for related SAMOSA settings. This is usually adequate for polishing Ferret solutions, but you should still configure

your own SAMOSA_setup file for some additional fine control if you want to use it as a standalone optimizer.

9.2 SAMOSA as a Standalone Optimizer

9.2.1 Starting SAMOSA

To use SAMOSA as a standalone optimizer, it can be launched from the Qubist component selector (see Figure 3.3), which is started by running your launchQubist.m file. Alternatively, you can run the 'startSAMOSA' function in the Qubist home directory, but this will load demos only, and not user-defined projects. This will bring up the simple interface shown in Figure 9.1.

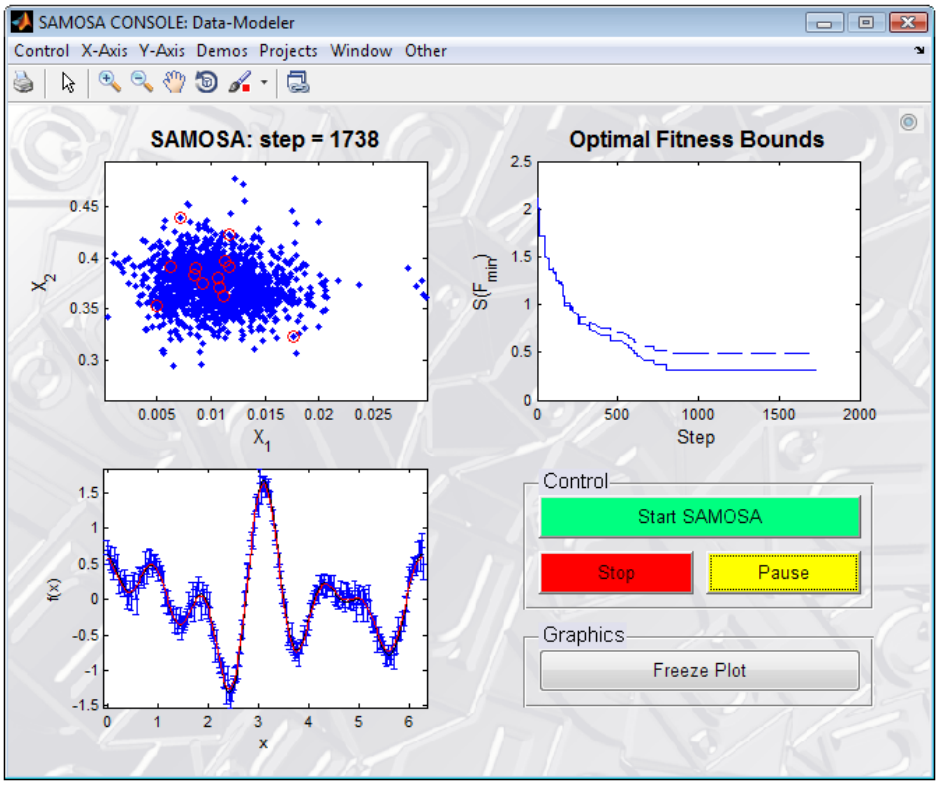

Figure 9.1: SAMOSA running the 'Data-Modeling/Data-Modeler' problem from the [Qubist_Home]/demos directory as a standalone optimizer.

SAMOSA's interface works very much like a simplified version of the Ferret console, and its use should be fairly obvious if you have read Section 4.3. You simply select your demo or project from the relevant menu and click the 'Start' button. There are buttons to stop or pause the run, and also a 'Freeze Plot' button that stops the graphics but keeps SAMOSA running on the MATLAB command line. SAMOSA can be run without any graphics whatsoever by building a custom launch program, as discussed in Section 9.9.

9.2.2 SAMOSA's Graphical Interface

The upper left graph of Figure 9.1 shows two different types of points with quite different meanings in the context of the simplex algorithm:

- Imagine a simplex as an N-dimensional generalization of a triangle or tetrahedron, where N is the number of parameters of the problem, and $P = 1 + N$ is the number of vertices on the simplex. The red open circles represent these vertices as the simplex contorts and crawls through the parameter space. Thus the red open circles represent the state of the simplex at a given step.
- SAMOSA *accumulates* optimal solutions as it crawls through the parameter space, and these optimal solutions are shown as filled blue circles. Unlike Ferret and Locust, SAMOSA does not require an analysis step to filter out optimal solutions. Rather, the list of optimal solutions is kept up to date as SAMOSA runs by adding new optimals as they are discovered, and removing old ones if and when they are no longer optimal. Note that SAMOSA is otherwise much like Ferret and Locust in that it will generally find many optimals for multi-objective problems, or single-objective problems where a fuzzy tolerance (`par.general.FAbsTol > 0`) has been set (see Section 9.10.3).

The projection shown in the upper left graph of Figure 9.1 can be changed by selecting a parameter or fitness value from the 'X-Axis' or 'Y-Axis' menus. SAMOSA does not support three-dimensional graphics at this time.

The graph on the upper right of the SAMOSA window shows the fitness bounds of the optimal set as a function of step number. A solid line shows the true minimum value for each objective, and a dashed line shows the maximum value of the objective within the optimal set. Recall that the optimal set can contain a range of fitness values for multi-objective problems and single-objective 'fuzzy' problems. The y-axis is labelled as '$S(F_{min})$', where 'S' is a 'squashing transformation' defined as

$$S(F_{min}) = sign(F_{min}) \log_{10} \left[(1 + |F_{min}|) \right]. \tag{9.1}$$

Why display such a complicated function? Note that $|S(F_{min})| \approx \log_{10} F_{min}$ when $F_{min} \gg 1$, but $|S(F_{min})| = 0$ when $F_{min} = 0$. These properties allow the graph to display fitness values with a large dynamic range on the same graph. Moreover, $S(F_{min})$ has the same sign as F_{min}, and the graph is continuous if the fitness value changes sign. The top two panels of Ferret's console window (see Figure 4.2) solve this problem by changing automatically to a split panel if a fitness function with a large dynamic range changes sign. The top panel shows positive logarithmic values and the bottom panel shows negative logarithmic values when both positive and negative values are present. However, the logic controlling this display is quite complex, and not really suitable for a lightweight optimizer like SAMOSA. The much simpler 'squashing transformation' implementation in SAMOSA's interface takes some getting used to, but overall I think this suits the optimizer better than Ferret's more complicated system would.

The bottom left plot of the SAMOSA window is the 'User Plot', which works exactly like the corresponding plot in Ferret (see Section 4.10) and uses the same syntax (see Section 9.7). This plot will display 'NO USER PLOT' if one has not been defined for the problem.

9.2.3 Setup File & Defaults

SAMOSA's default setup file (analogous to the defaultFerretSetup.m file; see Section 4.7 for details) is [Qubist_Home]/user/SAMOSA/defaults/defaultSAMOSA_setup.m. A full template of SAMOSA settings is

located in [Qubist_Home]/user/SAMOSA/templates/SAMOSA_setup_template.m. This file can be copied to your project directory, (normally) renamed to SAMOSA_setup, and modified to suit your problem. The details of the SAMOSA setup file are covered in Section 9.10.

9.3 SAMOSA as a Polisher

Using SAMOSA as a polisher is easy. You just specify `par.polish.optimizer='SAMOSA'` in your Ferret or Locust setup file, and SAMOSA will be called automatically as the polisher when you press the 'Polish' button in the analysis window of either of these optimizers. Usually I set `par.polish.optimizer='default'`, which causes SAMOSA to be used for multi-objective problems and MATLAB's built-in fminsearch optimizer for single-objective problems.

Recall that SAMOSA uses a different default file when it is called as a polisher by Ferret or Locust. The default files for polishing are [Qubist_Home]/user/Ferret/defaults/defaultSAMOSA_setup.m and [Qubist_Home]/user/Locust/defaults/defaultSAMOSA_setup.m. Normally, you don't need to worry about this. Ferret and Locust will use these defaults and automatically modify them using their own setup files.

9.4 The init Function

SAMOSA's init function is called exactly like Ferret's, which is explained in Section 4.6:

```
function extPar=init
```

Note that this is the standard technique that causes *extPar* to be passed to SAMOSA's fitness function as a local variable (see Section 9.5). SAMOSA can also take advantage of the 'global *extPar_*' technique discussed in Section 4.6.2. Note that the same comments and warning from that section apply to SAMOSA. This is considered an advanced technique that is designed for occasional situations when *extPar* needs to be shared with other functions via a global variable. Do not use this method as your standard way of passing the data from your init file to your fitness function.

9.5 The Fitness Function

A SAMOSA fitness function has the same signature as a Ferret fitness function (see Section 4.8), except that the little-used 'XPhysMod' return argument is not supported by SAMOSA. Thus, the only difference is that you can't modify 'X' (parameter values) from within your fitness function when using SAMOSA, and any such attempt will be ignored. If your fitness function works with Ferret, then it will work with SAMOSA without any changes.

The signature of a SAMOSA fitness function is as follows:

```
[F, {auxOutput}, {saveData}]=fitness(X, {extPar})
%
% *** Braces {} indicate optional fields. ***
%
% Input fields:
% X          -->   [NGenes x N matrix]
% extPar     -->   MATLAB structure
%
% Output fields:
% F          -->   [NObj x N matrix] (required)
% auxOutput  -->   {N-element MATLAB cell array} or {empty cell array} (optional)
% saveData   -->   {N-element MATLAB cell array} or {empty cell array} (optional)
```

The fitness F, auxiliary output *auxOutput*, and *saveData* have the same meanings as in a Ferret fitness function and are explained in Section 4.8. Recall that *extPar* is the external (to SAMOSA) parameter structure generated from the user's init function. The full set of SAMOSA's parameters from the SAMOSA or Ferret setup file can be accessed from the fitness function as `extPar.QubistPar`.

9.6 The Output Function and the Simplex Data structure

SAMOSA supports an output function that is similar, though not identical, to Ferret's output function, which was discussed in Section 4.9. Recall that Ferret's output function receives the entire *world* - Ferret's top-level data structure, which contains all that there is to know about every solution at the current generation, as well as some other information about the internal state of the genetic algorithm. SAMOSA's top-level data structure is called the *simplex*, which contains the entire state of the SAMOSA's simplex. After each move, this structure is sent to the output function, whose signature is as follows:

```
function output(simplex)
```

The output function is not allowed to return values, to prevent it from making potentially dangerous changes to the state of the simplex. Note that Ferret and Locust are actually allowed to modify their top-level data structures, as discussed in Sections 4.9.3 and 11.8.4, but these are rarely-used features that are intended for advanced users. This functionality is not implemented in SAMOSA.

9.6.1 The Simplex Data Structure

The *simplex* data structure is organized as follows:

```
simplex
       |
       .X          % Matrix of parameter values.  Each column represents the
       |           % parameters for a vertex of the simplex.
       |
       .F          % Matrix of fitness values.  Each column represents the fitness for
       |           % a vertex of the simplex.
       |
       .auxOutput  % Cell array of auxiliary output values.  See fitness function
       |           % syntax.  auxOutput is empty {} when not used.  When used, each
       |           % cell index corresponds to its respective columns in the X and F
       |           % matrices.
       |
       .saveData   % saveData output for compatibility with Ferret.  See fitness
       |           % function syntax.  Works like the auxOutput field.
       |
       .rank       % Vertex ranks.  The best vertex has rank=1.
       |
       .ihi        % Index of the worst vertex.
       |
       .move       % The type of the last successful move.
       |
       .iMove      % The index of the last successfully moved vertex.
       |
       .acceptMove % Boolean field.  True if the last move was accepted, and false if
       |           % not.
       |
       .tol        % Tolerances for stopping.
       |   |
       |   .X      % The tolerance in X.  Should be either a scalar or a column vector
       |   |       % with the same number of rows as X.
       |   |
       |   .F      % The tolerance in F.  Should be either a scalar or a column vector
       |           % with the same number of rows as F.
       |
       .evalCount  % Number of evaluations of the fitness function made.
       |
       .par        % The par structure generated by the setup file.  Includes all
           |       % control parameters.
           |
           .user.extPar  % User-defined parameters from init.
```

The `simplex.X`, `simplex.F`, and `simplex.auxOutput` fields are identical to their counterparts in Ferret. The *simplex* structure also contains a *saveData* field, which is handled differently than the corresponding structure in Ferret. In Ferret, *saveData* is written to disk and not carried around with the *world*. This is sometimes useful for saving intermediate results that would require a lot of memory if they were carried around with the *world*. SAMOSA allows a *saveData* field for compatibility with Ferret, but it is carried as part of the *simplex* structure. This is usually fine because the *simplex* structure has only $1 + N$ vertices,

where N is the number of parameters. In contrast, Ferret's *world* may contain several hundred individuals or more. *Nevertheless, you should not load large data structures into saveData when using SAMOSA unless your computer has sufficient memory.*

`simplex.rank` and `simplex.ihi` are respectively the ranks of the vertices, and the index of the 'highest' vertex with the worst fitness, in a Pareto-optimal sense. Ranks are enumerated such that $rank = 1$ corresponds to the best vertex. `simplex.ihi` should therefore correspond to the index (or indices) where `simplex.rank` is maximal.

`simplex.move` is the name of the last successful move. This field can take on the following values:

- 'none': on the first step only, before any vertices have actually moved.
- 'reflect': a reflection of the worst vertex through the opposite face.
- 'expand': an expansion of the same vertex.
- 'contract1D': a contraction of the simplex in 1 dimension.
- 'contractMultiD': a contraction of all vertices.

You can view these steps as they occur by setting `par.interface.verbose=2`. `simplex.iMove` is the index or indices of the last moved vertex or vertices. `simplex.acceptMove` is a boolean field that indicates whether the last move was accepted or rejected. It should always have a value of 'true' in the output function, since output is only called for accepted moves.

9.6.2 Built-in Stopping Criteria

The simplex structure contains fields `simplex.tol.X` and `simplex.tol.F`, which represent SAMOSA's stopping criteria for X and F respectively. These fields are column vectors with P rows, and are generated from `par.stopping.XTol` and `par.stopping.FTol` in the setup file. The rules for stopping are that convergence must be detected in *either* X *or* F, such that the range in X or F falls below the appropriate tolerance `simplex.tol.X` for all parameters, or `simplex.tol.F` for all objectives:

$$all(max(simplex.X, [], 2) - min(simplex.X, [], 2) < simplex.tol.X) \tag{9.2}$$

or the range in F falls below simplex.tol.F:

$$all(max(simplex.F, [], 2) - min(simplex.F, [], 2) < simplex.tol.F). \tag{9.3}$$

I find this to be a useful stopping criterion for single-objective problems, but it is probably not useful if your problem has multiple objectives. For multi-objective problems or more complicated single-objective problems, you can define your own custom stopping criterion in the output function, using the technique discussd in Section 9.6.2. Note that `simplex.evalCount` is a count of the number of solutions evaluated by the fitness function. SAMOSA will also stop when `simplex.evalCount=par.stopping.maxFEval`.

Finally, `simplex.par` is the parameter structure loaded from the setup file. The user's external parameters, which are passed to the fitness function, are in `simplex.par.user.extPar`.

9.6.3 Custom Stopping Criterion

Like Ferret's output function, SAMOSA's output function can be used to perform side calculations, but the user is responsible for writing any required results to disk, since the output function does not return any variables to SAMOSA. The main use of the output function is for implementing a customized stopping criterion. SAMOSA can be stopped by the following pseudo-code, in analogy to Ferret's custom stopping method, which is demonstrated by the code block in Section 4.9.2:

```
function output(simplex)

if checkStop(simplex)
        abortQubist;
end
```

Here, the Qubist function `abortQubist` is called to stop the run if the `checkStop` function returns a value of true, which has the effect of setting the global variable `abortSAMOSA_=true`. The `abortQubist` function determines which optimizer is running and sends a stop signal by setting the appropriate global variable, which is called $abortSAMOSA\_$ in this case. Note that you *could* achieve the same effect by setting `abortSAMOSA_=1` directly, but this is not recommended because this stop signal would not be recognized by the other optimizers, unless this is the behaviour that you want. The `abortQubist` method is preferred because this signal is understood by all Qubist optimizers.

You may be wondering why SAMOSA requires a different stop signal than Ferret. The problem is that SAMOSA can be called from an active Ferret run during the local optimization step, and *both* optimizers would be stopped by an abort signal intended only for SAMOSA if they used the same global variable. You will see in Chapter 10 that Anvil also uses its own stop signal called $abortAnvil\_$. However, Locust can use the same abort signal as Ferret ($abort\_ = 1$) because Locust is not used as a polisher and is never running at the same time as Ferret. My advice is to just call `abortQubist` for stopping and not worry about it. Note that SAMOSA does not distinguish between a regular stop and an 'emergency' stop (`abortQubist(2)` in Ferret).

9.7 The myPlot function

SAMOSA's myPlot function follows Ferret's model and the syntax is identical:

```
function myPlot(X)
function myPlot(X,extPar)
function myPlot(X,F,extPar)
function myPlot(X,F,auxOutput,extPar)
function myPlot(X,F,auxOutput,rankPareto,extPar)
```

Please refer to Section 4.10 for details. If your project's myPlot function works when using Ferret, then it should also work with SAMOSA without any modifications.

9.8 The OptimalSolutions File

When run as a standalone optimizer through the user interface, SAMOSA will write its results to an OptimalSolutions.mat file in a data subdirectory of the project directory. The name of the subdirectory is specified by the field `par.history.dataDir` in the setup file, and is usually called 'SAMOSA_data'. The *OptimalSolutions* structure is organized as follows:

```
OptimalSolutions
                |
                .X
                |
                .F
                |
                .auxOutput
                |
                .saveData
                |
                .par
```

These fields have exactly the same meaning as their counterparts in the *simplex* data structure (see Section 9.6), except that X and F will contain N_{opt} columns, where N_{opt} is the number of optimals discovered during the run. The *auxOutput* and *saveData* fields will be empty cell arrays if the fitness function does not return the corresponding output arguments, or they may contain N_{opt} cells.

When SAMOSA is used as a polisher, the OptimalSolutions.mat file may be replaced by a file named 'PolishedSolutions.mat'. Polished solutions are always added to *OptimalSolutions*, so it is unlikely that you would need to examine the PolishedSolutions.mat file directly.

9.9 Running SAMOSA from the Command Line

SAMOSA can also be run from the command line, by the following call:

```
[OptimalSolutions, simplex]=SAMOSA(par);
```

where *par* is the structure returned by the SAMOSA setup file. The following is a more detailed custom launcher for SAMOSA. Here I assume the following:

- QH is a string containing the global path to your Qubist home directory. On my Windows computer, this is 'C:\Users\JFiege\Documents\Qubist'.
- The project is one of the Qubist demos, called 'Data-Modeling/Data-Modeler-Small-PopSize'. This demo has all the bells and whistles of a more complicated project.
- The fitness function has the added complication that it contains a call to 'isAbortEval', which helps with parallel processing in Ferret and Locust. This function must be added to the path to avoid an error, even though SAMOSA doesn't do parallel processing.
- We don't want to run with graphics.

- We don't want to mess around with loading files at the end of the run. We want the *simplex* and *OptimalSolutions* structures as output parameters.

The following custom launcher satisfies these requirements. The code for this example can be found in the file [Qubist_Home]/user/SAMOSA/scripts/customLauncher.m.

```
function [OptimalSolutions, simplex]=customLauncher
% Modify this script if you need to make a custom launcher for SAMOSA that
% runs from the command line.

% Set the Qubist component.
global currentComponent_
currentComponent_='SAMOSA';
%
% Reset the path:
path(pathdef);
%
% Set QH=Qubist Home directory.
QH='/Users/fiege/Documents/Qubist_Builder_PL/Qubist';
%
% Add Qubist paths:
addpath(QH);
setQubistPath(QH);
%
% Add the project directory.
addpath(genpath(fullfile(QH, 'demos', 'Data-Modeling', 'Data-Modeler')));
%
% Force Qubist abort status to 0 so that SAMOSA can run.
forceAbortQubist(0);
%
% Load the default SAMOSA setup file and modify it using the project's setup file.
par=defaultSAMOSA_setup;
par=SAMOSA_setup(par);
%
% Turn off graphics.
par.interface.graphics=-1;
%
% Load the init file, and add it to par.
par.user.extPar=init;
%
% ------------------------------------------------------------------------
% Validate the fitness function.  Comment out these lines if to are running
% your project many times and are *sure* that your fitness function works.
validateFitnessFunction('SAMOSA', par);
%
% Abort if validation fails.
if getQubistAbortStatus; % Checks the abort status.
    disp('Stopping due to problems with the fitness function...');
    return
end
% ------------------------------------------------------------------------
%
% Start SAMOSA
[OptimalSolutions, simplex]=SAMOSA(par);
```

9.10 The SAMOSA_setup File

This section will run through all of SAMOSA's setup parameters. This optimizer is *tiny* compared to Ferret, or even Anvil or Locust, so the setup files do not have many options. A template setup file is located in [Qubist_Home]/Qubist/user/SAMOSA/templates/SAMOSA_setup_template.m. Normally, this file should be copied to the project directory and renamed to SAMOSA_setup.m.

