Benchmarking Evolutionary Algorithms: The Huygens Suite

Cara MacNish

School of Computer Science & Software Engineering
The University of Western Australia
35 Stirling Highway, Nedlands 6009
+61 8 6488 3453
cara@csse.uwa.edu.au

ABSTRACT

The success (and, importantly, potential success) of randomised population-based algorithms and their hybrids on ever more difficult optimisation problems has led to an explosion in the number of algorithms and variants proposed. It is difficult to definitively compare the range of algorithms proposed, and therefore to advance the field.

In this paper we discuss the difficulties of providing widely available benchmarking, and present a solution that addresses these difficulties. Our solution uses automatically generated fractal landscapes, and allows user's algorithms written in any language and run on any platform to be "plugged into" the benchmarking software via the web.

Categories and Subject Descriptors

G.1.6 [**Numerical Analysis**]: Optimization – *global optimization*;
G.4 [**Mathematics of Computing**]: Mathematical Software - *certification and testing, efficiency;*
I.2.8 [**Computing Methodologies**]: Artificial Intelligence - Problem Solving, Control Methods, and Search;
I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling - *curve, surface, solid and object representations.*

General Terms

Algorithms, Measurement, Performance, Standardization.

Keywords

Benchmarking, test suite, search, optimisation, evolutionary algorithms, particle swarm optimisation, memetic algorithms, hybrid algorithms, fractal landscapes.

1. INTRODUCTION

The success of population-based optimisation algorithms, such as genetic and evolutionary algorithms, swarm algorithms and ant colony optimisation, along with the steady increase in accessible computing power, has allowed previously impractical search and optimisation problems to be addressed across many application domains. This has led to an explosion in the number of algorithms, or variants of algorithms, proposed by researchers. In addition there has been renewed interest in (stochastic or non-stochastic versions of) more traditional algorithms, such as hill-climbing and best-first searches, as well as so-called *memetic* algorithms which combine the distributed and traditional approaches.

Effective means of comparing the multitude of proposed algorithms are limited. This in turn makes it difficult for the field to progress, since it is difficult to evaluate the relative merits of proposed algorithms and modifications.

In this paper we discuss the difficulties of providing widely available benchmarking, and present a solution that addresses these difficulties. Our solution uses automatically generated fractal landscapes, and allows user's algorithms written in any language and run on any platform to be "plugged into" the benchmarking software via the web.

In Section 2 we discuss some of the difficulties that have inhibited the broad adoption of benchmarking techniques for evolutionary and related algorithms, and begin to motivate our solution. Section 3 discusses the problem domain used in our benchmarking suite, and the design decisions that enabled its implementation. Section 4 describes the architecture that is used to overcome the practical issues of a widely available benchmarking server. Section 5 concludes the paper.

A test version of the benchmarking system is available at the time of writing at http://karri.csse.uwa.edu.au/cara/huygens/.

2. DIFFICULTIES IN UNIVERSAL BENCHMARKING

2.1 Practical Issues

There are a number of practical issues that inhibit the development and adoption of universal benchmarking facilities. We now discuss some of these and how we approach them.

- **Programming Languages and Architectures**

Researchers develop implementations for testing their algorithms in programming languages that they are familiar with, and perhaps have existing tools that they can build on. Problem sets to test the algorithms tend to be developed in the same language, and where they are not, specific interfaces are built to bridge the two. Where

Genetic and Evolutionary Computation Conference (GECCO)'05,
June 25-29, 2005, Washington, DC, USA..
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

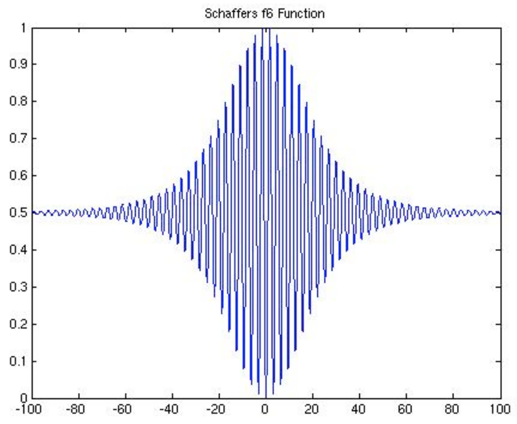

Figure 1. A cross section of Schaffer's F6 function.

other researchers use different languages or interfaces, it often requires many hours of programming and testing, and in many cases rewriting in a new language, in order to compare algorithms on the same problems. This is made more difficult by the fact that researchers often do not have the resources to extensively document their code.

Similar problems exist for different hardware platforms and associated operating systems. This is particularly true when executables are written for proprietary systems.

In this paper we present a novel approach to this problem that uses XML-RPC to separate the benchmarking code and its platform from the user's code and platform. This allows the user to develop in the language and architecture of their choice. This solution is detailed in Section 4.

- **Choice of Problem Domains**

Optimisation algorithms are developed and themselves optimised for a huge range of problem domains. Many of these are not well known or understood, and in some cases they are proprietary and cannot be released. In order for benchmarking problems to be widely accepted they must be readily familiarised.

Various attempts have been made to adopt standard problem sets by common use. Well-known examples are those by De Jong [1] and Schaffer [5]. These are typically mathematically described problems that are designed to be in some way challenging to the solver. A typical example is Schaffer's F6 function, shown in Figure 1.

While a focus on these problems has raised many important issues, they are arguably not very representative of naturally occurring optimisation problems. F6, for example, is difficult because the closer a candidate solution gets to the global minimum, the bigger the hill that must be climbed or "jumped" to move from one local minimum to the next. However we are not aware of any naturally occurring phenomenon which clearly has this property. The importance of choosing benchmark problems that are as representative as possible of a broad range of applications is discussed further in Section 2.2.

In this paper we choose (fractal) landscapes as our problem domain. These domains are appealingly simple and intuitive as a class of benchmarking problems. We would also argue that, while it is impossible to choose a domain that demonstrates all difficulties to which optimisation algorithms are applied in practice, there are a number of properties in these domains that generalise to others. These domains are discussed in more detail in Section 3.

- **Ready access to benchmark problems and comparative statistics**

In recent years various benchmark problems and problem generators have been made available on the web. An example is Spears' *Repository of Test Functions* [7] and Spears and Potter's *Repository of Test Problem Generators* [8].

Assuming one overcomes the issues discussed above, it may still not be easy to benchmark an algorithm against those of peers. Comparative results between pairs (or groups) of algorithms tend to be distributed among academic papers. To carry out a comparison it is typically necessary to obtain another author's code, or sufficient detail of their algorithm to reproduce it. Even then comparisons between a large range of algorithms is impractical.

This paper describes a system that allows authors to "plug in" their own algorithm and have it benchmarked against others, with the results available in real time on the web. The way that this is achieved is described in Section 4.

2.2 A Word on the No Free Lunch Theorem

The *No Free Lunch Theorem (NFL)* [2, 9] states, roughly speaking, that no algorithm performs better than any other when averaged over *all* fitness functions. More specifically, for any algorithms *a* and *b*, as well as *a* performs on function f_a, *b* will perform equally well on some function f_b. It has been further shown that the NFL theorem applies to any subsect of functions iff that subset is closed under permutation.

The NFL theorem implies that algorithms cannot be benchmarked to find a best general algorithm averaged over all fitness functions. If an algorithm performs well on a set of benchmarking problems, it cannot be claimed that it is better than any other algorithm other than on that set. (Of course this empirical result does not even *prove* it is better on other unseen problems in that class.)

The NFL theorem, however, takes no account of *structure* in naturally occurring classes of problems. The theorem assumes a uniform probability distribution over fitness functions. Further, it appears that naturally occurring classes of problems are unlikely to be closed under permutation. Igel and Toussaint [2] show that the fraction of non-empty subsets that are closed under permutation rapidly approaches zero as the cardinality of the search space increases. They further show that constraints on steepness and number of local minima lead to subsets that are not closed under permutation. (Similar results are provided for non-uniform probability distributions.)