9.10.1 par.user

The `par.user` fields tell SAMOSA the name of your fitness function and optional output function. They work exactly like the corresponding fields in Ferret (see Section 8.1).

```
% User
par.user.fitnessFcn='fitness'; % [string]: Name of the fitness function.
par.user.output=''; % [string]: Name of optional user-defined output function called ↘
        ↪ each generation.
```

The fitness function (`par.user.fitnessFcn`) is required, unless you want it to default to the standard name 'fitness', but the output function is optional. An empty string (`''`) indicates that no output function has been defined.

9.10.2 par.history

```
% History
par.history.dataDir='SAMOSA_data'; % [string]: Directory for SAMOSA data files, etc.
```

The History section of the setup file contains a single field `par.history.dataDir`, which gives the location of your run's data directory. This is most often a local path inside of the project directory, as shown here. However, it is also possible to specify a directory elsewhere on the file system by specifying a global path.

`par.history.dataDir` works exactly like the corresponding field in Ferret's setup file, and is discussed in greater detail in Section 8.2. Note that `par.history` is a slight misnomer because SAMOSA does not actually use History files. However, this name is used to maintain consistency with the setup files used by Ferret and Locust.

9.10.3 par.general

The `par.general` section of the setup file contains settings to tell SAMOSA about your problem's parameters and other basic information to initialize the simplex. Many of these fields are identical to Ferret's `par.general` field, discussed in Section 8.3.

```
% General
par.general.XLabels=; % [Cell array of strings]: Give names to some or all ↘
        ↪ parameters: 'A','B',...
par.general.FLabels=; % [Cell array of strings]: Give names to some or all fitness ↘
        ↪ values: 'FA','FB',...
par.general.X0=[]; % [real column or row vector, length=number of ↘
        ↪ parameters]: Initial central position of simplex.
par.general.min=[]; % [real vector]: Min bounds on X: Leave empty for unbounded min.
par.general.max=[]; % [real vector]: Max bounds on X: Leave empty for unbounded max.
par.general.eps=0.25; % [real > 0]: Initial scatter of vertices.
par.general.maxOptimals=Inf; % [integer > 0]: Maximum number of optimals to keep in ↘
        ↪ OptimalSolutions.
par.general.deterministic=false; % [logical]: Use Matlab random number generator, ↘
        ↪ or built-in deterministic generator?
par.general.FAbsTol=0; % [real vector > 0]: Absolute tolerances on fitness values.
```

`par.general.XLabels` and `par.general.FLabels` are cell arrays of strings, which contain the names of your parameters. These fields work exactly like their counterparts in Ferret (see Section 8.3). `par.general.XLabels={}` and `par.general.FLabels={}` indicate that generic names like $X1$, $X2$, etc. and $F1$, $F2$, etc. should be used.

`par.general.min` and `par.general.max` are the bounds of the box that SAMOSA will search, *if you want to run it as a bounded optimizer*. If you leave these fields empty (`par.general.min=[]` and `par.general.max=[]`), then SAMOSA will run as an unbounded optimizer. It is also possible to specify lower bounds but not upper bounds, or vice-versa. Moreover, you can use 'NaN' as a placeholder for unbounded parameters. For example, the setting

$$\begin{aligned} \texttt{par.general.min} &= [0;\ \texttt{NaN};\ 0;\ \texttt{NaN}]; \\ \texttt{par.general.max} &= [1;\ 1;\ \texttt{NaN};\ \texttt{NaN}]; \end{aligned} \tag{9.4}$$

indicates the following:

- Parameter 1 is bounded from 0 to 1.
- Parameter 2 has no minimum bound, but must not exceed 1.
- Parameter 3 has a minimum value of 0, but has no maximum bound.
- Parameter 4 is completely unbounded.

`par.general.X0` is the starting position for one vertex of the simplex. This can be left empty (`par.general.X0=[]`) if (and only if) `par.general.min` and `par.general.max` do not contain NaN values. SAMOSA will attempt to use the centre of your parameter space in this situation if `par.general.X0=[]`:

$$par.general.X0 = 0.5 * (par.general.min + par.general.max). \tag{9.5}$$

It is therefore crucial that either `par.general.X0`*, or both* `par.general.min` *and* `par.general.max` *are defined.* Otherwise, SAMOSA won't be able to obtain `par.general.X0` and won't know where to start, which will lead to an error message.

The initial state of the simplex is a random configuration of vertices about `par.general.X0`. `par.general.eps` is the magnitude of the scatter about the 'central' initial vertex $X0$ - specifically the standard deviation of the distance in each dimension. Any bounds that have been defined will be enforced during initialization.

As SAMOSA runs, it accumulates optimal solutions. Usually the number of solutions found is manageable, but you can limit them by setting `par.general.maxOptimals` to the maximum number that you want. `par.general.maxOptimals=Inf` by default.

Some of SAMOSA's operations are stochastic because it uses Pareto ranking for vertices and sometimes has to select randomly between vertices of equal rank. SAMOSA normally uses the built-in random number generator that is called by the 'rand' function. However, SAMOSA can alternatively use a simple 'deterministic' random number generator, for all stochastic functions, by setting `par.general.deterministic=true`. The effect of this setting is that you should obtain exactly the same run each time, as long as the initial conditions (i.e. $X0$) or other settings have not changed. This can be useful for debugging occasionally.

`par.general.FAbsTol` is used to set absolute fuzzy tolerances on fitness values and is similar to the `par.selection.FAbsTol` setting in the Ferret setup file (see Section 8.6). `par.selection.FAbsTol` is a real scalar, or a real vector of positive numbers whose length is equal to the number of parameters. *FAbsTol* is padded with zeros if it is a vector and its length is less than the number of parameters. Extra entries exceeding the number of parameters are truncated. Relative tolerances analogous to Ferret's `par.selection.FRelTol` parameter are not currently supported.

`par.general.FAbsTol` makes objectives 'fuzzy' and SAMOSA will not distinguish between vertices that differ in fitness by less than `par.general.FAbsTol` when computing vertex rank or adding vertices to the set of optimal solutions. FAbsTol is applied to all objectives for multi-objective problems. Running SAMOSA with `par.general.FAbsTol > 0` is somewhat like running Ferret in a mapping mode, with `par.selection.FAbsTol > 0`.

9.10.4 par.simplex

A simplex moves by a sequence of reflections, single-vertex expansions away from the centroid, single-vertex contractions toward the centroid, and multi-vertex contractions toward the best vertex. The `par.simplex` field gives a few options that control how the simplex explores the parameter space.

```
% Simplex
par.simplex.focusOnPolishing=false; % [logical]: true --> more thorough exploration. ↘
        ↪ false --> efficient polishing.
par.simplex.contractFactor=0.79; % [0 - 1]: % Factor to contract by on contraction ↘
        ↪ steps.
par.simplex.expandFactor=2; % [real > 1]: % Factor to expand by on expansion steps.
par.simplex.PKick=0.1; % [real > 1]: % Probability of a mutation-like kick. ↘
        ↪ Improves exploration.
```

`par.simplex.expandFactor` is a number greater than 1 that controls how much a vertex expands away from the centroid on an expansion step:

$$\mathbf{X}_{new} = \mathbf{X}_{centroid} + expandFactor * (\mathbf{X} - \mathbf{X}_{centroid}); \quad single - vertex\ expansion \tag{9.6}$$

Practically no expansion takes place when expandFactor approaches 1, and the expansion is greater as

expandFactor increases. The default setting is $expandFactor = 2$, which doubles the distance of a vertex from the centroid.

`par.simplex.contractFactor` is a number between 0 and 1 that controls the amount of contraction of a vertex on a contraction step:

$$\begin{aligned} \mathbf{X}_{new} &= \mathbf{X}_{centroid} + contractFactor * (\mathbf{X} - \mathbf{X}_{centroid}); \quad single - vertex\ contraction \\ \mathbf{X}_{new} &= \mathbf{X}_{best} + contractFactor * (\mathbf{X} - \mathbf{X}_{best}); \quad multi - vertex\ contraction \end{aligned} \tag{9.7}$$

Clearly, the simplex contracts all the way to $\mathbf{X}_{centroid}$ or $\mathbf{X}_{best}$ when $contractFactor = 0$, and no contraction takes place when $contractFactor = 1$. I find it useful to have the contractions relatively slow compared to the expansions, since it seems less likely that good solutions will be missed this way. I have the default set to 0.79, but this value is far from certain, and it is definitely a good control parameter to explore on your problem.

`par.simplex.PKick` is the probability that the simplex will receive a mutation-like kick to its non-optimal vertices. This operation is not done in a standard implementation of the simplex algorithm, but I find that occasional random kicks can help to bump the simplex out of a local minimum and improve the global search. Since it is a probability, `par.simplex.PKick` must be between 0 and 1. This 'kick' operator should be used sparingly because it effectively randomizes all vertices of the simplex except for the one with the best fitness, which can be disruptive to the search if overdone, and significantly increases the required number of evaluations of the fitness function. I prefer a value of `par.simplex.PKick=0.1`, which is set as the default. Some problems with many local minima may benefit from a higher setting, but easier searches may not require this feature at all.

`par.simplex.focusOnPolishing` tells SAMOSA which of two modes that it should use for multi-objective optimization:

- Standard mode (`par.simplex.focusOnPolishing=false`): The simplex will attempt to map out the Pareto front of your problem. You will probably see the simplex wandering around on the trade-off surface, accumulating points into *OptimalSolutions*, until it finally settles down to some point on the surface. This is a good mode for running SAMOSA as a standalone optimizer.

- Polish mode (`par.simplex.focusOnPolishing=true`): SAMOSA will 'target' the nearest point that it 'sees' on the Pareto front and zero in on it. The simplex will not wander the trade-off surface nor try to map it. This feature makes SAMOSA a very powerful polisher for multi-objective problems where the goal is to improve the quality of each solution on an already mapped-out trade-off surface. *Note that polish mode is configured automatically whenever SAMOSA is started as a polisher, and this over-rides any settings in the SAMOSA_setup file.*

I regard this ability to toggle between modes as the most innovative feature of SAMOSA. Finally, I note that the *focusOnPolishing* setting has no effect on single-objective problems.

9.10.5 par.stopping

Section 9.6.3 showed how to implement a customized stopping criterion for SAMOSA. However, the `par.stopping` section of the setup file gives the user access to some built-in options to stop a run.

```
% Stopping
par.stopping.maxNMoves=1000; % [integer > 0]: Maximum number of simplex moves.
par.stopping.maxFEval=NaN; % [Integer > 0]: Maximum number of function evaluations ↘
        (approximate).
par.stopping.XTol=[0;0]; % [real vector, length=number of parameters]: Stop if ↘
        ↪ simplex smaller than XTol in ALL dimensions.
par.stopping.FTol=0; % [real vector, length=number of objectives]: Stop if values ↘
        ↪ constrained within FTol in ALL objectives.
```

Without triggering any early-stop mechanism, a SAMOSA run will go for a maximum of `par.stopping.maxNMoves` moves. The default value of 1000 moves is reasonable for many problems, but this could be set higher or lower depending on the difficulty of the problem. You can also stop the run after a certain number of function evaluations by setting `par.stopping.maxFEval` to the desired number. Note that this number may not be exact; for example, if the final step is a multi-point contraction, then SAMOSA *must* evaluate all of the vertices that moved before shutting down. However, no new moves will be made once the number of solutions evaluated by the fitness function is greater than `par.stopping.maxFEval`, which occurs when SAMOSA detects that (`simplex.evalCount > par.stopping.maxFEval`).

The stopping tolerances `par.stopping.XTol` and `par.stopping.FTol` are represented in the simplex structure as `simplex.tol.X` and `simplex.tol.F`, and control the built-in stopping mechanism based on tolerances in X and F. Equations 9.2 and 9.3 give the exact stopping conditions.

9.10.6 par.interface

The `par.interface` section of the SAMOSA setup file contains options that affect how things are displayed in the SAMOSA window and on the MATLAB command line. Many of these settings are identical to their counterparts in the Ferret setup file, so this discussion will be brief.

```
% Graphics & messages
par.interface.verbose=1; % [integer: -1, 0, 1, or 2]: Verbose output?
par.interface.graphics=1; % [integer: 1 or -1]: Graphics on or off?
par.interface.plotStep=1; % [integer > 0]: Number of steps between plot updates.
par.interface.myColorMap='bone'; % [string: colormap name] User choice for colormap.
par.interface.myPlot=''; % [string]: Name of custom plot function.
par.interface.fontUnits='points'; % [string]: Must be 'points' or 'pixels'.
par.interface.titleFontSize=12; % [integer >= 1]: Self-explanatory.
par.interface.labelFontSize=10; % [integer >= 1]: Self-explanatory.
par.interface.axisFontSize=8; % [integer >= 1]: Self-explanatory.
par.interface.xAxis.type='X'; % [string: 'X' or 'F']: Default X-axis type
par.interface.xAxis.value=1; % [1, NPar] or [1,NObj]: Default X-axis variable
par.interface.yAxis.type='X'; % [string: 'X' or 'F']: Default Y-axis type
par.interface.yAxis.value=2; % [1, NPar] or [1,NObj]: Default Y-axis variable
par.interface.zAxis.type='F'; % [string: 'X' or 'F']: Default Z-axis type
par.interface.zAxis.value=1; % [1, NPar] or [1,NObj]: Default Z-axis variable
```

`par.interface.verbose` is an integer from 0 to 2 that determines how much output is sent to MATLAB's command line:

- `par.interface.verbose < 0`: Absolutely no output.
- `par.interface.verbose=0`: Almost no output.
- `par.interface.verbose=1`: (Default setting) SAMOSA reports each step on the fitness value(s) of its best vertex.
- `par.interface.verbose=2`: Like `par.interface.verbose=1`, except that SAMOSA also reports on the *type* of the last successful move that was accepted. You will see 'reflect', 'expand', 'contract1D', or 'contractMultiD' displayed after every successful step. This feature exists mainly because I think it's interesting to see what types of moves most often result in improvement.

Each move in a simplex algorithm requires a lot fewer fitness function evaluations than a generation in genetic algorithm. Therefore, it does not always make sense to display every move graphically. `par.interface.plotStep` is an integer greater than or equal to 1, which controls the number of moves between updates of the SAMOSA window. The interface is updated each move by default: `par.interface.plotStep=1`.

`par.interface.graphics` is an integer parameter that controls whether graphics are enabled or suppressed. Plotting is done by default (`par.interface.graphics=1`), but you can turn plotting off by setting this parameter to -1. These values are meant to be consistent with Ferret's `par.interface.graphics` setting (see Section 8.20), which also has an intermediate 'low-graphics' state `par.interface.graphics=0`, but SAMOSA does not use `par.interface.graphics=0`. If `par.interface.graphics=-1`, then you can close the SAMOSA interface and your project will continue to run on the MATLAB command line. Note that simply closing the SAMOSA window will not work when `par.interface.graphics=1` because the window will be re-initialized and plotting will resume!

The remaining fields in the SAMOSA setup file are the same as the corresponding fields in the Ferret setup file. Please refer to Section 4.7 for details.

Chapter 10

Anvil

The longer I work on an optimizer, the more it ends up looking like a genetic algorithm.

Anvil is Qubist's multi-objective simulated annealing/genetic algorithm hybrid optimizer, which has been part of the Qubist package since spring 2007. I originally intended it to be Qubist's main polisher for multi-objective problems, and it served this role quite nicely until SAMOSA took over as my preferred polisher in early 2009. Anvil is a *very* effective and thorough polisher, but I think its global optimization properties are sometimes a little too powerful for this role. Like Ferret, Anvil is a global optimizer that spends a lot of time exploring the parameter space and hunting for global minima - perhaps too much time for a 'polisher', where the goal is to *improve* on solutions that are already known to some degree of accuracy. However, using Anvil as a polisher remains a good choice if you want to try an alternative to SAMOSA. Anvil may require more time to run, but it will polish very effectively and might even extend the global exploration.

Anvil's niche in the set of Qubist optimizers is intermediate between the front line global solvers Ferret and Locust, and the (mostly) local optimizers SAMOSA and SemiGloSS. Anvil is the only optimizer in the Qubist package that can be used both as a polisher, and also as a standalone optimizer with good global search properties. Anvil isn't *nearly* as powerful as Ferret on tough problems, but it is a simple, snappy, lightweight code compared to Ferret's heavy machinery that may find good quality solutions more quickly. Anvil is also less powerful than Locust as a global optimizer, but simulated annealing is perhaps a better-understood technique than particle swarms that will be familiar to many users, and Locust can't be used for polishing. Anvil is much better as a standalone global optimizer than either SAMOSA or SemiGloSS. As a global optimizer, Anvil is well-suited for problems of easy to moderate difficulty, and harder problems where you need a good answer quickly, and care less about finding the absolute best answer that exists.

You should note that Anvil differs significantly from a typical simulated annealing code in the following ways:

- It is designed for both single-objective and multi-objective problems.
- It employs multiple 'tracks' that explore the parameter space independently, where each track represents a search point that moves through the parameter space by the usual simulated annealing

procedure of random perturbations and a temperature dependent acceptance criterion.

- Search points interact with each other via crossover and selection operators that are inspired by genetic algorithms. I've taken to referring to the set of search points as a 'population', which of course comes from the language of genetic algorithms.
- Anvil has its own critical parameter detection (CPD) system, which is inspired by Ferret's CPD system.
- Anvil uses an adaptive cooling schedule that helps to set the temperature dynamically, based on the distribution of energies in the population.

Anvil is predominantly a simulated annealing code, but its genetic algorithm-inspired components contribute significantly to its ability to search globally. Individual tracks operate by the rules of simulated annealing, but interact with each other through genetic algorithm-like operators. Multiple tracks help to cover the parameter space more thoroughly and help with the global search, while interactions between tracks lead to the same type of collective global behaviour that makes genetic algorithms so effective as global optimizers.

Anvil has evolved considerably over its two year existence. It started out as a pure simulated annealing code, intended mainly for polishing Ferret solutions, but I saw enough potential to expand it into a standalone optimizer. I later added multiple tracks plus crossover and selection operators when I wanted to improve its global search properties. I have often commented, partly in jest, to students of my Computational Physics course at the University of Manitoba that the longer I work on an optimizer, the more it ends up looking like a genetic algorithm. Nowhere is this more true than with Anvil.

10.1 Anvil as a Standalone Optimizer

10.1.1 Starting Anvil

Like the other Qubist optimizers, Anvil is usually started in standalone mode from the Qubist component selector shown in Figure 3.3. It can also be started using the 'startAnvil' function in the Qubist home directory, but this will only give you access to demos, and not your own projects. Starting Anvil will launch an interface like the one shown in Figure 10.1.

10.1.2 Anvil's Graphical Interface

The upper left graph represents a two-dimensional projection of the current 'population' of simulated annealing points (blue dots), and all known optimal solutions (pink dots). The projection can be changed using the 'X-Axis' and 'Y-Axis' menus.

The energy function plays the same role for a simulated annealing code as the fitness function does for a genetic algorithm. The energy function has about as much to do with the real physical energy of a system, as a genetic algorithm's fitness function has to do with the fitness of an imagined organism. Energy is just an indication of how good a particular solution is, and a useful analogy that fits with the simulated annealing paradigm. If you are trying Anvil on a problem that you originally set up for Ferret, then the fitness function that you designed for Ferret *is* your Anvil energy function, without any modifications.

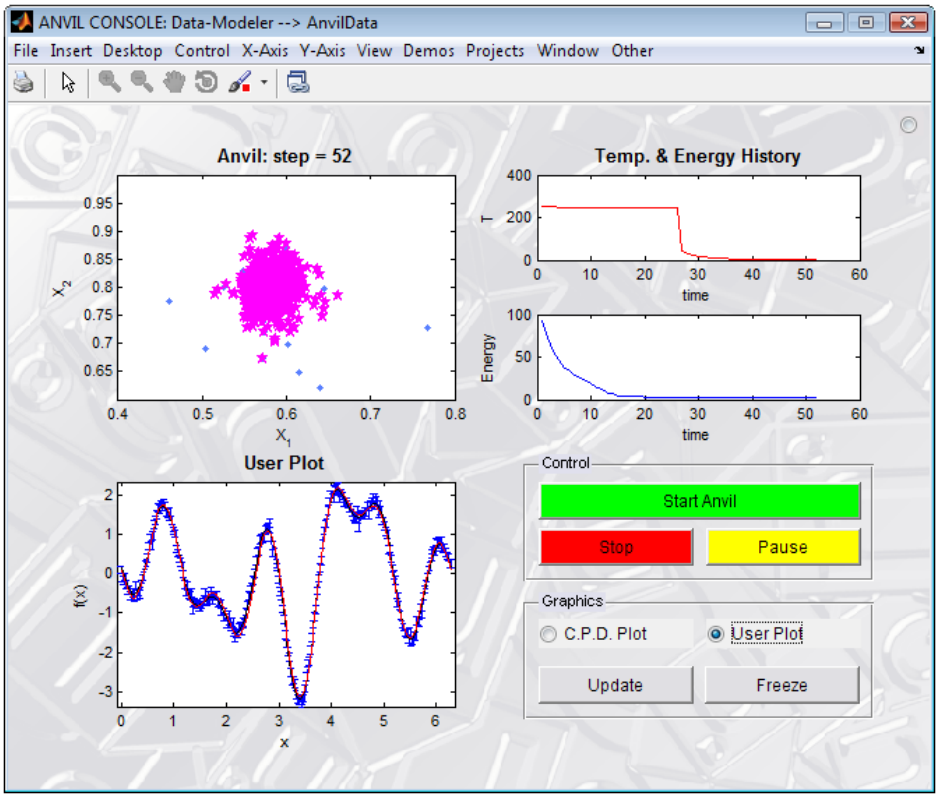

Figure 10.1: Anvil running the 'Data-Modeling/Data-Modeler' problem from the [Qubist_Home]/demos directory as a standalone optimizer.