Thus for naturally occurring classes of structured problems, one might expect some algorithms to perform better than others. (This is intuitively apparent - for example any given "random" search algorithm can be expected to perform badly on most if not all non-contrived problems.)

This does not save our benchmarking suite from a theoretical standpoint. We can be reasonably certain that, providing we take a sufficiently large sample of test problems from the suite, algorithms that perform well on those problems have general properties that will cause them to perform well on the unseen problems in the class (as opposed to just performing well on the seen problems). However to the extent that we cannot show the suite to be representative of larger classes of problems, we cannot make substantiated claims about the algorithms' general purpose performance. As the saying goes, there is no free lunch, but there may be a free appetiser. This suite aims to provide such an appetiser.

3. FRACTAL LANDSCAPES

3.1 Why Landscapes?

The problem domain chosen is finding a minimum in random (but repeatable) automatically generated landscapes. As well as being familiar and intuitively appealing, landscapes exhibit many of the properties that raise difficulties in search and optimisation algorithms.

- Landscapes have a very large number of local minima (sometimes referred to as *highly multimodal*). However these are achieved naturally and randomly, unlike mathematical functions such as F6 mentioned earlier.
- Landscapes are more complex than functions such as F6 in that detail does not diminish with scale. This is representative of naturally occurring problems, which do not necessarily "smooth off" as candidate populations converge to smaller scales. This is particularly important in regard to two issues that arise in poulation-based and hybrid algorithms.

  Exploration versus exploitation

  One of the most difficult issues in population-based methods is managing the balance between *exploration*, in particular jumping out of local wells to search for new minima, and *exploitation*, or convergence of the population to find a value suitably close to the local minimum. In many problems knowledge of scale of the problem or domain allows one to implicitly "cheat" in optimising the parameters of the algorithm for the problem.

  As a simple illustration consider a bowl function, such as appears in De Jong's test suite as F1 [1]. Any reasonable optimisation algorithm will solve this problem very quickly and easily, using either local search (eg. hill climbing) or convergence. This is because prior information is (implicitly) provided that the minimum lies within the given bounds, and therefore convergence parameters can be chosen with the sole implication of determining rate of convergence.

  If we assume, however, that the bowl could be part of a larger picture, the implications of the parameter choice change entirely. Sufficient random action is required to jump out of the well. If the available computing resource is, for example, number of evaluations, we are faced with a very difficult problem of how best to use them. (Consider, for example, you are hiking in a mountain range in the Gibson Desert and looking for the lowest place to start digging for water!)

  Examples where the scale or bounds of solutions are not known occur in other practical domains. Lets say for example we are using an evolutionary algorithm to find the optimal set of weights for an (arbitrary) neural network. Wherever we seed our population, we cannot be certain that the best solution lies within, say, any given hyper-sphere containing the initial population.

  Population search versus local search

  One approach to dealing with the exploration/exploitation problem has been to propose hybrid (or memetic) algorithms combining population-based methods with local search. Again, when the scale at which the detail occurs is known or bounded, this can make these algorithms perform above their general potential.

  To continue the bowl illustration, consider say an egg box (or muffin tray). It would be very easy to parameterise a hybrid algorithm so that the population operators search across the egg cups, while local search is used to rapidly find minima within the cups. This may report excellent results for this problem, but perform extremely poorly when the number of minima ("cups") within each cup is increased - suddenly the scales at which the population-based and local search are operating are inappropriate.

Landscapes are considered to have (or at least in some sense approximate) the fractal property of statistical self-similarity. As one zooms in on, say, a mountain range, rather than reaching a scale in which the landscape is smooth, successively more detail reveals itself.

Since we seek to find algorithms that do well in general rather than on more specific problems, we use problems in which the level of detail is maintained as scale decreases (and indeed while scale increases, if the algorithm is initialised at an arbitrary scale). Algorithms that do well on these problems should generalise easily to problems with more limited scale.

3.2 Midpoint Displacement Algorithms

Probably the most widely used algorithms for random fractal landscape generation are *midpoint displacement algorithms*. These are iterative algorithms that successively randomly perturb a shape at finer and finer scales. In the 2-dimensional case the midpoint of each line segment is found and perturbed up or down by a random amount, creating twice as many line segments for the next iteration. Similarly, in the 3-dimensional case the midpoint of a square is perturbed, creating four new "squares".

This approach has three major drawbacks from our point of view. First, it is difficult to justify as a model of landscape formation, bearing little or no relationship to natural geological processes. The second, arguably minor issue (assuming the landscape can be generated to arbitrarily fine resolution), is that the depths of points between the perturbed midpoints must be "guessed" by some form of interpolation. While the simplest would be linear interpolation, this is difficult to justify geographically.

The third and most important problem, however, is that of storage. The number of midpoints that must be stored increases exponentially with the depth of iteration. In the 2 dimensional case the number of midpoints increases with 2^n, while in the 3 dimensional case it increases with 2^{2n}. This may not be a big problem if landscapes are being generated for human viewing, such as in a film, since the resolution of the human eye is quite low. However if we wish to generate fractal landscapes with detail to the limit of the resolution that can be achieved on a standard computer (which we take to be 64 bit IEEE floating point) the files describing the landscapes would be far too large to store.

3.3 Our Approach

Our approach to generating landscapes is based on a (grossly) simplified model of the natural process of meteors impacting a

Table 1. Increasing boulder numbers with reducing scale.

Scale	Boulder size	Size of square with mean *n* boulders	Mean total no. boulders
1	0.1-1	1x1	*n*
0.1	0.001-0.1	0.1x0.1	100*n*
0.01	0.0001 -0.001	0.01x0.01	10 000*n*
10^{-3}	10^{-4}-10^{-3}	10^{-3}x10^{-3}	10^6*n*
...	...	...	...

moon (or planetary) surface. We assume a new moon begins as a sphere, and each meteor or boulder that hits it leaves a crater the size of the bottom half of the boulder.

While we don't have space in this paper for a detailed description of our algorithms, in the following we cover the main points.

3.3.1 A Suite of Landscapes

Using a single landscape for benchmarking would reward optimisers that are overly specific (in choice of parameters) and may generalise poorly to other landscapes (and other domains). In other words, it would encourage users to "overtrain" their algorithms.

Our approach is to generate a suite of landscapes. Users are given a number of test landscapes on which to hone their algorithms, but the benchmarking itself is carried out across a range of unseen landscapes.

3.3.2 Randomness and Scale

The suite of landscapes must be randomly generated. For language independence we use a portable (multiplicative congruential) pseudo-random generator [4]. The landscapes must also be repeatable. We therefore ensure an entire landscape can be generated from a single seed.

The seed provides a convenient way of indexing a sequence of landscapes. However sequential seeds produce related random deviates in multiplicative congruential algorithms. We therefore scatter or hash the seed (or index) to produce that actual starting point of the random number generator for each landscape.

While the positions of boulders can be chosen from a uniform distribution, the number of boulders of each size must be chosen so as to preserve statistical self-similarity. One way to put this is that for each order of magnitude square on the surface, we would expect, on average, the same number of boulders of that order of magnitude to strike the surface. This is illustrated in Table 1.

All landscapes, or *moons*, in the Huygens Suite are parameterised by *n*, the mean number of boulders per order of magnitude square, and the seed or index. For example, Figure 2 shows the landscape of Moon 20_101 (*n*=20, seed 101), the first training example provided for the *n*=20 series at the Huygens Server web site [3].