Good solutions have low energy, in analogy to how a cooling metal 'searches' for it's lowest energy state. The lower right graph is the energy of the system as a function of time or steps. The top right graph represents the temperature of all simulated annealing tracks, which governs the probability that the simulated annealing algorithm will accept a 'bad' perturbation that increases the energy (see equation 2.3).

Anvil's interface shares many of the features of the Ferret window (see Figure 4.2), but is much simpler. Demos and projects are selected from the 'Demo' and 'Project' menus and are started by clicking the 'Start Anvil' button in the upper button group. The run can be stopped by clicking the 'Stop' button, or paused with the 'Pause' button. The view of the lower left plot can be toggled between a 'C.P.D. plot' or the 'User Plot' using the radio buttons in the lower button group. This button group also contains buttons labelled 'Update' and 'Freeze'. The Freeze button suppresses graphics temporarily, while Anvil continues to run on the command line, and and the button label should change to 'Un-Freeze' while Anvil's graphics are frozen. The Update button allows you to update the graphical display manually when the plots are frozen.

10.1.3 Setup File & Defaults

Anvil's default setup file (analogous to the defaultFerretSetup.m file; see Section 4.7 for details) is [Qubist_Home]/user/Anvil/defaults/defaultAnvilSetup.m, when running as a standalone optimizer. A full template of Anvil settings is located in [Qubist_Home]/user/Anvil/templates/AnvilSetup_template.m. This file should be copied to your project directory, renamed to AnvilSetup.m or another name of your choice, and modified to suit your problem. Anvil setup options are discussed in Section 10.9.

When run in standalone mode, Anvil's results are written to an OptimalSolutions.mat file in the project's data directory. The name of the data directory is determined by `par.history.dataDir` in the setup file, and is set to 'AnvilData' by default. The structure of an Anvil OptimalSolutions.mat file is discussed in Section 10.7. Section 10.8 shows how to run Anvil directly from the command line and receive results as a return argument.

10.2 Anvil as a Polisher

As a polisher, Anvil works very much like SAMOSA. You must specify `par.polish.optimizer='Anvil'` in your Ferret or Locust setup file, and Anvil will be called automatically when you press the 'Polish' button in the analysis window.

Note that Anvil uses a different default file when it is called as a polisher by Ferret or Locust. The default files for polishing are [Qubist_Home]/user/Ferret/defaults/defaultAnvilSetup.m and [Qubist_Home]/user/Locust/defaults/defaultAnvilSetup.m. It is not usually necessary to worry about the defaults when polishing because Ferret and Locust automatically modify them using their own setup files.

When called as a polisher, Anvil will generate a 'PolishedSolutions.mat' file alongside the OptimalSolutions.mat file that is produced by Ferret or Locust. The structure of a PolishedSolutions.mat file is exactly like an OptimalSolutions.mat file. Users rarely need to load these files though, because polisher results are automatically merged with the OptimalSolutions.mat file generated by analyzing the run from Ferret or Locust.

10.3 The init Function

Anvil's init function works exactly like Ferret's init function, which was explained in Section 4.6:

```
function extPar=init
```

Note that this is the standard technique that causes *extPar* to be passed to Anvil's fitness function as a local variable (see Section 9.5). Anvil can also use the 'global *extPar_*' technique discussed in Section 4.6.2. The same comments and warning from this section apply equally well to Anvil. This advanced technique is intended for odd situations where you need to share data between your fitness function and another function, and *can't* do it efficiently using the standard method. One example is for situations when you want to call Anvil from *inside* of a Ferret fitness function to optimize a sub-problem that must computed to evaluate the top-level Ferret fitness function. This situation is discussed in Section 4.6.2, and there is a contrived, but hopefully illustrative example in Qubist's demo directory called 'Advanced/Ferret-SAMOSA-Anvil-Insanity'. The bottom line is that it is OK to use this technique when you *really* need it, but you should use the standard method wherever possible.

10.4 The Energy/Fitness Function

It is worth noting that Anvil uses the terms 'energy function' and 'fitness function' interchangeably, and the figure axes are actually labelled as 'F' for 'fitness'. Some simulated annealing aficionados might object to this admittedly loose terminology, but I often use the term 'fitness' for two reasons:

- Anvil really is a simulated annealing/genetic algorithm hybrid, so either 'energy' or 'fitness' are about equally valid.
- Anvil is designed as a secondary optimizer/polisher, which reads fitness functions written primarily for Ferret and, to a lesser extent, Locust. It's just confusing to globally optimize the 'fitness' function with Ferret or Locust, and polish the 'energy' function with Anvil, when we're really talking about the same function!

Anvil's fitness function is less complicated than Ferret's fitness function, because Anvil does not use either the *saveData* or $XPhysMod$ return arguments (see Section 4.8). Note that it is simpler than even SAMOSA fitness function, which allows *saveData*, but ignores $XPhysMod$. The signature of the Anvil fitness function is therefore

```
[F, {auxOutput}]=fitness(X, {extPar})
%
% *** Braces {} indicate optional fields. ***
%
% Input fields:
% X          -->   [NGenes x N matrix]
% extPar     -->   MATLAB structure
%
% Output fields:
% F          -->   [NObj x N matrix] (required)
% auxOutput  -->   {N-element MATLAB cell array} or {empty cell array} (optional)
```

10.5 The Output Function and the Metal Data Structure

Anvil supports an optional output function, which works just like the output functions used by Ferret and SAMOSA (see Sections 4.9 and 9.6). Anvil's output function receives Anvil's top-level internal data structure, called *metal*, which is analogous to Ferret's *world* and SAMOSA's *simplex* structures. The signature of the Anvil output function is as follows:

```
function output(metal)
```

Anvil's output function is called once per timestep, where a timestep is defined such that all Anvil tracks accept or reject a perturbation by equation 2.3. Anvil's output function has no return arguments. However, it can be used to implement a custom stopping criterion, as discussed in Section 4.9.2.

10.5.1 The Metal Data Structure

The *metal* data structure is organized as follows:

```
metal
     |
     .X                % Matrix of parameter values.  Each column represents the
     |                 % current parameters for a simulated annealing track.
     |
     .F                % Matrix of fitness values.  Each column represents the
     |                 % current parameters for a simulated annealing track.
     |
     .XCPD             % Matrix of parameter values, including CPD information.
     |                 % 'NaN' indicates an unspecified parameter.
     |
     .auxOutput        % Auxiliary output.  See fitness function syntax.  auxOutput
     |                 % is empty {} when not used.  When used, each cell index
     |                 % corresponds to its respective columns in the X and F
     |                 % matrices.
     |
     .T                % Temperature
     |
     .metric           % Distance information used internally for a niching operator.
     |                 % Probably not useful to users.
     |
     .OptimalSolutions % Structure containing all known optimal solutions at the
     |                 % present timestep.
     |
     .graphics         % Graphics information used internally for Anvil plots.
     |        |
     |        .T       % Temperature history, as displayed in the Anvil window.
     |        |
     |        .F       % Best (minimum) fitness for each objective, as a function
     |                 % of timestep.
     |
     .par              % par structure, as loaded from the setup file.
         |
         .user.extPar  % External parameters from the user's init file.
```

The X, F, $XCPD$, and *auxOutput* fields are exactly like their counterparts in Ferret and require little further explanation. Note that Ferret's $XPhys$ field is absent because Anvil doesn't doesn't normalize parameter values internally; `metal.X` contains the 'physical' parameters that would correspond to `world.pop{`p`}.indiv.genome.XPhys` for population p in Ferret. The `metal.X` and `metal.XCPD` fields are the same, except that $XCPD$ may include NaN values as placeholders to indicate parameters that are not specified. If a parameter contains mostly NaN values in the *metal* structure or the final *OptimalSolutions* structure, then this means that the solutions are about equally fit, on average, when these parameters are replaced by random numbers within the parameter range. If you wish to examine parameters in your output function, then you almost certainly want to look at X, and not $XCPD$.

`metal.T` is the temperature at the present timestep, which is used to accept or reject a perturbation probabilistically by equation 2.3.

`metal.metric` contains distance information that is used internally by Anvil's niching operator. Niching

was discussed in Section 8.9 in the context of the Ferret genetic algorithm, and is unusual in a simulated annealing code. However, Anvil is part genetic algorithm, and its biological half uses a niching operator to maintain the diversity of the 'population' of simulated annealing tracks.

Unlike Ferret and Locust, (but like SAMOSA), Anvil carries a working set of optimal solutions around with it as it runs, and therefore does not require an analysis step at the end of the run (see Section 4.15). This greatly simplifies the code, because it does not save History files (see Section 4.16.4) as it runs, which makes Anvil more useful as a polisher. Optimals are accumulated in `metal.OptimalSolutions` and are always current, meaning that solutions that are no longer optimal are removed automatically at each time step.

The `metal.OptimalSolutions` structure contains the internal fields discussed in Section 10.7, which have the same meaning as their counterparts in the *metal* structure. Note that because Anvil does not save History files, Anvil runs can't be restarted after a crash the way that Ferret runs can. Anvil is really meant for smaller problems or relatively quick polishing runs, so the extra complexity required for robust crash recovery is probably not warranted.

10.5.2 Built-In Stopping Criteria

Anvil has several built-in criteria that can be used to stop the code. These are as follows:

- The maximum number of time steps allowed by the user: `par.stopping.maxTimeSteps`.
- The maximum allowed number of steps without improvement: `par.stopping.maxItNoImprovement`.
- Tolerance in parameters: `par.stopping.XTol`.
- Tolerance in fitness: `par.stopping.FTol`

These stopping criteria are discussed in Section 10.9.8, along with detailed advice on how to set up the built-in stopping options.

10.5.3 Custom Stopping Criterion

The most common use for the output function is to implement a custom stopping criterion. To stop Anvil from the output function, with all necessary cleanup operations performed automatically, you just need to call the Qubist function `abortQubist`, which automatically sets a global variable called $abortAnvil\_ = 1$. Further details of the `abortQubist` function are provided in Section 4.9.2. The following is a skeleton output function that will stop Anvil when the user-defined function `checkStop(metal)` returns a value of `true`:

```
function output(metal)

if checkStop(metal)
        abortQubist;
end
```

Of course, it is possible to stop Anvil by directly setting the global variable `abortAnvil_=1`. However, this is not usually recommended because the other optimizers in the Qubist package won't respond to $abortAnvil\_$. Unless this is the intended behaviour, you should stop Anvil by calling the `abortQubist` function instead.

Like SAMOSA, it is possible to call Anvil from *inside* a Ferret fitness function if you need to do an internal optimization of a sub-problem during the evaluation of your Ferret fitness function. In this case, a call to `abortQubist` in the inner optimization problem will stop Anvil without affecting the outer Ferret run. On the other hand, Ferret will respond to an `abortQubist` call in the outer problem, which will stop the entire run. Please refer to the demo titled 'Advanced/Ferret-SAMOSA-Anvil-Insanity' for an illustrative example of this technique.

10.6 The myPlot Function

Anvil supports a myPlot syntax that works the same as it does for the other Qubist optimizers:

```
function myPlot(X)
function myPlot(X,extPar)
function myPlot(X,F,extPar)
function myPlot(X,F,auxOutput,extPar)
function myPlot(X,F,auxOutput,rankPareto,extPar)
```

The myPlot function is called once per timeStep, and the User Plot appears in the lower left axes on the Anvil window, provided that 'User Plot' is selected in the 'Graphics' button group and the graphics have not been frozen using the 'Freeze' button.

10.7 The OptimalSolutions File

When run in standalone mode, Anvil writes an OptimalSolutions.mat file in the user's data directory (`par.history.dataDir`) at the end of a run. The name 'PolishedSolutions.mat' is instead used when Anvil is called from Ferret or Locust as a polisher, but the file is identical; a different name is used only to avoid conflicts with Ferret and Locust OptimalSolutions.mat files. The internal structure of an Anvil OptimalSolutions.mat file is as follows:

```
OptimalSolutions
                |
                |
                .X          % Matrix of parameter values for optimals.  Each column
                |           % represents a parameter set and the number of columns
                |           % is equal to the number of optimal solutions.
                |
                .XCPD       % Matrix of parameter values for optimals, including
                |           % CPD information.  Identical to X, except for NaN
                |           % values, which indicate an unspecified parameter.
                |
                .F          % Matrix of fitness values for optimals.  The number of
                |           % rows is equal to the number of objectives, and the number
                |           % of columns is equal to the number of optimal solutions.
                |
                .auxOutput  % Cell array of auxiliary output values for optimals.
                |           % When  used, each cell index corresponds to its
                |           % respective columns in the X and F matrices.

                |
                .metric     % Distance information - for internal use only.
                |
                .par        % par structure, as loaded from the setup file.
                    |
                    .user.extPar  % External parameters from the user's init file.
```

Each of these fields has the same meaning as its counterpart in the *metal* structure, except that only optimals are included. The optimal parameter values and fitness values are contained in `OptimalSolutions.X` and `OptimalSolutions.F` respectively. The *auxOutput* cell array contains a cell for each column of X or F if the fitness function returns an *auxOutput* field, but the cell array will be empty if *auxOutput* is not used. The *metric* field contains distance information that is used internally by Anvil for it's Ferret-like niching operator. It is unlikely that a user would need to use anything in this field. Finally, `OptimalSolutions.par` contains all of the configuration options loaded from the project's setup file. The external parameters loaded from the setup file are contained in `OptimalSolutions.par.user.extPar`.

10.8 Running Anvil From the Command Line

The following script is an example of how you might start Anvil programmatically, perhaps from inside one of your own functions or scripts. This plays the same role for Anvil as the custom launcher for Ferret that was discussed in Section 4.14.2, or the custom launcher for SAMOSA that was presented in Section 9.9. Here, we make slightly tricker assumptions than we made for SAMOSA and assume the following:

- QH is a string containing the global path to your Qubist home directory. On my Windows computer, this is 'C:\Users\JFiege\Documents\Qubist'.
- The project is one of the Qubist demos, called 'Data-Modeling/Data-Modeler-Small-PopSize'. This demo has all the bells and whistles of a more complicated project.

- This is really a Ferret project, so nobody has bothered to make an AnvilSetup file. We need to convert FerretSetup, and would like to do this automatically.

- The fitness function has the added complication that it contains a call to 'isAbortEval', which helps with parallel processing in Ferret and Locust. This function must be added to the path to avoid an error, even though Anvil doesn't do parallel processing.

- We don't want to run with graphics.

- We don't want to mess around loading files at the end of the run. We want the *metal* and *OptimalSolutions* structures as output parameters.

The following custom launcher satisfies these requirements. The code for this example can be found in the file [Qubist_Home]/user/Anvil/scripts/customLauncher.m.

```
function [metal, OptimalSolutions]=customLauncher

% Set the Qubist component.
global currentComponent_
currentComponent_='Anvil'; % Set the component.
%
% Reset the path:
path(pathdef);
%
% Set QH=Qubist Home directory.
QH='/Users/fiege/Documents/Qubist_Builder_PL/Qubist';
%
% Add Qubist paths:
addpath(QH);
setQubistPath(QH);
%
% Add the project directory.
addpath(genpath(fullfile(QH, 'demos', 'Data-Modeling', 'Data-Modeler')));
%
% Force Qubist abort status to 0 so that Anvil can run.
forceAbortQubist(0);
%
% Get default Ferret setup and modify it using the project's setup file.
par=defaultFerretSetup;
par=FerretSetup(par);
%
% Translate it to an Anvil setup file.
par=translateFerretToAnvil(par,defaultAnvilSetup);
%
% Turn off graphics.
par.interface.graphics=-1;
%
% Load the init file, and add it to par.
par.user.extPar=init;
%
% ------------------------------------------------------------------------
% Validate the fitness function.  Comment out these lines if to are running
% your project many times and are *sure* that your fitness function works.
validateFitnessFunction('Anvil', par);
%
% Abort if validation fails.
if getQubistAbortStatus; % Checks the abort status.
    disp('Stopping due to problems with the fitness function...');
    return
end
% ------------------------------------------------------------------------
%
% Start Anvil!
[metal, OptimalSolutions]=Anvil(par);
```

10.9 The AnvilSetup File

Anvil's setup file is much simpler than Ferret's, and perhaps only slightly more complex than SAMOSA's. This section discusses the options that are available and offers some advice on setting up Anvil for your projects. A template setup file is located in [Qubist_Home]/Qubist/user/Anvil/templates/AnvilSetup_template.m. Normally, this file should be copied to the project directory, renamed to AnvilSetup.m or another name of your choice, and modified to suit your project's needs.

10.9.1 par.user

The `par.user` branch of the Anvil setup file provides the name of the fitness function and the optional output function. `par.user` is similar to the corresponding branches of the Ferret and SAMOSA setup files.

```
% User
% par.user.fitnessFcn='fitness'; % [string]: Name of the fitness function.
% par.user.output=''; % [string]: Name of optional user-defined output function ↘
        ↪ called each timestep.
```

`par.user.fitnessFcn` is a string that sets the name of the energy or fitness function, whose syntax is outlined in Section 10.4. `par.user.output` is the name of the optional output function, as discussed in Section 10.5. Section 10.5.3 shows how to use Anvil's output function to implement a custom stopping criterion.

10.9.2 par.history

```
% Data Directory
par.history.dataDir='AnvilData'; % [string]: Directory for Anvil data files, etc.
```

`par.history` contains a single field called `par.history.dataDir`, which is a string that contains the name of the data directory. This is most often a local path, relative to the project directory, as shown here, but global paths are also allowed. Like SAMOSA, Anvil does not use history files, but retains the '`par.history`' name for consistency with Ferret and Locust. Your OptimalSolutions.mat file will be written to this directory at the end of the run.

10.9.3 par.general

`par.general` provides basic information about the problem that Anvil is intended to solve. All fields have the same meaning as in Ferret. Please refer to Section 8.3 for further information.

```
par.general.min=[-1, -1]; % [real vector]: Minimum parameter values.
par.general.max=[1, 1]; % [real vector]: Maximum parameter values.
par.general.cyclic=[]; % [integer vector > 1]: Which parameters are cyclic?
par.general.XLabels=; % [Cell array of strings]: Give names to some or all ↘
        ↪ parameters: 'A','B',...
par.general.FLabels=; % [Cell array of strings]: Give names to some or all fitness ↘
        ↪ values: 'FA','FB',...
par.general.HolderMetricExponent=2; % [real]: The exponent used for the metrics.
```

10.9.4 par.anneal

The `par.anneal` section of Anvil's setup file sets important parameters that affect Anvil's cooling schedule and the initial size of perturbations.

```
% Annealing constants
par.anneal.T0=250; % [real > 0]: Starting temp.
par.anneal.TMin=0; % [real >= 0]: Min temp.
par.anneal.TCool=1000; % [0 - 1]: Cooling timescale.
par.anneal.Tf=0.00; % [real >= 0]: Temp for exit.
par.anneal.PReheat=0.01; % [0 - 1]: Probability of re-heating.
par.anneal.sigma=0.1; % [real > 0]: Initial perturbation size.
```

`par.annealing.T0` is the initial temperature of the simulated annealing tracks. The cooling schedule is exponential, but is modified adaptively by an algorithm that monitors the distribution of energy (or fitness) values at each timestep and attempts to set the temperature to a value that improves convergence while also ensuring adequate exploration of the parameter space. If we ignore the adaptive cooling algorithm for a moment, then the cooling law is given by the following formula:

$$T_{i+1} = T_{min} + \left(1 - \frac{1}{T_{cool}}\right)(T_i - T_{min})\,. \tag{10.1}$$

such that tracks cool to a minimum temperature of `par.anneal.TMin`, over a timescale of approximately `par.anneal.TCool`. The run will will stop if the temperature reaches the final temperature `par.anneal.Tf`. In the event that the fitness stops improving before temperature Tf is reached, the system has the opportunity to 're-heat' with probability `par.anneal.PReheat` each time step. This is done to hopefully bump the simulated annealing tracks out of a presumed local minimum so that the run can continue to make progress. The $PReheat$ probability must be a real number between 0 and 1, and should be set to a value such that

$$par.anneal.PReheat \times par.anneal.TCool > 1 \tag{10.2}$$

for the reheating strategy to be effective. Otherwise, cooling will dominate, and re-heating may not be able to ramp up the temperature long enough to bump the system out of a local minimum.

The actual thermal evolution of an Anvil run is more complex than equation 10.1 would suggest because this equation ignores adaptive changes to the temperature. Adaptive temperature control can, in fact, dominate the thermal evolution, especially near the beginning of a run. This occurs automatically and is affected by a single parameter `par.tracks.window`, which is discussed in Section 10.9.5 below.

The final parameter in this section is `par.anneal.sigma`, which is the size scale of the simulated annealing perturbations, relative to the range of parameters. For example, if the search range of a parameter i is

given by $range(i) = par.general.max(i) - par.general.min(i)$, then perturbations $\Delta\mathbf{x}$ will have a size of approximately $\Delta\mathbf{x}_i \approx par.anneal.sigma \times range(i)$ at the beginning of the run. *Note however that Anvil automatically scales the size of the perturbations during the run to maximize the number of accepted moves.* Therefore, this control parameter is only likely to affect the search during about the first `par.tracks.window` timesteps of a run.

10.9.5 par.tracks

An Anvil run always uses multiple simulated annealing tracks, which sample the parameter space more thoroughly than the single track used by a traditional simulated annealing code. Multiple tracks are also used by Anvil for adaptive temperature control and for auto-scaling perturbations, as discussed in Section 10.9.4.

```
% Tracks
par.tracks.N=100; % [integer]: The number of Anvil tracks.
par.tracks.window=25; % [integer]: Anvil keeps the last par.tracks.window steps in ↘
        ↪ memory.
% Affects adaptive cooling, dynamic perturbation size, and stopping criteria.
```

The number of concurrent tracks is equal to `par.tracks.N`, which must be an integer. The default value is `par.tracks.N=100`, and I do not recommend values below about 10.

Anvil maintains a record of recent successful moves in memory, and this information is used for adaptive temperature control and to dynamically scale the size of the perturbations. The number of successful moves retained in this buffer is given by `par.tracks.window`. The default setting is `par.tracks.window=25`, and must be greater than zero. A slightly smaller value will make the adaptive temperature control and dynamic perturbation scaling more reactive, but it will likely become sporadic if this parameter is too small, so I do not recommend values less than about 10. Larger values make temperature control and dynamic perturbation scaling less responsive so that changes happen more slowly.

10.9.6 par.XOver

Anvil features a genetic algorithm-like crossover operator that allows simulated annealing search points to communicate and cooperate as they search the parameter space. The `par.XOver` field controls Anvil's genetic algorithm-like crossover operator.

```
% Crossover: A GA-like crossover operator.
par.XOver.PXOver=1; % [0 - 1]: Probability of crossover between Anvil tracks.
par.XOver.strength=0.5; % [0 - 1]: The strength of the crossover operator.
par.XOver.dispersion=0.1; % [0 - 1]: Randomization during crossover.
```

Crossovers occur between randomly chosen tracks each time step with a probability equal to `par.XOver.PXOver`. The default value is `par.XOver.PXOver=1`, which is fairly common for a genetic algorithm.

The crossover operator is implemented like Ferret's real-valued crossover operator (see Section 8.8), in which solutions move toward each other on a straight line joining them in the parameter space. The fraction of the distance moved follows a Gaussian random distribution with a standard deviation equal to `par.XOver.strength`. Thus, the default setting of `par.XOver.strength=0.5` means that a track engaging

in a crossover with another typically moves about halfway on a line joining them. The exact value is not critical.

A 'dispersion' is added in a random direction with a magnitude equal to $par.XOver.dispersion \times |\delta \mathbf{x}|$, where $|\delta \mathbf{x}|$ is the magnitude of the distance moved. A similar dispersion step is used in Ferret's crossover operator, as discussed in Section 8.8. Anvil's dispersion operator plays a role similar to a mutation operator in a genetic algorithm, which helps to bump solutions out of local minima and improve the exploration of the parameter space.

10.9.7 par.selection

The `par.selection` field controls Anvil's genetic algorithm selection operator, which is based on a binary tournament, like Ferret. The selection operator causes simulated annealing search points to compete as they collectively search the parameter space. A search point that loses a competition is deleted from the search and replaced with a copy of the winner, and continues to search from that point. Thus, the selection operator gradually concentrates the simulated annealing tracks in the best region found by *any* track, which is likely to be in the vicinity of the global minimum.

```
% Selection
par.selection.PSelect=1; % [0 - 1]: Selection pressure for a GA-like selection ↘
        ↪ operator.
par.selection.FAbsTol=0.1; % [real >= 0]: Absolute fitness range (+/- FAbsTol) to ↘
        ↪ use as a fuzzy fitness band.
par.selection.FRelTol=0; % [real >= 0]: Relative fitness range (+/- FAbsTol) to use ↘
        ↪ as a fuzzy fitness band.
```

`par.selection.PSelect` is the probability that a track will enter into a tournament each timestep. The default value is $PSelect = 1$, which is common for a genetic algorithm.

`par.selection.FAbsTol` and `par.selection.FRelTol` work exactly like the corresponding 'fuzziness' operators in Ferret - differences in fitness values less than the absolute or relative tolerance are ignored in competitions (see Section 8.6). Like Ferret, Anvil will attempt to map out any region of parameter space that falls within the fitness tolerance of the minimum value discovered during the search.

10.9.8 par.stopping

Anvil contains several built in stopping criteria that may be useful for some problems. These are set using the `par.stopping` field of the setup file.

```
% Stopping criteria
%
par.stopping.maxTimeSteps=100; % [integer > 0]: Maximum number of time steps
par.stopping.maxItNoImprovement=25; % [integer]: % Maximum number of steps with no ↘
        ↪ improvement.
%
% Tolerance on X and F - mainly for single-objective problems.
par.stopping.XTol=0; % [real > 0]: Tolerance on the solution.
par.stopping.FTol=0; % [real > 0]: Tolerance on the energy.
par.stopping.boolean=@and; % [boolean operator]: Should be '@and' or '@or'.
%
% Max number of solutions allowed in final optimal set.
par.stopping.maxNOptimals=1000; % [integer]: Max number of optimals.
```

Anvil is stopped if the number of steps taken reaches the maximum allowed number of steps `par.stopping.maxTimeSteps`. Anvil can also monitor its progress and be set to abort if there is no improvement in the solutions, over all tracks, for a certain number of generations. Specifically, Anvil will exit if no track has improved after `par.stopping.maxItNoImprovement` timesteps. This parameter has a default value of 25, should be set to an integer greater than zero, and works for both single and multi-objective problems.

Exit criteria based on tolerances are also available, which are mainly used for single-objective problems:

- The standard deviation of all parameter values, over the set of all tracks, falls below par.stopping.XTol.
- The standard deviation of all fitness values, over the set of all tracks, falls below par.stopping.FTol.

These two criteria are combined with a boolean operator: `par.stopping.boolean=@and`, if *both* must be true to exit, or `par.stopping.boolean=@or`, if Anvil should exit when *either* criterion is true. Note the function handle syntax of these fields - the '@' symbol is required. Because these exit criteria are based on standard deviations, they require a sufficiently large number of tracks `par.tracks.N` (see Section 10.9.5) to get a reasonable standard deviation estimate. I would worry about using these criteria if the number of tracks is fewer than about 10.

Finally, `par.stopping.maxNOptimals` is the maximum number of solutions allowed in the final optimal set. Note that unlike Ferret and Locust, Anvil does not require an analysis step (see Section 4.15) and therefore must carry its optimal solution set around with it during the run. Moreover, Anvil uses a niching technique (see Section 10.9.9) to spread solutions evenly over the optimal set, and this requires a metric to be computed over the number of optimals. A large number of optmals can degrade performance, so Anvil allows you to limit the size of the optimal set. `par.stopping.maxNOptimals=1000` by default, but you should consider reducing it if your run seems to slow down as it progresses.

10.9.9 par.niching

Anvil's selection operator employs a simplified form of niching (see Sections 5.1 and 8.9) to help keep tracks spread out over the optimal set. Like Ferret's niching operator, this allows Anvil to map solution sets for problems with optimal sets that don't reduce to a single point. These problems are typically

multi-objective problems, or single-objective problems where a fuzzy tolerance has been defined (`par.selection.FAbsTol > 0` or `par.selection.FRelTol > 0`).

```
% Niching
par.niching.priority='XF'; % [string: re-order 2 letters, 'X', or 'F']: Priority of ↘
        ↪ niching. If empty, use Pareto niching.
par.niching.X=0.25;  % [0:1]: X-Niching: Typically ~0.25 is about right.
par.niching.F=0;  % [0:1]: F-Niching: Typically ~0.25 is about right.
par.niching.exponent=2; % [real: usually > 0 & <~ 2]: Used in the niche function.
```

Anvil's niching parameters work exactly like the settings that control Ferret's niching system, except that there is no 'pattern niching' in Anvil, so Ferret's 'P' option in `par.niching.priority` has no counterpart. Therefore, the `par.niching.priority` setting must be either `par.niching.priority='XF'` for X priority, `par.niching.priority='FX'` for F priority, or `par.niching.priority=''` for no priority, which implies Pareto niching. See Section 8.9 for details.

10.9.10 par.CPD

Anvil has a simple critical parameter detection (CPD) system that works something like Ferret's CPD system. The goal of CPD is the same in both codes: to find parameters that have little effect on the solutions, when such parameters exist, and to use this information to effectively reduce the dimensionality of the problem.

```
% Critical Parameter Detection
par.CPD.PDeactivatePar=0; % [0:1]: Probability of parameter deactivation.
par.CPD.PReactivatePar=0; % [0:1]: Probability of parameter reactivation.
```

`par.CPD.PDeactivatePar` and `par.CPD.PReactivatePar` work exactly like Ferret's `par.CPD.PDeactivateGenes` and `par.CPD.PReactivateGenes`. Please refer to Sections 2.2.6 and 8.10 for details.

10.9.11 par.interface

The `par.interface` settings are exactly like those used by SAMOSA. Please refer to Section 9.10.6 for details.

```
% Graphics & messages
par.interface.graphics=1; % [integer: 1 or -1]: Graphics on or off?
par.interface.verbose=1; % [logical]: Verbose output for warnings, etc.?
par.interface.plotStep=1; % [integer >= 1]: How often to update graphics?
par.interface.myPlot=''; % [string]: Name of custom plot function.
par.interface.fontUnits='points'; % [string]: Must be 'points' or 'pixels'.
par.interface.titleFontSize=10; % [integer]: Self-explanatory.
par.interface.labelFontSize=8; % [integer]: Self-explanatory.
par.interface.axisFontSize=8; % [integer]: Self-explanatory.
par.interface.xAxis.type='X'; % [string: 'X' or 'F']: Default X-axis type (X or F)
par.interface.xAxis.value=1; % [integer >= 1]: Default X-axis variable (1-NPar or NObj)
par.interface.yAxis.type='X'; % [string: 'X' or 'F']: Default Y-axis type (X or F)
par.interface.yAxis.value=2; % [integer >= 1]: Default Y-axis variable (1-NPar or NObj)
par.interface.zAxis.type='F'; % [string: 'X' or 'F']: Default Y-axis type (X or F)
par.interface.zAxis.value=1; % [integer >= 1]:Default Y-axis variable (1-NPar or NObj)
```

This concludes our discussion of Qubist's secondary optimizers, which are most often used as polishers for Ferret's optimal solution sets. If you have followed this guide up to this point, you will have a complete understanding of how to run Ferret and polish your results. We will now turn to Locust, the second front line optimizer in the Qubist package. Using Locust and polishing results is similar to running Ferret. Therefore, the chapter on Locust will be brief.

Chapter 11

Locust

'We hope that, when the insects take over the world, they will remember with gratitude how we took them along on all our picnics.'

-Bill Vaughan

11.1 Introduction

Locust is Qubist's multi-objective particle swarm optimizer (PSO). It is much newer than Ferret and faster on many problems, but perhaps less well-suited for the most difficult problems. Still, Locust is a very powerful code that I regard as a front line optimizer of the Qubist package. It is an excellent alternative to Ferret and ideal for problems of moderate to high difficulty. Locust shares many features with Ferret, including built-in parallel computing capabilities (Chapter 6 and Section 8.5), analysis techniques (Section 4.15), crash recovery/resume features (Sections 4.16.1 and 4.17), History files (Section 4.16.4), a visualization interface (Section 5.2), and the ability to call SAMOSA, Anvil, SemiGloSS, and fminsearch as polishers (Sections 4.16 and 8.18). In some ways, Locust *is* Ferret, since I borrowed many of its features directly from Ferret during development. Many of Locust's features will therefore look familiar if you have already tried Ferret because the two optimizers share a great deal of code.

11.1.1 History

I started working (rather casually) on PSOs during spring 2007, and was immediately struck by the collective behaviour that emerged from the dynamics of rather simple interacting particles. I was also quite thrilled to see that even my first simple PSO worked quite nicely, albeit on fairly simple test problems. I have always enjoyed dynamics as an astronomer and a physicist, and I believe that my academic background gave me some insight into how this fascinating new tool worked. Locust developed consistently for the next two years, even while I put the bulk of my effort into perfecting Ferret. It became a reasonably powerful multi-objective code during this time, and inherited Ferret's interfaces for visualization and polishing.

Between 2007 and spring 2009, Locust's development went down one road that I came to regret. As I commented at the top of Chapter 10, my optimizers often start to look like genetic algorithms if I work on them for long enough. This is quite natural - I've been developing Ferret almost obsessively for about seven years at the time of this writing. I think about optimization most naturally in terms of evolutionary computing, and I've learned many genetic algorithm tricks during this time. The paradigm of evolution provides an especially rich and flexible framework for optimization, and this flexibility admits the possibility of novel hybridization schemes with other types of optimization methods.

Sometimes, hybridization with a genetic algorithm is successful, as is the case of Anvil. Sometimes it's not so useful, as was the case of Locust. Locust-1 had a solid foundation in particle swarm dynamics, but this was mixed with genetic algorithm components, and frankly I did not really understand how the evolutionary parts interacted with the particle swarm dynamics. When I first sat down to write this chapter in July 2009, I spent the better part of a month testing and critiquing Locust and trying to sort out the useful parts from the junk. In the end, I determined that the particle swarm parts were well designed and effective, but *none* of the genetic algorithm components were helping at all. I removed all of the genetic algorithm parts, which caused the code to shrink by about a third. During my testing, I also made some very significant enhancements to the swarm dynamics, which turned Locust into a much more powerful tool than previous versions. This encouraged me to add Ferret's crash recovery and parallel computing features, and once I had connected everything, the newly streamlined Locust warranted an updated version number and a more prominent role in the Qubist package. Locust-2 emerged as an exciting new optimizer suitable for use as the primary optimizer for many problems.

Locust-2 has already been through quite a lot of testing, and its performance characteristics complement Ferret perfectly. Locust can do almost all of the test problems that come with Qubist, except for the ones that focus heavily on linkage-learning. Generally, it finds solutions much more quickly than Ferret, although it does not map extended optimal solutions quite as thoroughly. The same conclusion seems to hold true for two difficult 'real-world' problems in astrophysics that graduate students at the University of Manitoba have tried using both Ferret and Locust. You should definitely try Locust if you care more about speed than mapping an extended optimal or trade-off surface in great detail!

11.1.2 PSO Basics

There are excellent books on PSOs (e.g. Eberhart [2001]) and additional resources can be found readily in journals and on the web. I will only briefly outline the basic ideas of PSOs here. PSOs are similar to genetic algorithms in that they sample many points in the search space simultaneously. Where a genetic algorithm has a *population* of *individuals* exploring the search space at any given *generation*, a PSO has a *swarm* of *particles* moving through the search space at any given *time step*. The dynamics of a simple PSO are surprisingly simple. Each particle in the swarm is simultaneously attracted to its own 'personal best' solution - the best solution that the particle has personally seen - and the 'global best' solution - the best solution that the entire swarm has ever seen. The law of attraction follows a simple spring law: $F \propto |\Delta \mathbf{x}|$, where $|\Delta \mathbf{x}|$ is the distance between a given particle and either the personal best solution $\mathbf{x}_p$ or the global best $\mathbf{x}_g$. Assuming that the force and velocity are approximately constant over a time step, the new velocity and position of particle i after a time step Δt are given by

$$\begin{aligned} \mathbf{v}_i(t + \Delta t) &= \mathbf{v}_i(t) + [c_p \xi_p (\mathbf{x}_p - \mathbf{x}_i) + c_g \xi_g (\mathbf{x}_g - \mathbf{x}_i)] \, \Delta t \\ \mathbf{x}_i(t + \Delta t) &= \mathbf{x}_i(t) + \mathbf{v}_i(t) \Delta t, \end{aligned} \tag{11.1}$$

where c_p and c_g play the role of spring constants for the personal and global best solutions respectively. Some randomness is injected via the uniform random variables ξ_p and ξ_g, which are typically drawn from

the range 0 to 1. The stochastic terms play a role similar to the mutation operator in a genetic algorithm; they add randomness to the search, which helps the particles to explore previously unexplored parts of the parameter space.

Equations 11.1 amount to a simple Euler integration scheme for a dynamical system of equations that move each particle every time step. The roles of the personal and global best solutions are clear; the personal best solution represents a particle's memory of the best region of parameter space that it has seen, and the global best solution represents the entire swarm's collective memory. In effect, the global best solution ties all of the particles together to encourage collective behaviour.

Particle swarm optimization is a young and rapidly changing field of research that still has many open questions, which are discussed in a recent review by Poli, Kennedy & Blackwell [2007] in the inaugural edition of *Swarm Intelligence*. There are different methods to initialize the swarm velocities and deal with particles that leave the boundaries of the search. Equation 11.1 is perhaps the simplest set of swarm equations, but there are also many different implementations possible, which strive to balance thorough exploration of the parameter space against the need to exploit high performance regions when they are found. Most PSO implementations include a damping term in equations 11.1, which decreases the velocity magnitude in a time t_{damp} to help the swarm settle down as it zeros in on the optimal region. This damping term is in some ways analogous to the cooling of a simulated annealing code, although there is no analogue in a genetic algorithm. It is also common to limit the velocity to prevent runaway growth. Locust uses damping, but does not require a velocity limit. Locust uses an exact solution to the swarm equations, rather than the Euler integration scheme in equation 11.1, as discussed in Section 11.1.3. As a multi-objective optimizer, it also requires some non-standard techniques designed to fill out the Pareto surface, which are also discussed in Section 11.1.3.

11.1.3 Locust as an Enhanced Multi-Objective PSO

In physics, a simple harmonic oscillator (SHO) potential is a potential energy function of the form

$$U = \frac{1}{2}k|\mathbf{x}|^2, \tag{11.2}$$

where $k > 0$ is a spring constant and $|\mathbf{x}|$ is the distance from the centre of attraction at the origin. It is easily shown that this potential results in forces that are consistent with Hooke's law for springs $\mathbf{F} = -k\mathbf{x}$, and that the exact solution for a particle moving in a SHO potential (in a space of arbitrary dimensionality greater than 2) is a closed orbit within a plane. The equations governing this motion can be solved exactly to give position and velocity as a function of time, and it is easy to generalize this result to the case that there are two centres of attraction rather than one. Locust does not really use the approximate SHO solution given by equation 11.1. Rather, it solves the exact analytical trajectories traced by equation 11.1 in the limit $\Delta t \to 0$, with a damping term included. Solving these equations exactly requires some effort, but my numerical experiments show that the exact solution results in more stable and reliable PSO. I believe that this is because the exact solution eliminates the build-up of errors in the orbits, which would result from applying equation 11.1 directly with a finite Δt. The exact solution is of course slightly more costly to evaluate than the Euler approximation, but this extra computational expense is insignificant for any realistic problem, where the computational time is normally dominated by the evaluation of the fitness function.

Choosing $\mathbf{x}_p$ is straightforward because it represents the personal best solution (or *pbest* solution) that any particle has encountered in it's travels. Thus, one simply keeps track of the position of the lowest value of the fitness function $F(\mathbf{x})$, following Ferret's convention that low values of F correspond to more desirable

solutions. The most common particle swarm implementation is the version discussed in Section 11.1.2, where the global best solution (or *gbest* solution) $\mathbf{x}_g$ is evaluated over the entire swarm. This swarm topology can be thought of as a fully connected graph, where the entire swarm of particles 'talk' to each other via the *gbest* solution. It is easy to imagine other swarm topologies, where the network of communication between swarm members is less densely connected, so that each particle only talks to a few other particles in its neighbourhood. In this case, different particles may have different *gbest* solutions, depending on the best solutions that have been discovered by its neighbours. This scenario is referred to as a static *lbest* (for 'local best') topology when the network connecting particles is static such that it does not change throughout the run. Swarms based on sparsely connected networks can be thought of as being divided into sub-swarms, where each sub-swarm shares a common *gbest* solution. Such a topology is able to avoid local minima better because the sub-swarms essentially explore the space in parallel. On the other hand, the fully connected *gbest* topology is best for exploiting a single isolated solution late in a run, because it focusses the efforts of the entire swarm on the region of parameter space in the vicinity of the *gbest* solution.