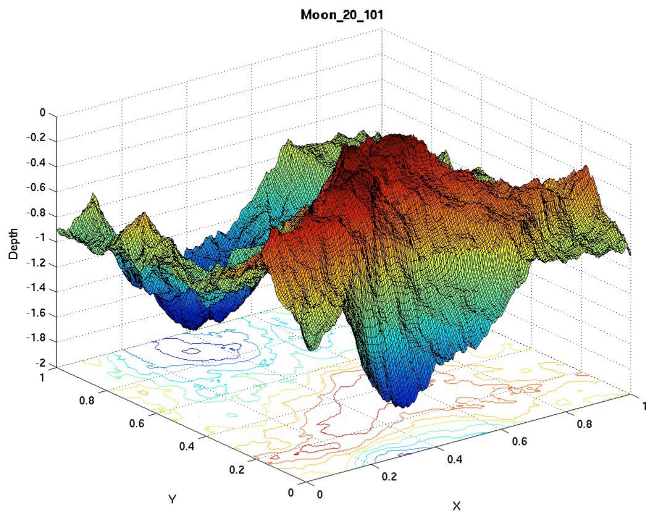

Figure 2. The landscape of Moon 20_101 from the Huygens Suite. Notice the landscape wraps around in both *x* and *y* dimensions.

3.4 Implementation

There are a number of ways that this model can be implemented. We describe the issues that arise in a number of these, motivating the approach taken.

3.4.1 Storing the Landscape

Perhaps the most obvious approach is to generate landscapes by storing the depth of the surface (initially zero) and simulate the impact of the boulders by deducting the boulder dimension from the depth. This allows the landscapes to be generated and stored in advance, and probes of the surface (candidate solutions) to be evaluated very quickly.

The obvious problem with this approach, as with the midpoint displacement algorithms, is that the storage space increases with resolution. Given that we wish to represent detail to the limit of 64 bit floating point, the arrays of points would be far too large to store.

3.4.2 Storing the Boulders

An alternative approach is to store the boulders, that is their centres and radii, and regenerate the landscape as needed. The benchmarking process only requires evaluation at specific points (*x,y* co-ordinates), so it is not necessary to generate an entire landscape (except for viewing purposes). Each time we wish to evaluate a point, we would cycle through all the boulders and determine the effect each one has on that point.

This approach has problems with both storage and running (evaluation) time. It can be seen from Table 1 that the number of boulders increases with the inverse square of the scale. Thus for a resolution of 10^{12} we would need to store more than 10^{24} boulders, and cycle through these each time a point is evaluated. Clearly this is impractical in both time or space.

3.4.3 Regenerating the Boulders

As mentioned earlier we would like each landscape to be generated from a single random seed. This requires that we are able to regenerate the boulders from the seed in a deterministic sequence. It is therefore possible to overcome the storage

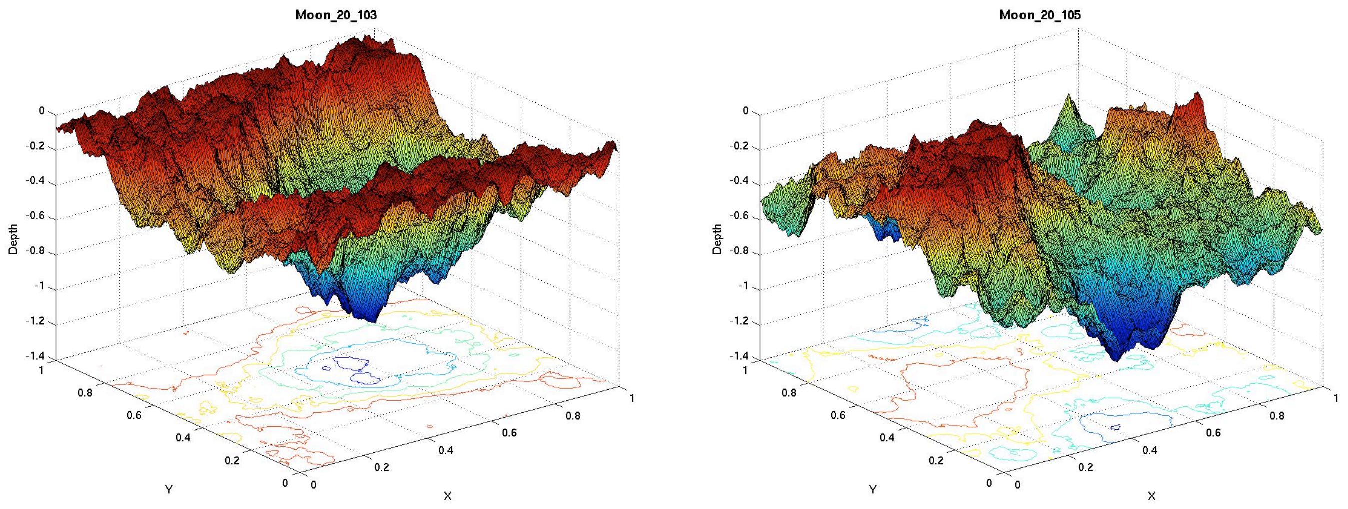

Figure 3. Two additional landscapes from the 20 boulder series showing the distinctively different landscapes arising from small seed changes.

problems referred to above - we simply record the seed and regenerate the boulders on each evaluation. The seed and generation algorithm can be regarded as an implicit encoding of the landscape (much as $ax+by+c$ is an explicit encoding of a plane). While this overcomes the storage problem, it clearly exacerbates the execution time problem, and again is entirely impractical.

3.4.4 A Recursive Approach

While the number of boulders increases rapidly with decreasing scale, only a small proportion can impact a particular location being evaluated. As mentioned earlier, the average number of boulders at a given scale centred in a square of that scale is constant. A given location can only be impacted by the boulders centred in the square in which it lies and its eight neighbours. This number of boulders increases only linearly with the exponent of the resolution. If we can find a way of generating only these boulders then we can evaluate any point in a practical time.

This is the approach taken in the benchmarking software. The trick is to associate a recursively generated seed with each square at each scale. That is, the seed associated with a square at scale $10^{-(m+1)}$ is determined by its parent seed at scale 10^{-m}, along with its relative position within that parent square. It is possible to generate these seeds uniquely for all squares to a scale of 10^{-7} before the 32-bit random number generator seeds are exhausted and reuse of seeds is required.

Once the recursive algorithm has generated the seed for a square at a given scale, the boulders at that scale centred in that square can be generated using the pseudo-random generator. It then determines which square at the next scale down the point lies in, and calls the recursive algorithm on this square and its eight neighbours. In practice this can be continued until at least a scale of 10^{-12}, at which point the resolution of 64 bit floating point becomes insufficient.

3.5 Results

The landscapes used in the Huygens Suite are all evaluated to a scale of 10^{-12}. No storage is required as evaluation is done recursively from a single seed. Evaluation time is linear in (the absolute value of) the scale exponent, so the full resolution of the computer is available while execution time remains fast. Thus the technical goals have been met.

We believe the landscapes to be challenging, intuitive, and visually convincing (tho we leave the latter judgement to the reader, who we invite to view a range of sample landscapes at the website). To illustrate the widely varying domains that arise from small seed changes, we have included two further landscapes in Figure 3.

Finally, to illustrate that the fractal self-similarity property is satisfied we have included in Figure 4 six cross sections of the landscape in Figure 2 at successively smaller scales. Both the x and z (depth) scales are reduced by 10 in each successive picture. Notice that there is no systematic change in the amount of detail (number of local minima, height of peaks, etc) as one progresses through the sequence. Indeed the images could be shuffled in any order without looking out of place.

3.6 Whats in a Name?

The benchmarking suite in this paper is named after the Huygens Probe, which recently made a successful landing on Saturn's moon Titan. (The probe is in turn named after Christiaan Huygens, the astrologer who discovered Titan.)

The analogy with the crater-based landscapes developed in this paper is obvious. However the choice of Huygens (rather than say Titan) is more specifically motivated.

First, when developing a benchmarking methodology, the computing resource to be optimised must be chosen. In evolutionary algorithms a wide variety have been used. Examples include number of epochs or iterations to reach the global minimum (which in our case is unknown) to within a given accuracy, best value achieved in a given number of evaluations, average population fitness, through to environment dependent measures such as CPU time.