The standard *gbest* approach would not make much sense for a multi-objective particle swarm algorithm like Locust, which aims to populate a spatially extended Pareto front with a set of optimal solutions that are equally good. Extended solutions sets are also possible when a fuzzy tolerance is specified for a single objective problem, which often represents the χ^2 error tolerance of a data-modeling problems. In such cases, a single global best solution would put too much emphasis on a single point within the set of optimal solutions, and the mapping of the optimal set would be extremely poor. Locust uses a dynamic swarm topology that is similar to the *lbest* topology, except that the proximity of particles in Euclidean space is used to define the initial neighbourhoods, and these neighbourhoods evolve dynamically throughout the run. Neighbourhoods merge and divide as required to map out the extended structure of the optimal set.

The transition from a fully connected *gbest* algorithm to an *lbest*-style algorithm with a dynamic topology is the main enhancement that resulted in the development of Locust-2 during the summer of 2009. The resulting code is much more powerful and versatile than its predecessor, and works equally well on single and multi-objective problems. The underlying dynamic swarm topology is quite different from other topologies discussed in the literature, and has the benefit that it essentially self-optimizes. A large number of neighbourhoods will be preserved to map a spatially extended solution set, but the swarm topology will correctly collapse to a single neighbourhood late in a run if only a single solution exists. I view this technique as the optimal balance between exploration and exploitation: the parallel action of many sub-swarms evade local minima early in the run for all problems, and are retained to the end when the focus is on mapping an extended solution set, but swarms reduce to the maximally exploiting *gbest* algorithm late in the run for problems where only a single best solution exists.

11.2 Running Locust from the Graphical User Interface

Locust is usually started from the Qubist component selector shown in Figure 3.3. It can also be started using the 'startLocust' function in the Qubist home directory, but this will only give you access to demos, and not your own projects. The Locust interface is shown in Figure 11.1. Like the other optimizers, demos and projects are selected from the menus labelled 'Demos' and 'Projects'. User-defined projects are added in the launchQubist.m file, following the procedure outlined in Section 4.4. Once a project has been loaded, it can be run by clicking the 'Start Locust' button.

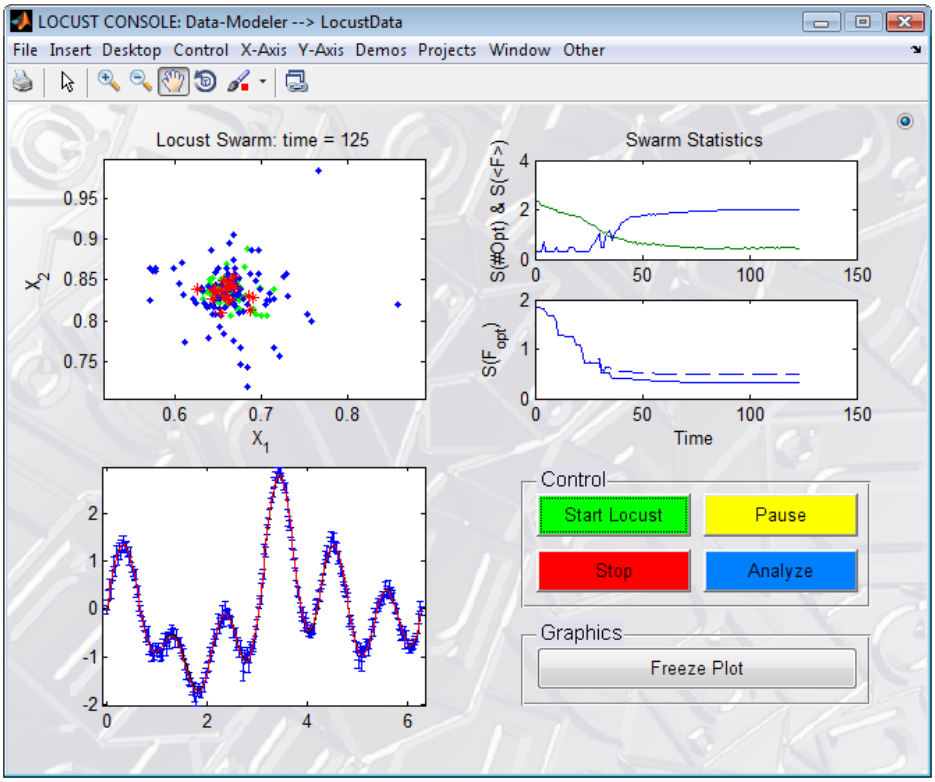

Figure 11.1: Locust running the 'Data-Modeling/Data-Modeler' problem from the [Qubist_Home]/demos directory.

11.2.1 Locust Figures

The Locust user interface shows a few graphs that update while the project is running. Locust's interface is much simpler than the Ferret console (Figure 4.2), but still quite informative. I prefer to run Locust through the interface, rather than on the command line (see Section 11.3) because the graphs provide real-time information about how well the code is converging, and can prevent a lot of wasted time if things go wrong. All figures update each time step, unless the graphics are frozen by clicking the 'Freeze Plot' button. This bypasses all graphics to allow Locust to run faster, even though this usually provides only a marginal improvement, except for simple problems where the fitness function evaluates much more rapidly than the graphics calls. When clicked, the 'Freeze Plot' button changes to 'Unfreeze Plot'; clicking this button again turns the graphical system back on.

The upper left graph represents a two-dimensional projection of the swarm at the current time step, shown with blue dots. 'Personal best' solutions are shown as green dots, and 'global best' solutions are shown as red asterisks. The projection can be changed using the 'X-Axis' and 'Y-Axis' menus.

The upper right pair of graphs provide convergence information for the run. The blue line in the top plot indicates how many swarm members are within the optimal set, as a function of time step. The green line indicates the median fitness of the swarm for single-objective problems, or the median rank of swarm

members for multi-objective problems. *Recall that Locust, like all of the Qubist optimizers, is a minimizer, and lower fitness values correspond to better solutions.* The lower plot shows the minimum fitness value (solid line) and maximum fitness value (dashed line) for members of the optimal set as a function of time step, for each objective function. Different objective functions are cycled through different colours, using MATLAB's usual colour order for plotting (type 'help plot' on the MATLAB command line for details). All functions are 'squashed' using the squashing transformation given by equation 9.1, and the motivation for using this function are discussed following the equation. During a Locust run, you should see the fitness values and the median rank decrease. For multi-objective problems and fuzzy single-objective problems, the number of solutions in the optimal set should increase until it levels off once the swarm is mostly within the optimal set.

The lower left plot is the User Plot. Any graphical commands given called by your 'myPlot' function are sent directly to this window. Please refer to Section 11.9 for details.

11.3 Running Locust from the Command Line

Locust can be run from the command line without graphics, but a custom launch file is required. This section presents a custom launcher for Locust where I assume the following:

- QH is a string containing the global path to your Qubist home directory.
- The project is one of the Qubist demos, called 'Data-Modeling/Data-Modeler-Small-PopSize'. This demo has all the bells and whistles of a more complicated project.
- The fitness function has the added complication that it contains a call to 'isAbortEval', which helps with parallel processing in Ferret and Locust. See Section 6.4.4 for details.
- We plan to run this project in the background, so we don't want any graphics.

```
function customLauncher
% Modify this script if you need to make a custom launcher for Locust that runs ↘
        ↪ from the command line.

% Set the Qubist component.
global currentComponent_
currentComponent_='Locust'; % Set the component.
%
% Reset the path:
path(pathdef);
%
% Set QH=Qubist Home directory.
QH='/Users/fiege/Documents/Qubist_Builder_PL/Qubist';
%
% Give the project path and setup file name.
currentProjectPath_=[QH, '/demos/Data-Modeling/Data-Modeler'];
%
% Add Qubist paths:
addpath(QH);
setQubistPath(QH);
%
% Add the project directory.
addpath(currentProjectPath_);
%
% Force Qubist abort status to 0 so that Locust can run.
forceAbortQubist(0);
%
% Add the project path.
addpath(genpath(currentProjectPath_));
%
% Parallelization info.
setGUIDataRoot('parallel', false);
launchDir=fileparts(which(mfilename));
setGUIDataRoot('launchDir', launchDir);
%
% ----------------------------------------
% Load the init file.  You should use getExtPar to do this, because
% getExtPar sets options that are required for parallel runs.  If you
% don't want to do parallel runs, you *can* simple do:
% extPar=init;
%
runInfo.projectPath=currentProjectPath_;
runInfo.initFile='init';
extPar=getExtPar(runInfo);
%
% *** Additions to extPar here... ***
extPar.projectPath=currentProjectPath_; % ESSENTIAL!
extPar.mode='RunNoGraphics'; % Uncomment to run with no graphics ↘
        ↪ (normal for batch processing).
% ----------------------------------------
%
% Start Locust.  Note that validation of the fitness function is done
% inside LocustDriver.
LocustDriver(extPar);
```

Note that this example of a custom launcher for Locust does not return an OptimalSolutions structure directly. This is similar to the custom launcher for Ferret (Section 4.14.2), but unlike SAMOSA (Section 9.9) and Anvil (Section 10.8). The reason for this difference is that the front line optimizers, Ferret and Locust, require an analysis step, as outlined in Sections 4.15 and 11.10. The optimal solution sets from these optimizers are not available until after analysis is complete, and therefore cannot be returned from their custom launchers. SAMOSA and Anvil, on the other hand, are much simpler optimizers that are designed to produce their optimal solution sets directly, without requiring an extra analysis step. This property makes it possible to use these optimizers as polishers.

Please refer to the comments in Section 4.14.2, especially those that pertain to parallel computing. If you want to do parallel runs, it is essential to obtain *extPar* by running the program `getExtPar` rather than calling your init file directly by a command like `extPar=init`, since the `getExtPar` command also helps to ready Qubist for parallel computation. If you don't want to do any parallel computing, then you can call your init file directly, and also skip the steps labelled 'Parallelization Info'.

11.4 Analysis and Polishing

Locust is much like Ferret in that the final step in computing the OptimalSolutions structure is an analysis step, which loads all of the run's History files from disk, compares all saved solutions, and generates an OptimalSolutions.mat file in the project's data directory, which is given by the `par.history.dataDir` field of the project's setup file. Locust's analysis procedure works exactly like Ferret's analysis, which was discussed in Section 4.15, and gives you access to all of the visualization tools discussed in Chapter 5.

Locust is designed to make use of the Qubist optimizers SAMOSA, Anvil, or SemiGloSS, or MATLAB's built in optimizer fminsearch as polishers after a run is analyzed. The procedure for using the polishers in Locust is identical to the method for Ferret, which was discussed in Section 4.16.

11.5 The Locust History File

Like Ferret, Locust generates History files as it runs. These are used in the analysis procedure and for resuming runs. Normally, users do not need to load History files or work with them directly, but the following is an outline of their structure for completeness.

```
History
       |
       .X         % Matrix of parameter values for swarm members.  Each column
       |          % represents a parameter set and the number of columns is equal to
       |          % the number of optimal solutions.
       |
       .F         % Matrix of fitness values for swarm members.  The number of rows
       |          % is equal to the number of objectives, and the number of columns
       |          % is equal to the number of optimal solutions.
       |
       .auxOutput % Cell array of auxiliary output values for swarm members.  auxOutput
       |          % is empty {} when not used.  When used, each cell index corresponds
       |          % to its respective columns in the X and F matrices.
       |
       .rank      % Rank of solutions in the swarm, without regard to fuzzy tolerances.
       |          % Each element in this row vector gives the rank of the solution
       |          % represented by the corresponding column in X and F.
       |
       .graphics  % Data used to make plots in the Locust console window.
       |        |
       |        .swarmStats  % Swarm statistics, used for the upper right graphs in the
       |        |            % Locust console window.
       |        |
       |        .FOptimalSet % Fitness values from the current optimal set.
       |                     |
       |                     .min  % Minimum fitness values
       |                     |
       |                     .max  % Maximum fitness values
       |
       .par       % par structure, as loaded from the setup file.
           |
           .user.extPar  % External parameters from the user's init file.
```

11.6 The Init Function

Locust's init function is called exactly like Ferret's init function, which is explained in Section 4.6:

```
function extPar=init
```

This is the standard technique that makes *extPar* available as a local variable that is sent to the project's fitness function. Note that Locust is also able to use the 'global *extPar_*' technique discussed in Section 4.6.2, which makes the output from the init function available as a global variable called *extPar_*. This is *not* the standard way of doing things, and rarely the best technique to use. You should only consider using the global *extPar_* method after carefully considering the advice in Section 4.6.2.

It is possible to specify members of the initial swarm by hand, which is sometimes useful to speed up convergence if you have *a priori* knowledge of good solutions in the parameter space. This is done by outputting the desired solutions from your init function as a matrix `extPar.X0`. The technique is exactly

the same as used for Ferret, as discussed in Section 4.6.3.

11.7 The Fitness Function

Locust's fitness function uses the same syntax as Anvil's fitness/energy function, as discussed in Section 10.4. Like Anvil, Locust ignores the *saveData* and *XPhysMod* fields, which are allowed (but rarely used) by Ferret. The signature of a Locust fitness function is therefore

```
[F, {auxOutput}]=fitness(X, {extPar})
%
% *** Braces {} indicate optional fields. ***
%
% Input fields:
% X          -->   [NGenes x N matrix]
% extPar     -->   MATLAB structure
%
% Output fields:
% F          -->   [NObj x N matrix] (required)
% auxOutput  -->   {N-element MATLAB cell array} or {empty cell array} (optional)
```

11.8 The Output Function and the Swarm Data Structure

Like all of the other optimizers, Locust optionally gives you access to its top-level data structure called the *swarm* once per time step. Locust's output function is usually called as follows:

```
function output(swarm)
```

The output function is useful for occasionally for performing side-calculations during a run. You must, of course, save the results of such calculations, because the output function has no output arguments that can be used for this purpose. The output function can also be used to implement a custom stopping criterion, as discussed in Section 11.8.3. Section 11.8.4 discusses an advanced technique that allows you to modify the swarm during the evaluation of the output function.

11.8.1 The Swarm Data Structure

The *swarm* data structure is organized as follows:

```
swarm
     |
     .coords            % Structure representing position, velocity, and fitness
     |      |           % information for the swarm.
     |      |
     |      .X          % Matrix of parameter values for the swarm.  Each column
     |      |           % represents the parameter set for a swarm member at the
     |      |           % current time step and the number of columns is equal to
     |      |           % the swarm size.
     |      |
     |      .V          % Matrix of swarm velocities.  Same size as the X matrix.
     |      |
     |      .F          % Matrix of fitness values for the swarm.  Each column
     |      |           % represents the fitness value (or values for multi-objective)
     |      |           % problems), resulting from the fitness evaluation of the
     |      |           % corresponding column in the X matrix.
     |      |
     |      .auxOutput  % Cell array of optional auxiliary outputs from the fitness
     |                  % function.  auxOutput is empty {} when not used.  When used,
     |                  % each cell index corresponds to its respective columns in the
     |                  % X and F matrices.
     |
     .pBest             % Structure representing position and fitness information for
     |     |            % 'personal best' solutions.
     |     |
     |     .X, .F, .auxOutput  % Same meaning as in the .coords structure.
     |
     .gBest             % Structure representing position and fitness information for
     |     |            % 'global best' solutions.
     |     |
     |     .X, .F, .auxOutput  % Same meaning as in the .coords structure.
     |     |
     |     .globals     % Indexing information for global best solutions.
     |             |
     |             .allGlobals  % The indices of particles (from coords) chosen as
     |             |            % global bests.
     |             |
     |             .myGlobals   % The index of the global best solution for each
     |                          % particle in the swarm.
     |
     | * continued * --->
```

```
| <--- * continued from previous page. *
|
.graphics          % Statistical information used for plots in the top-right of
|        |         % the Locust window.
|        |
|        .swarmStats   % Matrix of info used for the 'S(#Opt) & S(<F>)' plots,
|        |             % as a function of time step.
|        |
|        .FOptimalSet  % Info used for S(F_min) plot.
|                    |
|                    .min, .max  % Max and min fitness values of particles in the
|                                % optimal set as a function of time step.
|
.timeStep        % The current time step.
|
.NEval             % The cumulative number of fitness function evaluations.
|
.par               % The user's par structure from the LocustSetup (or
    |              % FerretSetup) function.
    |
    .user.extPar   % The user's external parameters from the init function.
```

The `swarm.coords` branch of the *swarm* data structure contains the current position (`swarm.coords.X`), velocity (`swarm.coords.V`), fitness (`swarm.coords.F`), and auxiliary output (`swarm.coords.auxOutput`) information for the swarm at the current time step. The *swarm* structure also contains `swarm.pBest` and `swarm.gBest` branches that contain the same kind of information for the personal best and global best solutions respectively. Note that velocity information is not included in the *pBest* and *gBest* branches because this just wouldn't make sense. Personal and global best solutions are snapshots in time of solutions that were encountered by members of the swarm during their travels; the personal and global best solutions are not actually moving. I am not sure why a user would need the velocity information, other than for the sake of interest, but velocity information for the swarm is available to the output function in `swarm.coords.V` anyway.

The *gBest* branch contains extra indexing information to associate swarm particles with their global best solutions. `gBest.index.allGlobals` is the full list of indices for global solutions, where each index corresponds to the index of the swarm member (i.e. columns of `swarm.coords.X` and `swarm.coords.F`) that discovered it. If you are running with multiple global solutions, which is normal for Locust, you may notice that *allGlobals* contains mostly different indices at the start of a run, but that repeated indices become more common as the swarm converges. This is due to the merger of neighbourhoods, as discussed in Section 11.1.3. Mergers occur most prominently in single-objective problems with non-fuzzy fitness functions, where the swarm *should* collapse to a small region by the end of the run and multiple neighbourhoods are not required because there is no need to map out a complex solution set. You may see some mergers in multi-objective or fuzzy single-objective problems, but you are just as likely to see neighbourhoods divide. For these problems, Locust usually reaches a near-steady state late in the run (much like Ferret), where the number of neighbourhoods is roughly constant and the code is focused on mapping the optimal set as thoroughly as possible.

The `swarm.graphics` branch contains arrays of data used to generate the two plots at the upper right side of the Locust window. I have occasionally been asked how to retrieve this data, and the corresponding

graphical data for the other optimizers, so I have made it available.

The *swarm* structure contains the time step (`swarm.timeStep`) and the total number of function evaluations (`swarm.NEval`) for the run. You should treat $NEval$ as approximate, especially when running Locust in parallel mode. Accurate benchmarking information is available in the OptimalSolutions structure at the end of the run, as discussed in Section 11.10.

11.8.2 Built-In Stopping Criteria

Like the other Qubist optimizers, Locust has several built-in criteria that can be used to stop the code. These are as follows:

- The maximum number of time steps allowed by the user: `par.stopping.maxTimeSteps`.
- The maximum number of solutions evaluated by the fitness function: `par.stopping.maxFEval`.
- Tolerance in parameters: `par.stopping.XTol`.
- Tolerance in fitness: `par.stopping.FTol`

These stopping options are discussed in Section 11.11.7.

11.8.3 Custom Stopping Criterion

As usual, you can stop Locust by calling Qubist's `abortQubist` function, which sets the global variable $abort\_ = 1$ within the output function. Locust can use the same generic global variable as Ferret, rather than a more specific name like '$abortLocust\_$', because Locust is not used as a polisher, and is therefore never active at the same time as Ferret from the same MATLAB session. Further details of the `abortQubist` function are provided in Section 4.9.2. The following is a simple output function that will stop Locust when the user-defined function `checkStop(swarm)` returns `true`:

```
function output(swarm)

if checkStop(swarm)
        abortQubist;
end
```

11.8.4 Advanced Features: Modifying $extPar$ and the $swarm$

Section 4.9.3 discussed an advanced feature in Ferret that allows you to modify the *extPar* data structure (generated by your init function) or even the top-level *world* structure from within the output function. Locust has an analogous feature that lets you modify *extPar* or the top-level *swarm* data structure from within the output function. Thus, advanced users may occasionally find the following forms of the output function useful:

```
function extPar=output(swarm)
function [dummy,swarm]=output(swarm)
```

The first usage is straightforward. Just grab the *extPar* structure with a line like `extPar=swarm.par.user.extPar`, modify as required, and return the data structure.

The second usage is more complex and should be used with extreme caution. This feature gives you access to the entire *swarm* data structure and allows you to change it as you see fit. You may crash Locust if you change it in a way that makes the swarm invalid. This feature is intended for advanced users on problems where they need to do something special to the *swarm* between time steps. It is not required for most problems, and you should do significant testing if you try this.