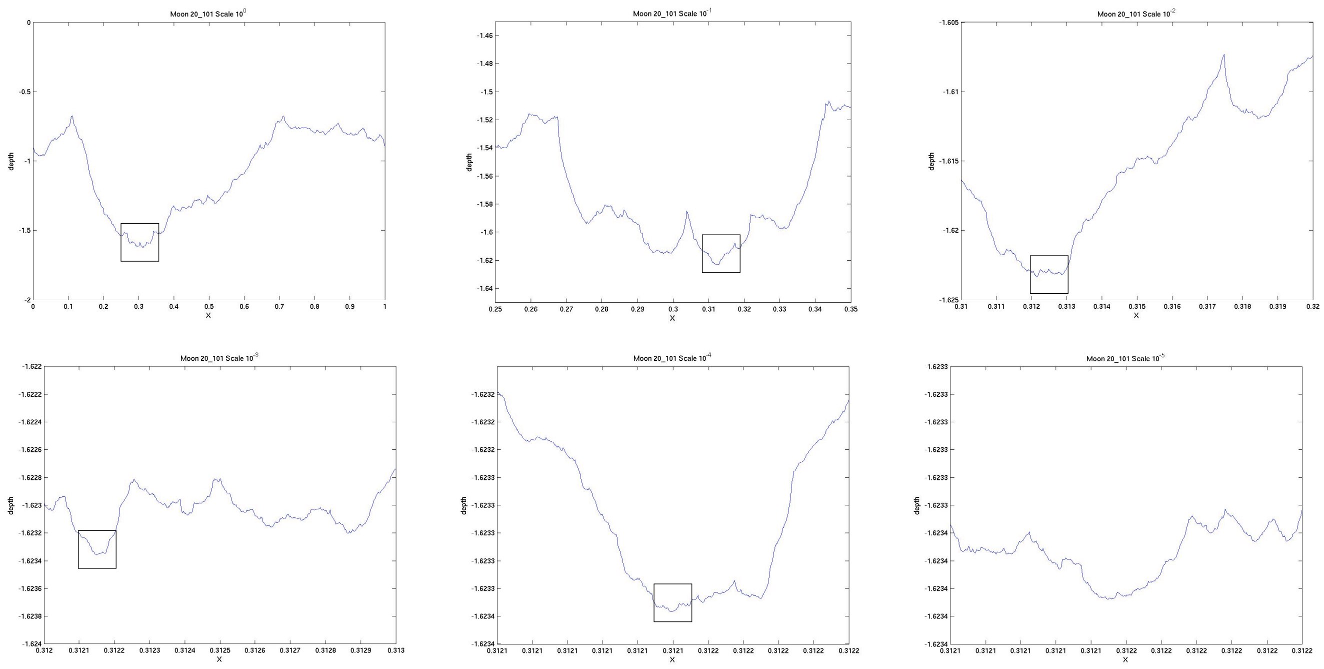

Figure 4. A one dimensional slice of terrain 20_101 shown at magnifications of 10^0 to 10^6. The square shows the detail in the subsequent plot.

In order to allow comparison of a very broad range of algorithms, including hybrid and traditional search algorithms, we have steered clear of paradigm-specific concepts such as epochs. Similarly to support multiple computing environments we have steered clear of enviroment-dependent measures. We have taken the view that in most practical applications, as with the Huygens Probe, the expensive operation is evaluating the fitness at a given location. We therefore allow each algorithm a fixed number of evaluations, or probes, for each landscape to produce its best solution. (Further, in the non-training case, the algorithm does not know which landscape it is solving at any given time, so probe information cannot be accrued across multiple attempts.)

The second analogy is that the probes are sent to the surface from a remote location and the data sent back. The reasons for this are given in the next section.

4. THE HUYGENS SERVER

Now that we have addressed the content of the benchmarking suite, we address the question of making it readily accessible to all users.