Section 4.9.3 discussed a demo called 'Advanced/Data-Modeler-modWorld', which modifies the *world* from Ferret's output function. The idea of the demo is that we are modeling some artificial data with a Fourier series. We start with two Fourier terms, requiring four parameters, and refine the Fourier series by adding a term, and two parameters, every 25 generations. The same demo is also set up to modify the *swarm* data structure when run with Locust, where we add a Fourier term every 25 time steps.

Locust is set up to call the output function called outputLocust.m for this particular demo:

```
function [dummy, swarm]=outputLocust(swarm)
dummy=[];

% Add parameters every NRefine generations:
NRefine=25;

if mod(swarm.timeStep, NRefine) == 0
    [NPar, N]=size(swarm.coords.X); %#ok
    swarm.par.general.min=[swarm.par.general.min, 0, 0];
    swarm.par.general.max=[swarm.par.general.max, 1, 2*pi];
    swarm.par.general.cyclic=[swarm.par.general.cyclic, NPar+2];
    %
    %
    % Add 2 new parameters by adding 2 rows to swarm.coords.X:
    XMod=[rand(1,N); 2*pi*rand(1,N)];
    swarm.coords.X=[swarm.coords.X; XMod];
    %
    % Fitness will be re-evaluated automatically.  No need to do it
    % here.
    %
    % Tell Locust that you want to restart.
    setGUIData('restart', true);
end
```

You need to design a trigger to determine when the swarm is modified. For the purpose of this demo, I trigger a refinement to the Fourier series every 25 time steps, but you will probably want something fancier for your problem. Normally, you would write some code to monitor the convergence of the swarm, and trigger a refinement when it is sufficiently converged. The triggering criterion depends on the problem and is entirely up to you.

The following steps are required to modify the *swarm* from inside your output function:

- Set new parameter limits if required by modifying `swarm.par.general.min` and `swarm.par.general.max`.

- Modify `swarm.par.general.cyclic` to indicate which parameters are cyclic, if this information has changed.
- Modify `swarm.coords.X` to suit the needs of your problem.
- Send a 'restart' signal to Locust by executing the line `setGUIData('restart', true)`. Errors are likely to occur if you modify the swarm and forget to send this signal to Locust. Note that you should only send the restart signal if your trigger evaluates to *true* and you have actually modified the swarm.

The restart signal tells Locust to repair the *swarm* as required, restart certain internal systems, and re-evaluate all solutions in the *swarm*. Re-evaluation is necessary because the number of parameters (or their meanings) may have changed, and this changes the evaluation of all swarm particles. This is computationally expensive, and this feature should therefore be used sparingly.

Section 4.9.3 contained an extensive discussion and some cautions about how Ferret's restart mechanism affects its linkage-learning and critical parameter detection systems. Clearly, these cautions do not apply to Locust, which does not contain either of these systems.

11.9 The myPlot Function

Locust supports a myPlot syntax that works exactly as it does with the other Qubist optimizers:

```
function myPlot(X)
function myPlot(X,extPar)
function myPlot(X,F,extPar)
function myPlot(X,F,auxOutput,extPar)
function myPlot(X,F,auxOutput,rankPareto,extPar)
```

The myPlot function is called once per timeStep, and the resulting User Plot appears in the lower left axes on the Locust window.

11.10 Analysis and the OptimalSolutions File

Like Ferret, Locust generates History files as it runs, and an analysis step is required to condense the History files into the final optimal set. Analysis can be started at any time during a run, or at the end of a run, by clicking the 'Analyze' button on the Locust window. Analysis can also be started automatically at the end of a run by setting `par.analysis.analyzeWhenDone=true` in the setup file.

Figure 11.2 shows the Locust analysis window, which is started automatically when a run is analyzed. This is very similar to the Ferret analysis window (see Figure 5.2), and provides access to all of Ferret's visualization features, which are the subject of Chapter 5.

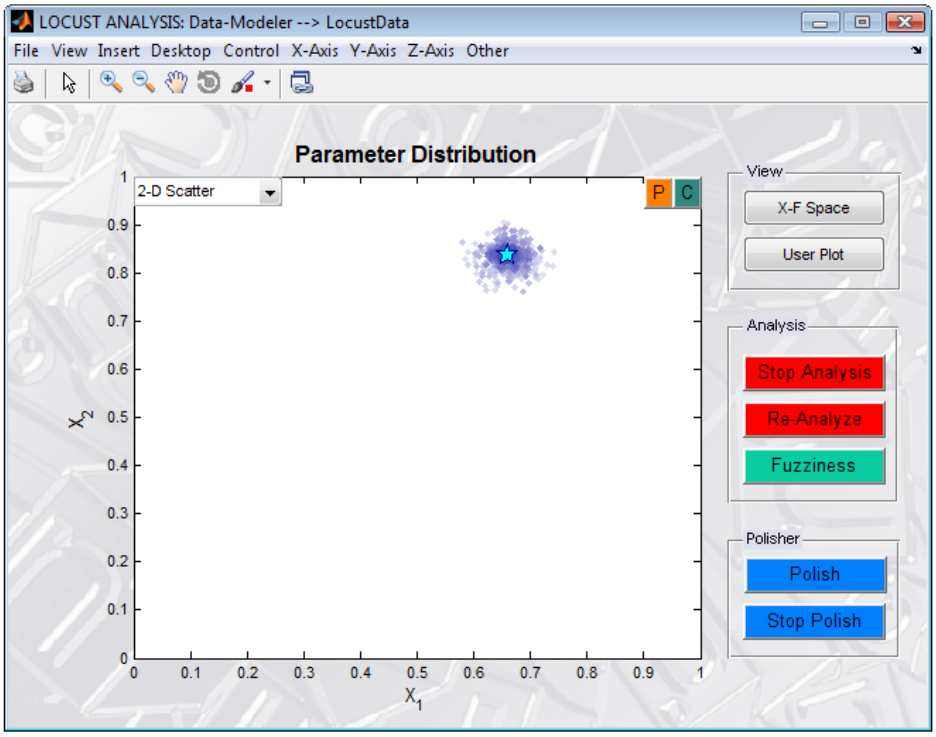

Figure 11.2: Locust analysis of the 'Data-Modeling/Data-Modeler' problem from the [Qubist_Home]/demos directory.

The OptimalSolutions.mat file encapsulates the final results for the run. This file is normally loaded into MATLAB by the user and used for post-processing calculations in user-defined code. The OptimalSolutions.mat file is structured as follows:

```
OptimalSolutions
                |
                .X         % Matrix of parameter values for optimals.  Each column
                |          % represents a parameter set and the number of columns
                |          % is equal to the number of optimal solutions.
                |
                .F         % Matrix of fitness values for optimals.  The number of
                |          % rows is equal to the number of objectives, and the
                |          % number of columns is equal to the number of optimal
                |          % solutions.
                |
                .auxOutput % Cell array of auxiliary output values for optimals.
                |          % auxOutput is empty {} when not used.  When used, each
                |          % cell index corresponds to its respective columns in the
                |          % X and F matrices.
                |
                .rank      % Rank of solutions in the optimal set, without regard
                |          % to fuzzy tolerances.  Each element in this row vector
                |          % gives the rank of the solution represented by the
                |          % corresponding column in X and F.
                |
                .std.X     % Standard deviation of X.  This is used by some of the
                |          % polishers.
                |
                .par       % par structure, as loaded from the setup file.
                |   |
                |   .user.extPar  % External parameters from the user's init file.
                |
                .benchmark     % Useful information for benchmarking.
                          |
                          .gen    % Vector of sequential generation numbers
                          |
                          .NEval  % The total number of solutions evaluated by the
                          |       % fitness function.
                          |
                          .time   % Numerical times corresponding to when each
                          |       % timestep finished.  The format is the same as
                          |       % MATLAB's 'now' function.
                          |
                          .timeString  % The corresponding human-readable formatted
                          |            % time strings.  The format is the same as
                          |            % MATLAB's 'datestr' command.
                          |
                          .FOptimalSet % Fitness values from the current optimal set.
        |                 |
                          .min         % Minimum fitness values
                          |
                          .max         % Maximum fitness values
```

The fields of a Locust OptimalSolutions.mat file are very similar to the corresponding fields of a Ferret OptimalSolutions.mat file. Please refer to Section 4.18 for a detailed explanation. Note that `OptimalSolutions.benchmarking` contains useful information for benchmarking your run, and Section 4.18 gives a short program that shows how to plot this information. This is especially useful when comparing the performance of Ferret and Locust when the two codes are run on the same project.

11.11 The LocustSetup File

The Locust setup file is relatively simple and compact compared to the other Qubist optimizers. As usual, I will walk you through the setup options and try to provide some useful advice. Locust's default setup file is located in [Qubist_Home]/Qubist/user/Locust/defaults/defaultLocustSetup.m. You can modify these defaults if you are sure that you know what you are doing, but it is better to copy the template setup file from [Qubist_Home]/Qubist/user/Locust/templates/LocustSetup_template.m into your project directory, rename it to LocustSetup.m (or the setup file name that you set in your launchQubist.m file), and modify the settings there.

11.11.1 par.user

The `par.user` branch of the setup file contains the name of the fitness function (`par.user.fitnessFcn`) and the name of the optional output function (`par.user.output`). Both of these fields must be strings and not function handles. The name of the output function can be an empty string, to indicate that no output function is used, but a fitness function name must be given.

```
% User
par.user.fitnessFcn='fitness'; % [string]: Name of the fitness function.
par.user.output=''; % [string]: Name of optional user-defined output function called ↘
        ↪ each time step.
par.user.extPar=[]; % Do not edit this line.
```

11.11.2 par.history

The `par.history` branch of the setup file contains two options that control how History files are saved.

```
% History
par.history.dataDir='LocustData'; % [string]: Directory for Locust data files, etc.
par.history.NStepsPerHistoryFile=25; % [integer >= 1]: How many time steps per ↘
        ↪ History file?
```

`par.history.dataDir` is a string containing the name of the data directory where the History file directory and the OptimalSolutions.mat file will be stored. This is usually specified as a local directory within the project directory, as shown here, but full global paths are also allowed.

`par.history.NStepsPerHistoryFile` is the number of time steps that are to be saved per History file. The advice that I gave in the section on Ferret's History file and crash recovery options (Section 4.16.1) applies here also. Smaller values of NStepsPerHistoryFile are better for projects with expensive fitness functions, because a run can be resumed from the last saved History file after a crash or error. Larger

values are more suitable for projects where the fitness function is cheap and the run duration is short.

11.11.3 par.general

`par.general` contains basic options that tell Locust about the structure of your problem's parameter space. All of these options are also in Ferret's `par.general` section and are discussed in Section 8.3 in detail. I will only briefly review them here:

- `par.general.min` and `par.general.max` respectively give the minimum and maximum bounds of the parameter space. They must be be row vectors of length equal to the number of parameters. Locust checks bounds at each step, and particles are not allowed to leave these bounds.

- `par.general.cyclic` is a row vector containing an index of parameters designated as cyclic. All values must be integers between 1 and N_{par}, where N_{par} is the number of parameters. Cyclic parameters are especially intuitive for a PSO. Designating a parameter as cyclic imposes periodic boundary conditions on that parameter; if a particle goes beyond the minimum or maximum sides of the parameter bounds, it reappears on the other side with the same velocity.

- `par.general.XLabels` and `par.general.FLabels` are optional cell arrays of strings that respectively give the names of the parameters and the objective. Parameter and objective names that are not specified are given the generic names 'X_1', 'X_2', 'X_3', etc. for parameters and 'F_1', 'F_2', 'F_3', etc. for objectives.

- `par.general.HolderMetricExponent` is the metric exponent used to calculate distances for Locust's niching mechanism. I always use `par.general.HolderMetricExponent=2`, which specifies a Euclidean space. I have not fully explored this parameter, but I don't believe that different values are likely to make much difference.

```
% General
par.general.min=[-1,-1]; % [real vector]: Minimum values of all parameters.
par.general.max=[1,1]; % [real vector]: Maximum values of all parameters.
par.general.cyclic=[]; % [integer vector > 1]: Which parameters are cyclic?
par.general.XLabels=; % Give names to some or all parameters: 'A','B',...
par.general.FLabels=; % Give names to some or all fitness values: 'FA','FB',...
par.general.HolderMetricExponent=2; % The exponent used for the metrics.
```

11.11.4 par.parallel

I designed Qubist's parallel computing system specifically for Ferret and ported it to Locust during the development of the greatly improved Locust-2 version, as discussed in Section 11.1.1. Almost all of the code for parallel computing is shared between these optimizers, and Locust's parallel computing options are identical to Ferret's. Please refer to Chapter 6 and Section 8.5 for details.

```
% Parallel Computing
par.parallel.NWorkers=0; % [integer >= 0]: Number of worker nodes to launch initially.
par.parallel.minChunkSize=10; % [integer]: Minimum number of evaluations in each ↘
        ↪ work chunk.
par.parallel.timeout=30; % [integer >= 0]: Maximum time in seconds before an ↘
        ↪ unresponsive node disconnects.
par.parallel.latency=0; % [real > 0]: Time to pause in seconds after writing to the ↘
        ↪ scratch directory.
par.parallel.useJava=true; % [logical]: Is Java required for worker nodes?
par.parallel.writeLogFiles=true; % [logical]: Are log files required?
```

11.11.5 par.swarm

The `par.swarm` options control the size, dynamics, and topology of the particle swarm. These options will probably have a significant effect on your runs. Therefore, they should be modified to fine-tune your setup, but this must be done with some care.

```
% PSO constants
par.swarm.N=100; % [integer >= 1]: Number of particles in the swarm.
par.swarm.cg=0.5; % [real]: Global best constant.
par.swarm.cp=1; % [real]: Personal best constant.
par.swarm.dt=1; % [real]: Time step (usually 1)
par.swarm.TDamp=10; % [real]: Damping constant.
par.swarm.globalFrac=1; % [real]: Number of global neighbourhoods as fraction of swarm size.
par.swarm.globalRange=0; % [real]: Fuzziness of global neighbourhoods.
par.swarm.PCompeteGlobal=0.1; % [real]: Probability of globals competing.
par.swarm.PKick=0; % [real << 1]: Probability of perturbing swarm with a random 'kick'.
par.swarm.kickSize=0.05; % [real <~ 0.1]: Size of the kick.
```

`par.swarm.N` is the number of particles in the swarm. I recommend using about 100 to perhaps a few hundred particles for most problems. Locust will often find good solutions with as few as $\sim$10 particles, but very small swarms worry me when the goal is to find solutions that are truly optimal in a global sense. PSOs work by exploring the parameter space by the action of many particles sampling the space in parallel. If the number of particles is small, then the sampling will be less thorough, and it is possible to miss the global solution or solution set. At the other extreme, I don't see much benefit to extremely large swarms (more than about 1000 particles), except for perhaps on very difficult problems in very large parameter spaces. However, you would almost certainly be better off using Ferret than Locust, if your problem is really this difficult. My advice is to start with about 100 particles and decrease the number of particles gradually, if you want, and if experience shows that this decreases the run time without losing reliability. You should increase the size of the swarm if you notice that you are getting different answers each time that you run the code, because this indicates that Locust is probably getting trapped in a local minimum. Larger swarms sample the parameter space more thoroughly and this often helps such consistency problems.

`par.swarm.cg` and `par.swarm.cp` are respectively the global best and personal best constants used in equation 11.1. They should both be around 1, although I prefer to use a cg value that is a little smaller than cp ($cp = 1$; $cg = 0.5$). Decreasing cg relative to cp places more emphasis on exploration of the parameter space because the particles search more independently and are influenced less by the global best solution or solutions. Increasing cg relative to cp places more emphasis on the exploitation of the global

solution or solutions, at the expense of exploration, because all particles will be drawn to the optimal region more rapidly.

Particles communicate with each other via the global solution(s), and the strength of this communication is critical to the success of the algorithm. Too little communication ($cg \ll 1$) causes all of the particles to act almost independently. Try setting $cg = 0$ as an experiment, and you will discover that using N particles to explore the space independently is *extremely* inefficient (i.e long convergence times, if convergence occurs at all) compared to the default setting ($cg = 0.5$), which allows enough communication for collective swarm behaviour to emerge. Conversely, setting $cg \gg 1$ results in poor solutions because the particles converge too quickly, before the parameter space has been sufficiently explored. These are good parameters to modify as you fine-tune your setup file. However, I would keep cg and cp similar in size and avoid extreme values.

`par.swarm.dt` is the time step between updates to the swarm positions and velocities, and the default setting is `par.swarm.dt=1`. Equation 11.1 shows that this will cause particles to move most of the distance between their present positions, and their personal best and global best orbit centres during a time step. Such large jumps would cause very large errors in the orbits if Locust really used equation 11.1 to update positions and velocities. Recall, however, that Locust uses an exact analytical solution to the orbital dynamics, which diminishes these numerical errors to an insignificant level. Therefore, the time step dt affects the rate of sampling of the parameter space as particles move around on their orbits, but has no significant effect on the accuracy of the obits. I find that $dt \approx 1$ provides the best sampling and the fastest convergence. A fun thing to try is to set $dt \approx 0.01$, which allows you to see the actual motion of particles moving smoothly on their orbits. This is particularly useful for demonstrations and talks, or when explaining the concepts of PSOs. Turning dt back up to $dt \approx 1$ is like setting the particles on identical orbits (excluding the stochastic effects of ξ_g and ξ_p variable in equation 11.1), but observing them with a strobe that only flashes once per orbit or so.

PSOs require some damping so that orbits eventually decay and particles converge to the optimal region. `par.swarm.TDamp` controls the time required for damping to occur. Smaller values of $TDamp$ correspond to more damping with shorter damping times, while larger values of $TDamp$ result in less damping and longer damping times. I have implemented this parameter so that `par.swarm.TDamp=1` corresponds to a *critically damped oscillator* - an oscillator whose motion damps out in the shortest time possible. $TDamp > 1$ corresponds to an underdamped oscillator, whose motion damps out over multiple orbits. $TDamp < 1$ would correspond to an overdamped oscillator, whose damping is so strong that oscillations do not even occur. *Clearly, Locust must operate in the regime of underdamped orbits, since many orbits are required to explore the space. Therefore, you should not use values of TDamp less than 1, and I recommend a value of about* $TDamp = 10$. Values much smaller than this default will cause orbits to damp before sufficient exploration occurs, while values that are much larger will prevent orbits from ever settling into the optimal region.

`par.swarm.globalFrac` is the number of global neighbourhoods as a fraction of the size of the swarm. For example, the default value of `par.swarm.globalFrac=1` means that each particle is in its own global neighbourhood at the start of the run, and the particle's global best solution is the same as its personal best. This swarm topology won't persist for very long for most problems because global neighbourhoods will start to merge automatically, but this technique allows the start of the run to be devoted entirely to exploration by individual particles. Collective swarm behaviour, which requires multiple particles per global neighbourhood, won't emerge until later in the run after neighbourhoods start to merge. This allows a gradual transition from individual exploration of the parameter space to collective exploitation of the optimal region, which works extremely well in practice. You don't *have* to use `par.swarm.globalFrac=1`, but I would definitely recommend setting `par.swarm.globalFrac` to a significant fraction of the swarm size, and definitely not less than 0.1 or so. A high value for $globalFrac$ cannot hurt your run, but it might

help a lot. The only potentially negative effect of a large *globalFrac* is that it tends to delay convergence until neighbourhoods merge. However, this delay of convergence often buys significant improvements to the exploration of the parameter space, which justifies the extra computational expense for many problems.

Setting `par.swarm.globalRange = 0` (the default value) tells Locust to assign each particle's global solution as the nearest global in the parameter space. Thus, neighbourhoods have hard boundaries, and mixing only occurs when particles cross neighbourhood boundaries. Setting `par.swarm.globalRange > 0` causes global neighbourhoods to become 'fuzzy' in the sense that particles will be assigned globals probabilistically following an exponential distribution such that the n'th nearest global with be chosen with probability

$$P \propto e^{-n \cdot globalRange}. \tag{11.3}$$

For example, if $globalRange = 1$, the second nearest global will be chosen $1/e \approx 0.37$ times as frequently as the nearest global. I usually just use $globalRange = 0$, but non-zero values might be useful if you want greater mixing between global neighbourhoods. In my experience, this parameter does not have a large effect on the behaviour of the code.

Global neighbourhoods merge by competitions in a binary tournament between globals. Every time step, a fraction of globals approximately equal to `par.swarm.PCompeteGlobal` enter into a binary competition in which their fitness values are compared in a Pareto-optimal sense. If there is a clear winner, taking into account fuzzy fitness functions (see Section 11.11.6), then the inferior global is replaced with a copy of the superior solution, which causes their corresponding global neighbourhoods to merge. Mergers are not always permanent, since neighbourhoods can also split when multiple solutions are present in a neighbourhood that are equally good. A large value for *PCompeteGlobal* is like a high selection pressure in a genetic algorithm; it emphasizes exploitation of the optimal region over exploration of the space, and increases the rate of convergence at the expense of the thoroughness of the search. Low values of *PCompeteGlobal* will decrease mergers, resulting in more neighbourhoods later in the run, thus encouraging more exploration while delaying convergence. I find that the default value of $PCompeteGlobal = 0.1$ works well for many problems, but this is a good parameter to adjust if you want to adjust the convergence rate of your run.

Locust contains a 'kick' operator, which works much like a genetic algorithm's mutation operator, and can similarly help on problems that have many local minima. `par.swarm.PKick` is the probability that a Gaussian random kick will be applied to both the position and velocity of all particles simultaneously during a given time step. This has the effect of stirring up the swarm and hopefully knocking it out of a local minimum. Obviously, this must be done very sparingly, or it will interfere with convergence. I recommend that you only use this feature if Locust seems to be getting stuck frequently or of there is evidence of local minima in the parameter space. `par.swarm.PKick` should be set to no more than a few percent, and values greater than 0.05 are probably too disruptive to be useful. `par.swarm.kickSize` is the size of the kick, relative to the last move of each particle. Values of $kickSize = 0.05$ are usually adequate, but values that are much larger will almost certainly result in poor convergence.

11.11.6 par.selection

Locust allows fuzzy tolerances that affect how personal best and global best solutions are selected, as well as how winners are chosen in the binary tournaments that allow global neighbourhoods to merge (see Section 11.11.5). Fuzzy tolerances also affect how the final optimal set is chosen during the analysis procedure.

```
% Selection
par.selection.FAbsTol=0; % Absolute fitness range (+/- FAbsTol) to use as a fuzzy ↘
        ↪ fitness band.
par.selection.FRelTol=0; % [0 - 1]: Fraction of fitness range to use as a fuzzy ↘
        ↪ fitness band.
par.selection.exploitFrac=0.5; % [0:1]: Fraction of swarm devoted to exploiting ↘
        ↪ best-ranked solutions, as opposed to exploring.
```

`par.selection.FAbsTol` and `par.selection.FRelTol` correspond respectively to the absolute and relative tolerances of the fitness functions and are applied whenever one solution is compared to another during a Locust run or during analysis. Just like the corresponding options in Ferret, the net result is that Locust is insensitive to differences in the fitness function that are less than these tolerances, and the fitness function gains a certain amount of fuzziness when either $FAbsTol$ or $FRelTol$ is greater than zero. These control parameters work exactly like the corresponding options in Ferret. Please refer to Section 8.6 for further details.

`par.selection.exploitFrac` works like the strategy parameter of the same name in Ferret. This option tells Locust what fraction of the swarm to devote to exploiting the best solutions in the optimal region, as opposed to exploring the extended structure of the optimal set in problems with fuzzy fitness functions. Note that $exploitFrac$ has no effect on non-fuzzy problems where $FAbsTol = 0$ and $FRelTol = 0$.

11.11.7 par.stopping

Locust contains simple built-in stopping criteria that are very similar to the stopping criteria in Ferret, SAMOSA, and Anvil. These can be used for simple problems when a custom stopping criterion is not required or has not yet been implemented in the output function (see Section 11.8).

```
% Stopping criteria
par.stopping.maxTimeSteps=Inf; % [integer > 0]: Maximum number of time steps.
par.stopping.maxFEval=Inf; % [Integer > 0]: Maximum number of function evaluations ↘
        ↪ (approximate).
%
% Tolerance on X and F - mainly for single-objective problems.
par.stopping.XTol=NaN; % [real > 0, or real vector (length=# of parameters)]: ↘
        ↪ Tolerance on parameters.
par.stopping.FTol=NaN; % [real > 0, or real vector (length=# of objectives)]: ↘
        ↪ Tolerance on objectives.
par.stopping.boolean=@or; % [boolean operator]: '@or' or '@and' for exit criteria.
```

Locust will stop if the number of time steps exceeds `par.stopping.maxTimeSteps`, or the number of parameter sets evaluated by the fitness function exceeds `par.stopping.maxFEval`. The function evaluation count should be accurate enough for any reasonable purpose, but it not guaranteed to be exact. These criteria are turned off by setting $maxTimeSteps$ and $maxFEval$ to NaN or Inf, which is the default.

Locust provides an additional stopping criterion based on the convergence of the swarm. These stopping criteria are as follows:

- The standard deviation of the particle coordinates (`swarm.coords.X`) must be less than `par.stopping.XTol` in all parameters. $XTol$ may be either a scalar or a vector whose length is equal

to the number of parameters.

- The standard deviation of the particle objectives (`swarm.coords.F`) must be less than `par.stopping.FTol` in all objectives. *FTol* may be either a scalar or a vector whose length is equal to the number of objectives.

These convergence criteria are turned off by setting `par.stopping.XTol=0` or `par.stopping.FTol=0`.

`par.stopping.boolean` must be a function handle whose value is either @or, or @and. If `par.stopping.boolean=@and`, Locust will exit when both of the *XTol* and *FTol* criteria are satisfied. The code only requires one criterion to be true if `par.stopping.boolean=@or`.

11.11.8 par.niching

As a multi-objective optimizer, Locust's goal is to map the Pareto front as thoroughly as possible by encouraging solutions to spread out as much as possible over the trade-off surface. Of course, parameter space mapping is also required for single-objective problems with optimal regions that are not point-like, which is most commonly due to a fuzzy objective function (See Section 11.11.6, as well as the example in Section 5.1.). Ferret accomplishes this by means of its niching system, which is discussed in Section 8.9. When I started thinking about multi-objective particle swarms, it quickly became obvious that I needed something like Ferret's niching system to keep particles well-spread out and to keep global neighbourhoods from merging too rapidly. Fortunately, Ferret's niching system was very well developed by this point, and similar techniques could be applied to Locust with little modification.

```
% Niching
par.niching.priority='XF'; % [string: re-order 2 letters, 'X', or 'F']: ↘
        ↪ Priority of niching. If empty, use Pareto niching.
par.niching.X=0.25;  % [0:1]: X-Niching: Typically ~0.25 is about right.
par.niching.F=0;  % [0:1]: F-Niching: Typically ~0.25 is about right.
par.niching.exponent=2; % [real: usually > 0 & <~ 2]: Used in the niche function.
```

`par.niching.priority`, `par.niching.X`, `par.niching.F`, and `par.niching.exponent` work exactly like the corresponding options in Ferret, with the slight exception that there is no analogue to 'pattern niching' in Locust. Therefore `par.niching.priority` can only contain the letters 'X' and 'F'; `par.niching.priority='XF'` and `par.niching.priority='FX'` are the only valid choices. Ferret's niching options are explained in detail in Section 8.9, and everything in that section applies equally well to Locust.

11.11.9 par.analysis

Locust's analysis procedure is based entirely on Ferret's, with the only modifications due to differences between Ferret's *world* data structure and Locust's *swarm* structure. Section 8.16, which discussed Ferret's analysis options, applies equally well to Locust.

```
% Analysis
par.analysis.FRelTol=0; % [0 - 1]: Fraction of fitness range to use as a fuzzy ↘
        ↪ fitness band.
par.analysis.FAbsTol=0; % Absolute fitness range (+/- FAbsTol) to use as a fuzzy ↘
        ↪ fitness band.
par.analysis.analyzeWhenDone=false; % [logical]: Do analysis automatically when ↘
        ↪ evolution stops.
par.analysis.conserveMemory=false; % [logical]: Minimize memory usage during ↘
        ↪ analysis?  (Analysis will be slower)
par.analysis.maxItNoProgress=100; % [integer >= 1]: Max iterations with no progress ↘
        ↪ before analysis stops.
par.analysis.postProcess=''; % [string]: Name of code to process OptimalSolutions ↘
        ↪ when analysis is done.
```

11.11.10 par.polish

Locust's polishing system can be started from the analysis window by clicking on the 'Polish' button. The `par.polish` options are identical to Ferret's polishing options, and are discussed thoroughly in Section 8.18.

```
% *** Polisher Setup ***
% par.polish.optimizer='fminsearch';
% par.polish.optimizer='SAMOSA';
% par.polish.optimizer='Anvil';
% par.polish.optimizer='SemiGloSS';
par.polish.optimizer='default'; % string]: Use defaults.
%
% fminsearch:
par.polish.fminsearch.NTracks=Inf; % [integer]: Maximum number of tracks. ↘
        ↪ 'Inf' --> all in optimal set.
par.polish.fminsearch.options=[]; % options structure from optimset.
%
% SAMOSA uses its own setup file.  Here, we just add the number of tracks requested.
par.polish.SAMOSA=defaultSAMOSA_setup; % [m-file name]: Name of SAMOSA setup function.
par.polish.SAMOSA.NTracks=Inf; % [integer]: Maximum number of tracks.  'Inf' --> all ↘
        ↪ in optimal set.
%
% Single/Multi-Objective: Anvil & SemiGloSS uses their own setup files.
par.polish.Anvil=defaultAnvilSetup;  % [m-file name]: Name of Anvil setup function.
par.polish.SemiGloSS=defaultSemiGloSS_setup; % [m-file name]: Name of SemiGloSS ↘
        ↪ setup function.
```

11.11.11 par.interface

Locust's `par.interface` options affect the fonts and colours of the Locust console window and the analysis window, as well as the default axes that are displayed. None of these options are critical to the run and I rarely change them. This section of the setup file is copied almost verbatim from Ferret's `par.interface` section, with the addition of `par.interface.plotStep`. The *plotStep* option controls how often the

graphics are updated. For example, the default setting of `par.interface.plotStep=1` tells Locust to update the graphics every time step, `par.interface.plotStep=10` would update graphics only every tenth time step, etc. Please refer to Ferret's `par.interface` options in Section 8.20 for information about the other options.

```
% Graphics & messages
par.interface.graphics=1; % [integer: 1 or -1]: Graphics on or off?
par.interface.myColorMap='bone'; % [string: colormap name] User choice for colormap.
par.interface.plotStep=1; % [integer >= 1]: How often to update graphics?
par.interface.myPlot=''; % [string]: Name of custom plot function.
par.interface.fontUnits='points'; % [string]: Must be 'points' or 'pixels'.
par.interface.titleFontSize=12; % [integer >= 1]: Self-explanatory.
par.interface.labelFontSize=10; % [integer >= 1]: Self-explanatory.
par.interface.axisFontSize=8; % [integer >= 1]: Self-explanatory.
par.interface.xAxis.type='X'; % [string: 'X' or 'F']: Default X-axis type
par.interface.xAxis.value=1; % [integer >= 1]: Default X-axis variable ↘
        ↪ (1 - NPar or NObj)
par.interface.yAxis.type='X'; % [string: 'X' or 'F']: Default Y-axis type
par.interface.yAxis.value=2; % [integer >= 1]: Default Y-axis variable ↘
        ↪ (1 - NPar or NObj)
par.interface.zAxis.type='F'; % [string: 'X' or 'F']: Default Y-axis type
par.interface.zAxis.value=1; % [integer >= 1]: Default Y-axis variable ↘
        ↪ (1 - NPar or NObj)
par.interface.NPix=50; % [integer >= 1]: Size of the grid for contour and mesh plots.
par.interface.NContours=10; % [integer >= 1]: Number of contour levels for contour ↘
        ↪ plots.
```

Chapter 12

Other Qubist Tools

'In short, intelligence, considered in what seems to be its original feature, is the faculty of making artificial objects, especially tools to make tools, and of indefinitely varying the manufacture.'

-Henri Bergson, *Creative Evolution*

The purpose of this chapter is to show you how to use a few extra tools in the Qubist package, which can help you to manage runs and understand your results. These tools are as follows:

1. The Resume Tool for Ferret and Locust (Section 12.1)
2. The Analyze History Tool for Ferret and Locust (Section 12.2)
3. The OptimalSolutions Viewer (Section 12.3)
4. The Merge OptimalSolutions Tool (Section 12.4)
5. The Ferret History Explorer (Section 12.5)
6. the Qubist Movie Tool (Section 12.6).

The first three of these tools are the most useful and I use them very frequently. The Merge OptimalSolutions tool is also very useful when compiling the results from several runs, especially when generating the final results for a project. I don't use the Ferret History Explorer all that often, but this has been occasionally useful for examining the long-term evolution of a run, particularly when trying to understand what Ferret did (and how to tweak its setup file) when a run surprises me or obtains poor results. Finally, the Movie Tool is a handy feature that I use when making movie clips of a run for a seminar or demonstration.

All of these tools are included in Qubist and are normally launched from the component selector interface that is started by running your launchQubist.m file. *However, the Analyze History Tool, the OptimalSolutions Viewer, and the Merge OptimalSolutions Tool are also available as components of a standalone MATLAB toolbox called 'QubistFreeTools' from nQube.* This software includes the entire set of

visualization tools from the full Qubist package, which is meant to allow users to view their results on any computer, and to share their OptimalSolutions.mat files conveniently with others.

12.1 The Ferret/Locust Resume Tool

Section 4.17 discussed a simple technique for resuming Ferret runs that works when neither the project directory nor the data directory have been moved. This simple method is most often used to resume suspended or crashed runs, but I also discussed several common scenarios in that section where these conditions are not true and the simple technique therefore cannot be used. Briefly, this situation arises when:

- Files are moved for archiving and you need to restart an old run.
- If you send your History files to someone, and they need to restart the run.
- If your project is coded such that it names the run automatically using the date, and you need to restart it on a different date than when it was originally started.

These situations are not at all contrived, and I find that the Resume Tool gets a lot of use. Note that it can be used for Locust runs as well as Ferret runs.

The Resume Tool is started from the Component Selector window (see Section 3.2) and is just a series of dialogue boxes that ask you to locate the data and project directories on disk, and identify a file to load paths, if required. The exact procedure should be carried out as follows:

1. Run launchQubist.m to start the Component Selector window, as shown in Figure 3.3.
2. Select 'Resume Ferret/Locust' and click the 'Launch' button.
3. This brings up a window that allows you to navigate to your **data** directory (it should open in your most recently used data directory). Note that this is probably not the same as your project directory. The data directory is the one that contains the History folder, which in turn contains the History files.
4. This brings up another directory navigation window, but this time you are asked to locate your **project** directory. This is the directory that contains your setup, fitness, and init files. Navigate to your project directory and click 'Open'. Note that Ferret/Locust should automatically pick up the location of your project directory from the History files in the data directory, and start the directory navigator from this window, unless the project directory has been moved.
5. Ferret/Locust will launch and you should see a dialogue box asking if you wish to execute an init file, or other file, to load any additional paths, as shown in Figure 12.1. Recall that Ferret/Locust loads *only* the project directory pointed to by the launchQubist.m file, as discussed in Section 4.6. *Even subdirectories of the project directory are not loaded automatically.*

   Usually, you load additional paths by including 'addpath' lines in the init file, but you can also load paths from any other file if the init file calls it. If you require additional paths, click 'yes' in the dialogue box, which will start a file navigation window that will allow you to select your path-loading file. Navigate to your file, click 'Open', and your Ferret or Locust run should resume. Note that if your additional paths are defined in your init file, then running this file will load them, but the

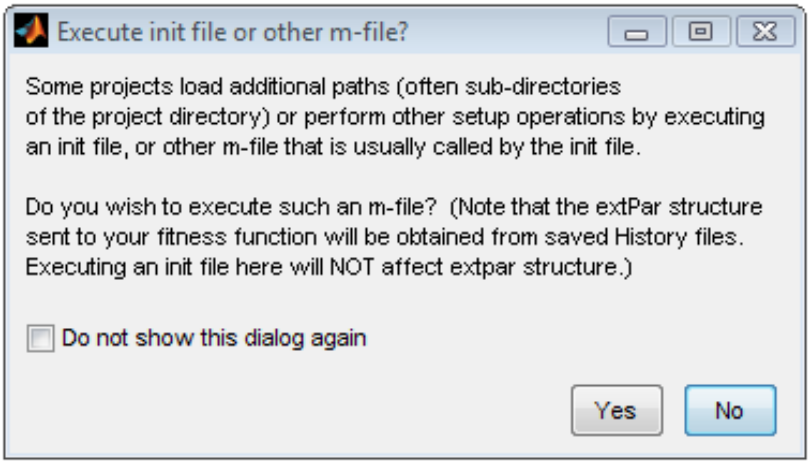

Figure 12.1: Dialog box allowing you to load additional paths prior to resuming a run.

extPar structure normally generated by init *will not be passed to Ferret/Locust* (see Section 4.6). The *extPar* structure will be recovered from History files instead. For this reason, I usually put path-loading instructions in a separate file that is called by by init file, so that I do not have to run the entire init file just to load a few paths.

Note that Resume Tool displays instructions in the MATLAB command window throughout this process, and also in the directory navigation window for some operating systems. These steps should become second nature after you have done this a few times.

12.2 The Analyze History Tool

It is possible to analyze previously saved History files using the 'Analyze History Tool', which can be started from the 'Analyze History' radio button in the launcher window. This is useful for analyzing previously completed runs after the Ferret or Locust console window has been closed. I use this feature frequently when doing multiple sets of Ferret/Locust runs for a project (often for testing), when I find it more convenient to keep the queue of runs moving, rather than pausing for analysis when each run finishes. The 'Analyze History' interface allows me to reserve my faster computers for the runs themselves, and I can do my analysis and visualization on a less powerful machine afterwards, as each set finishes. The Analyze History interface is a simple menu-driven system that presents the user with a set of dialogue boxes to load the History files and other project information.

The steps required to use the Analyze History Tool are similar to the steps required for the Resume tool discussed in Section 12.1 and much of the following sequence is copied almost verbatim from that section:

1. Run launchQubist.m to start the Component Selector window, as shown in Figure 3.3. Select 'Analyze History' and click the 'Launch' button.
2. This brings up a window that allows you to navigate to your **data** directory (it should open in your most recently used data directory). Note that this is probably not the same as your project directory.

The data directory is called FerretData or LocustData by default and contains the History directory, which in turn contains the History files. Navigate to your data directory and click 'Open'.

3. This brings up another directory navigation window, but this time you are asked to locate your **project** directory. This is the directory that contains your setup, fitness, and init files. Navigate to your project directory and click 'Open'. Note that the Analyze History Tool should automatically pick up the location of your project directory from the History files in the data directory, and start the directory navigator in this directory if it still exists on your file system.

4. A dialogue box will open asking whether you wish to optimize your analysis for speed or minimal memory. Choose 'Speed', unless your project uses extremely large data files or you have had problems previously with the 'Speed' option. This has the same effect as setting `par.analysis.conserveMemory=true` in the setup file, as discussed in Section 8.16.

5. The analysis window will start and you should see a dialogue box asking if you wish to execute an init file, or other file, to load any additional paths. This is the same dialogue box shown in Figure 12.1. Recall that Qubist loads *only* the project directory pointed to by the launchQubist.m file, as discussed in Section 4.6. *Even subdirectories of the project directory are not loaded automatically.*

   Usually, you load additional paths by including 'addpath' lines in the init file, but you can also load paths from any other file that is called by the init file. If you require additional paths, click 'yes' in the dialogue box, which will start a file navigation window that will allow you to select your path-loading file. Navigate to your file and click 'Open' if you need to do this. Note that if your additional paths are defined in your init file, then running this file will load them, but the *extPar* structure normally generated by init *will not be passed to Ferret or Locust* (see Section 4.6). The *extPar* structure will be recovered from History files instead. For this reason, I usually put path-loading instructions in a separate file that is called by the init file, so that I do not have to run the entire init file just to load a few paths.

6. The analysis window will open and the analysis procedure will start automatically.

Note that Analyze History tool displays instructions in the MATLAB command window throughout this process, and also in the directory navigation window for some operating systems.

12.3 The OptimalSolutions Viewer

An OptimalSolutions.mat file contains the full set of results from a Ferret or Locust run, but is usually of modest size, because it contains only the best solutions condensed from the much larger set of History files used to generated it. Because of their importance and portability, it is useful to have a separate lightweight viewer that can load OptimalSolutions.mat files directly for visual analysis, without requiring the rest of the Qubist package. The OptimalSolutions Viewer tool is part of the Qubist package, but is also available in the *QubistFreeTools* package. This allows you to do your run and analysis on a fast workstation or server, copy a single OptimalSolutions.mat file to your laptop or send it to a colleague, and examine the results offline.

The Qubist OptimalSolutions viewer is designed to load a saved OptimalSolutions.mat file into the same visualization environment discussed in this chapter, without re-analyzing the History files. The OptimalSolutions viewer is integrated with the Qubist package and is normally started by the component selector.

The following steps are required to use the OptimalSolutions viewer:

1. Start the viewer from the component selector (Figure 3.3) by selecting 'View OptimalSolutions' if you are using the full Qubist package, or by running the 'OptimalSolutionsViewer' command if you are using the *QubistFreeTools* package.

2. A dialogue box will pop up asking you to load an OptimalSolutions.mat file, and instructions will be given in the MATLAB command line window. Navigate to your OptimalSolutions.mat file and open it.

3. A dialogue box will open asking you to select the project directory associated with your OptimalSolutions.mat file. Navigate to your project directory, if it is on the file system of the computer that you are using for visualization, and click 'Open'. You can click 'Cancel' if you just want to view your solutions, but do not want to do any further analysis or custom graphics. Loading a project directory here simply adds the project directory to the path, which allows you to use a custom 'myPlot' function (see Sections 4.10 and 8.20) if you have defined one, or if you want to polish your solutions.[1]

4. The Ferret or Locust Analysis window will open, with your OptimalSolutions.mat file displayed.

5. A dialog box will open asking if you wish to execute an init file, or other file, to load any additional paths. This is the same dialogue box shown in Figure 12.1. If your init file, or some other file, loads paths (including sub-directories) required for your project, you should click 'Yes'. This will open a final dialogue box, which will allow you to select the required file. Canceling here will not affect the visualization of the results, although your 'myPlot' function will not display any graphics, and polishing won't work if other paths are required.

6. You can now use all of the visualization features discussed in Chapter 5, with the exception that the 'Polish' and 'Stop Polish' buttons will be grayed out if you are using the free visualization package.

12.4 The Merge OptimalSolutions Tool

Ferret and Locust are extremely reliable, but they *are* fundamentally stochastic global optimizers, and that means that things can go wrong occasionally. Therefore, it is perfectly reasonable to perform multiple runs with identical settings as a check, especially if you are doing your final production runs in the final stages of a project. Alternatively you might want to vary your settings between runs to see how your results are affected, but merge all of these different runs together to obtain your final optimal set. Analyzing each of these runs will result in an independent OptimalSolutions.mat file, but the set of OptimalSolutions.mat files can be merged together using the Qubist 'Merge OptimalSolutions' tool.

Two or more OptimalSolutions.mat files can be merged by following these steps:

1. Start the viewer from the component selector (Figure 3.3) by selecting 'Merge OptimalSolutions' if you are using the full Qubist package, or by running the 'mergeOptimalSolutions' command if you are using the QubistFreeTools toolbox.

[1] Note that polishers are not included in the QubistFreeTools package.

2. A dialogue box will pop up asking you to select an OptimalSolutions.mat file. Navigate to the first one that you wish to merge and click 'Open'. Instructions are given on the MATLAB command line.

3. This dialogue box will pop up again. Navigate to the next one that you wish to merge, and click 'Open'. Repeat this step until all of your OptimalSolutions.mat files have been loaded, and then click 'Cancel'.

4. A dialogue box will open asking you to select the project directory associated with your OptimalSolutions.mat file. Navigate to your project directory, if it is on the file system of the computer that you are using for visualization, and click 'Open'. You can click 'Cancel' if you just want to view your solutions, but do not want to do any further analysis or custom graphics. Loading a project directory here simply adds the project directory to the path, which allows you to use a custom 'myPlot' function (see Sections 4.10 and 8.20) if you have defined one, or if you want to polish your solutions.[2]

5. The Ferret or Locust Analysis window will open, with your optimal merged solutions displayed.

6. A dialog box will open asking if you wish to execute an init file, or other file, to load any additional paths. This is the same dialogue box shown in Figure 12.1. If your init file, or some other file, loads paths (including sub-directories) required for your project, then you should click 'Yes'. This will open a final dialogue box, which will allow you to select the required file. Canceling here will not affect the visualization of the results, although your 'myPlot' function will not display any graphics, and polishing won't work if other paths are required.

7. Ferret/Locust will merge the OptimalSolutions.mat files and re-analyze them as a combined set of solutions. An identical copy of a new file called 'MergedSolutions.mat' will be written alongside *each* copy of the OptimalSolutions.mat files that you loaded. A MergedSolutions.mat file is exactly the same as an OptimalSolutions.mat file except in name; even loading it will produce a data structure called *OptimalSolutions* - *not MergedSolutions* - in the MATLAB workspace. These files work seamlessly with the OptimalSolutions Viewer and even the Merge OptimalSolutions Tool.

8. You can now use all of the visualization features discussed in this chapter, with the exception that the 'Polish' and 'Stop Polish' buttons will be grayed out if you are using the free visualization package.

12.5 The Ferret History Explorer

The Ferret History Explorer is a simple tool that allows you to navigate your History files and visualize the data that they encapsulate by reconstructing the state of Ferret's console window at any generation of a run. This is sometimes useful for trying to understand how a run evolved, especially if things didn't work out as intended or if something surprising happened. I have often used the History Explorer and the Movie Tool (see Section 12.6) to study the convergence of Ferret's population, which can sometimes provide important insights regarding how to tweak the setup file to improve performance and reliability. The History Tool can also be used for demonstrations, or for obtaining a snapshot of the Ferret window from a previous generation.

The following instructions will allow you to load your project into the Ferret History Explorer:

[2]Note that polishers are not included with the free visualization tools.

1. Start the Ferret History Tool from the component selector window (see Section 3.2).

2. When the History Explorer starts, you will be asked to select a Qubist data directory, which is the directory that contains the History folder and also the OptimalSolutions.mat file, if you have analyzed the run. Don't confuse the data directory with the History directory - the data directory is one directory up from the History directory. Navigate to the data directory and click 'OK'.

3. You will be asked to load your project directory. This is important because the Ferret console window is completely regenerated from the History files. The History Explorer will still work if you skip this step, but the 'User Plot' will probably fail, since Ferret won't be able to find your myPlot function, unless it happens to be on your path for some other reason. The Ferret console window may also appear with the wrong name if you don't load a project directory.

4. A window will pop up asking if you want to execute your init file or another m-file to load additional paths. You should answer 'Yes' only if you need additional paths to run your myPlot function. Your 'User Plot' won't work if it requires additional paths and you don't load them.

Once loaded, the Ferret console window will appear and should automatically load the final generation from your run. A simple user interface will also appear, like the one shown in Figure 12.2. The Ferret History Explorer contains a single slider and text box that you can use to display the Ferret console window from any generation of your run, even if you had graphics turned off during your run. The maximum slider position is equal to the total number of generations from your run. Select the generation to display using either of the following methods:

- Enter a valid generation number into the text box and click the 'return' key, or

- Select a generation using the slider and click the button labelled 'Load & View Saved Generation'.

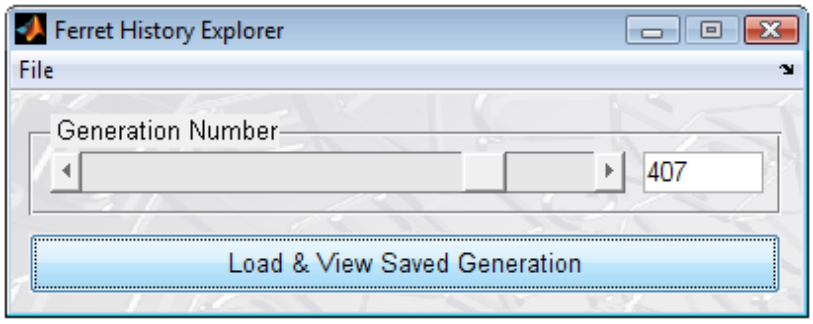

Figure 12.2: The Ferret History Explorer user interface.

The value displayed in the text box is synched to the slider position, so that both components should display the same generation number after a value is entered. Note that the History Explorer does not yet work for Locust, but this will probably change in a future update.

12.6 The Qubist Movie Tool

I have often found it useful to show movies of Ferret runs when giving seminars. I find this especially effective when discussing the code with audiences who are unfamiliar with evolutionary computing ideas, which is a fair description of just about every scientific audience that I have ever spoken to. Section 4.13 discussed a simple technique for generating .avi movies from an active Ferret window. This section introduces the Qubist Movie Tool, which allows you to generate movies of Ferret runs (and other image data) with somewhat greater flexibility, and without having the Ferret window from your run active.

MATLAB contains useful built-in features for generating movies, and I have automated the use of these features in Qubist's Movie Tool, which is designed to make it easy to generate movies from frames that are saved as Ferret runs. Unfortunately, the tool only works with Ferret at this time, but it is likely that I will add support for at least Locust in the near future. Note that the Movie Tool can also stitch together other sequentially numbered image files (i.e. pic_1.jpg, pic_2.jpg, pic_3.jpg, etc.) in a directory, in any format that can be read with MATLAB's 'imread' command.

12.6.1 Running the Movie Tool

Figure 12.3 shows a snapshot of the Qubist Movie Tool. To use the tool on frames saved by a Ferret run, you should first ensure that you have added a line `par.movie.makeMovie=true` to your setup file *before* running Ferret. This will cause Ferret to automatically save a movie frame each generation in the 'movieFrames' folder of the run's data directory. Click on 'Select First Frame', navigate to the 'movieFrames' directory inside of the data directory for your project, and select the first frame of your movie. If you discover that the movieFrames directory is empty, then you probably forgot to include the line `par.movie.makeMovie=true` in your setup file, and you will have to generate your movie from History files instead.

The Qubist Movie Tool can load History files, which is especially useful if you want to generate a movie from a run that did not save any frames (`par.movie.makeMovie=false`). To use this feature, just navigate to the History directory and load the first History file. This will cause Ferret to re-generate its console window for each generation from the saved History information, take a snapshot, and save the snapshots as the frames of a movie. Several simple extra steps will be required to set the desired view and ensure that the 'User Plot' window works, as discussed in Section 12.6.2.

Choose a compressor for your project using the drop-down menu in the Movie Tool. Only compressors supported by MATLAB on your system will be displayed. For example, my MacIntosh doesn't support any compressors, so the only compressor option is 'none'. On the other hand, my Windows computer shows several options to choose from. Let me say at this point that I am definitely not an expert on video formats, and chances are good that many readers will know a lot more about this topic than I do. I cannot recommend one video format over another, and frankly, I've often had to mess around with multiple formats to generate a working movie. My advice is to either consult a video expert or the MathWorks about this topic, or take my approach and play around with different compressors (and possibly different system architectures) to get this to work. Hopefully, you will be able to find a compressor that works on your system and with your AVI player.

There is a slider on the Movie Tool that allows you to select the frame rate in frames per second. Anecdotally, this option is actually the reason that the Movie Tool exists. I had set `par.movie.makeMovie=true` in my project, which causes Ferret to generate an AVI movie automatically at the end of the run, but I was annoyed to discover that the frame rate was too fast for my presentation, and

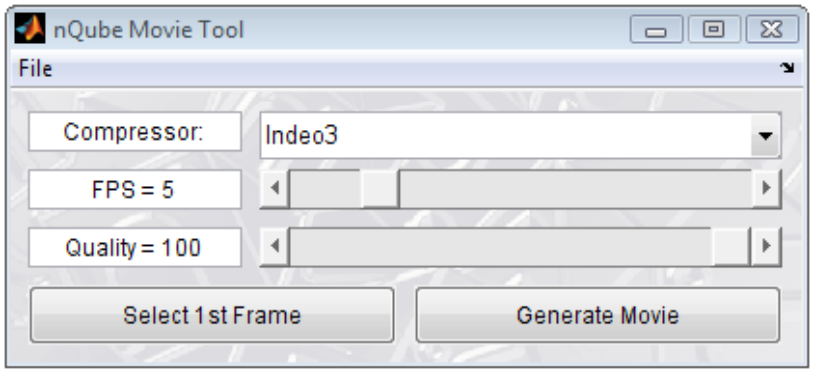

Figure 12.3: The Qubist Movie Tool.

there was no time to re-run the project. I quickly hacked together a short program to re-load the frames and regenerate the movie with a more suitable frame rate. A few days later, I wrapped this program in a nice user interface, and the first version of the Movie Tool was born.

A second slider allows you to select the quality of the movie, which is a number ranging from 10 to 100. Higher quality numbers correspond to larger files and nicer looking movies. I normally select the maximum value of 100.

Finally, make your movie by clicking the 'Generate Movie' button. It should appear in the project's data directory, alongside the OptimalSolutions.mat file, with a name derived from the name of the movie frame directory. For example, if your movie frame directory is called 'movieFrames' (Ferret's default), then your movie will be called 'movieFrames.avi'. You should see a message indicating that your movie was generated successfully and its location on your disk.

If your movie does not look the way you want, just change the compressor, frame rate, and quality settings and try again. The Movie Tool *should* be able to delete the old movie file and replace it with the new, modified one. However, this has been problematic on occasion, especially on Windows computers. If you see an error message about not being able to write the movie file, you may need to delete the old file manually before continuing.

12.6.2 Making Movies from History Files

A few extra steps are required after clicking the 'Generate Movie' button if you are using the Movie Tool to generate movies from History files rather than saved frames. These steps are as follows:

1. You will be asked to choose the project directory, which is the directory that contains your fitness function. Navigate to your project directory and click 'Open'. The Movie Tool should make a reasonable guess about the location of your project directory, as long as you have not moved your files since the run was done. This step is required only to provide paths for your 'myPlot' function (see Section 4.10), which will be called (if it exists) while Movie Tool generates frames. If you don't have a myPlot function or don't care if it is included in your movie, then you can skip this step by clicking 'Cancel'. The Ferret console window will open.

2. You will be asked if you want to execute your init file or another file that loads paths. This is also done to ensure that the Movie Tool has all of the path information that it needs to execute your myPlot function. Click 'Yes' and navigate to your path-loading file if you have one, or click 'Cancel' if you don't need to do this.

3. You will be presented with a message box asking you to 'Set the desired view and click 'OK' to start capturing frames.' At this point, most of the controls on the Ferret Console window will be active and you can set your axes to the desired projection, select whether you want to see the 'Linkage', 'C.P.D. Plot', or 'User Plot' view (Usually the User Plot is most informative for movies, if you have written a good myPlot function), and even select which file you want to display in the 'Project Notes/Info' box. Set your desired view and click 'OK' to start capturing frames. Let this run until the end of the run is reached, or you can stop it prematurely by clicking the 'Stop Run' button. You should get the usual message that your movie has been successfully generated and the name of the file that was saved.

Chapter 13

Concluding Remarks

> *'The world is a thing of utter inordinate complexity and richness and strangeness that is absolutely awesome. I mean the idea that such complexity can arise not only out of such simplicity, but probably absolutely out of nothing, is the most fabulous extraordinary idea. And once you get some kind of inkling of how that might have happened, it's just wonderful. And... the opportunity to spend 70 or 80 years of your life in such a universe is time well spent as far as I am concerned.'*
>
> -Douglas Adams, from Richard Dawkins' *Eulogy for Douglas Adams*

I think Douglas Adams had it right with this quote - the origin of complex structure and behaviour in a universe governed by rules that usually turn out to be simple at a fundamental level, is a deep mystery that deserves contemplation and wonder. This quote is one that resonates strongly with me and is really at the heart of my fascination with global optimization. The most amazing thing in the world to me is that complex and perhaps even intelligent behaviour can develop from simple interacting parts, as an emergent property of the system. Ferret, Locust, and Anvil are all made up of simple parts that interact by simple rules:

- individuals mutate, compete, and recombine via genetic operators for Ferret,
- particles interact by swarm dynamics in Locust,
- search paths obey statistical physics in Anvil, but interact via genetic operators.

These programs aren't nearly as complex as real systems found in nature, but they are complex enough that new and sometimes unexpected behaviours emerge as a global property of the system. Unlike most natural systems though, you can tinker with the rules of a computer program to try to get the emergent behaviour that you want - in this case thorough exploration of the parameter space and global optimization.

Actually finding and refining the simple rules at the heart of these optimizers was a subtle and often counter-intuitive business that required what seemed like an endless sequence of numerical experiments. There is no formula or deductive process that can reliably tell you how the large-scale emergent behaviour of these systems will react when you change the rules governing how its component parts interact, but over

time it is possible to gain some intuition for how things work. I hope that you will gain similar intuition by playing with the setting of the optimizers and trying them on a variety of problems.

The Qubist package has been my passion for the last seven years and has greatly exceeded my wildest dreams of where this might lead. If someone had asked me in 2002 how far this would go, I would have expressed enthusiasm for a novel class of optimizers that I had recently become interested in, and my hope that they would solve the data-modeling problem that got all of this started. But I would not have guessed that it would lead me *here* or that my first simple, awkward genetic algorithms would develop into anything on this scale. Qubist, and especially Ferret, has been the source of great moments of insight, uncountable late nights, and at times enormous frustration. There have been many occasions when the technical obstacles seemed insurmountable, but the joy of an occasional deep insight exceeded the pain of numerous dead ends. Overall, I have had fun developing these tools. I persevered and I think things worked out well.

Writing this user's guide was a major project of its own. I started writing something like a manual or user's guide approximately five times since 2006, and abandoned all attempts except for the final one that led to this guide. In all honesty, I find developing the software and my academic research vastly more enjoyable than writing documentation, and I was too often distracted from this task by developing new features or fixing problems in the code. Moreover, I have juggled the Qubist project with my academic position at the University of Manitoba. All too often, other things had to take priority, and this user's guide was unfortunately at a lower priority than maintaining the code itself when my day job got demanding.

The time was right for this project in 2009, when I felt that Ferret-4's development had leveled off somewhat for the time being, and I was reasonably happy with the other optimizers in the Qubist package. I 'bit the bullet' and wrote most of this text during the summer of 2009. The biggest revelation of this process was that I had all but forgotten many features of my own software. I rediscovered, checked, critiqued, and in some cases fixed dusty old corners of the code that I had not looked at in years. I also critically reviewed newer components like Anvil and Locust. Anvil only required moderate streamlining, but I thoroughly overhauled Locust when I realized that it was overly complex and contained some features that were of uncertain value. This overhaul also resulted in significant new insights that dramatically improved the code, which ultimately resulted in Locust-2. The net result of this user's guide on Qubist is a more streamlined, unified, and thoroughly error-checked software package that I can distribute with pride and confidence.

I have many people to thank for their support throughout the development of Qubist and the writing of this guide. Dr. Jim Hesser's encouragement and boundless enthusiasm were especially inspiring during the early days of the project, when I was a Plaskett Research Fellow at the National Research Council Dominion Astronomical Observatory (DAO) in Victoria. I am also thankful to the Millimetre Astronomy Group at the DAO, especially Paul Feldman and Russell Redman, who provided data and helped pose the tough data-fitting problem that got me thinking in this direction back in 2002. I have been treating Qubist as professional, commercial software since 2007, when nQube was formed. However, Ferret could not have developed to its present level without the many comments and bug reports from my colleagues, students, and friends at the University of Manitoba. I especially thank my Ph.D. student Adam Rogers and summer research student Andrew Cull, whose ability to break my software is only exceeded by their diligence in sending well-documented bug reports. Another summer research student, Blair Cardigan Smith, worked with me on an especially difficult data-modeling problem and really helped to probe Ferret's limitations, which I strove to expand. I thank my colleagues Dr. Boyd McCurdy and Dr. Peter Potrebko at CancerCare Manitoba for their enthusiastic collaboration on the most important and inspiring application of Qubist to date in the field of cancer radiotherapy. Finally, I thank my friend and former student Stasi Baran for reading this entire guide, providing excellent editorial comments, and correcting many errors in the text.

Having just finished writing a fairly lengthy user's guide, I have taken a complete inventory of the Qubist package. Perhaps it is not surprising that this would lead to a comprehensive 'bird's eye' view of the entire code, which I have not experienced in some time. As Qubist's sole developer, I am sometimes guilty of focusing too much on the trees and not enough on the forest. This package has developed by the accumulation of many small insights, plus a few larger ones, and a staggering amount of hard work dealing with the intricate details of this complex software. However, I have been asking myself the 'big picture' questions lately - especially the biggest of them all: "So where will Qubist go from here?" I have many ideas in this regard, but this is not the place to reveal them in detail. I certainly anticipate ongoing core development of the Qubist package, perhaps with greater focus on particle swarms and visualization, and perhaps even greater emphasis on applications and collaborations in academia and industry. What I can say with certainty is that Qubist has taken on a life of its own, and development will continue vigorously for the foreseeable future. I am more excited about this work now than at any time in the past, and I will continue to push Qubist forward with all of my ability, to bring previously intractable problems in optimization and data modeling into the realm of that which is technically feasible.

Bibliography

Eberhart, R. C. (2001) *Swarm Intelligence.* Morgan Kaufmann

Fiege, J. D., Wiegert, T., English, J. (2007) *Astrophysical Journal, in preparation*

Fiege, J. D., Johnstone, D., Redman, R. O., and Feldman, P. A. 2004, Ap.J., 616, 925-942

Fonseca, C.M. & Fleming, P.J. (1993) *Genetic Algorithms: Proceedings of the Fifth International Conference.* also http://citeseer.ist.psu.edu/fonseca93genetic.html

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

Goldberg, D. E. (2002) *The Design of Innovation: Lessons From and for Competent Genetic Algorithms.* Norwell, MA: Kluwer Academic Publishers

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

Horn, J., Nafpliotis N. & Goldberg, D.E. (1994) *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence.* also http://citeseer.ist.psu.edu/horn94niched.html

Poli, R., Kennedy, J., and Blackwell, T., 2007, Swarm Intelligence, 1, 33

Index