4.1 Design

The benchmark server is designed according to the following principles:

1. Access to the benchmarking software and results should be freely available through the web.
2. Use of the software should be independent of the user's computing platform.
3. The user should be able to develop his or her algorithm in the programming language of his or her choice.
4. The user should be free of any need to understand (or even see) the benchmarking code.
5. The user should be able to initiate the benchmarking process. (For example, the user should not have to submit an algorithm and rely on a human at the "other end" to run it.)

These principles are achieved by the structure illustrated in Figure 5. The fundamental design decision that allows these principles to be satisfied is the separation of the user and benchmarking (server) environments. The user's algorithm runs in the user's own environment. This way the user does not need to download and build the benchmarking code. Nor does he or she have to submit code for a second party to build and run (also avoiding the security risks this entails). It also means the user can benchmark existing algorithms or write new algorithms in the language of his or her choice.

4.2 Implementation

In order to benchmark the user's algorithm, the software must be able to obtain, via the internet, candidate solutions (tuples representing points in the search space) from the algorithm, and return the evaluations (fitness values) of those candidates to the algorithm. This can be achieved using remote procedure calls (RPC).

As one of our goals is language independence, we require a language independent RPC protocol. The protocol we have employed is XML-RPC [10]. This is a relatively simple open standard for sending procedure calls between different environments over the internet. It uses the standard HTTP transport mechanism, and encodes data using XML (plain text).

In addition there is a strong XML-RPC user community, providing clients and servers in a wide range of programming languages (examples include C, C++, Java, php, Perl and Python).

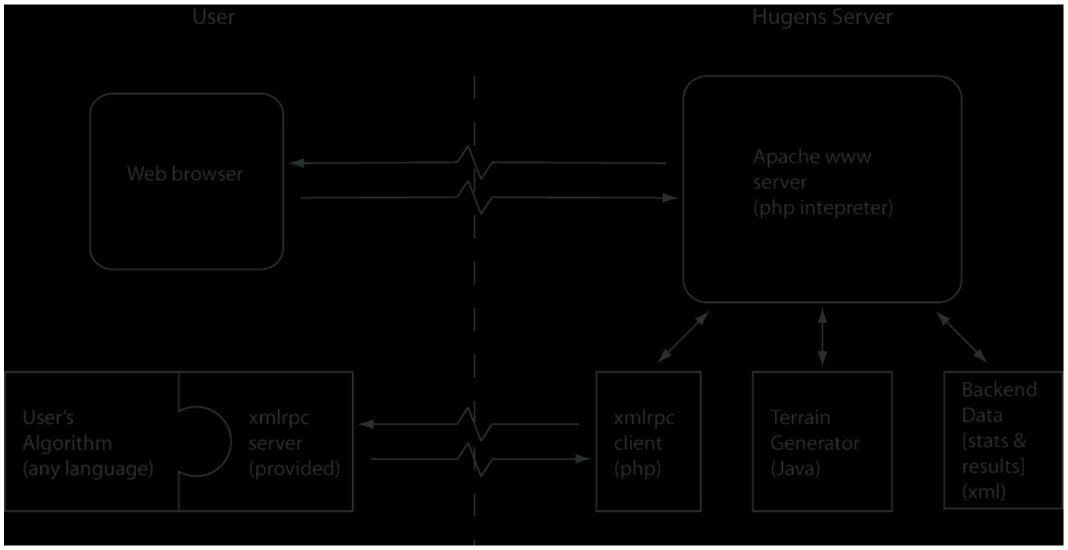

Figure 5. Architecture of the Huygens Benchmark Server.

The user is simply required to "plug in" their own code to the appropriate XML-RPC server as illustrated by the shaded box in Figure 5. To do this the user's code must handle two requests (that is, fill in two methods/procedures/function calls). The first supplies the number of evaluations allowed and requests the first probe (candidate solution) from the user algorithm. The second supplies the fitness value of the previous probe, and requests the next probe.

4.3 Control

The user initiates a benchmarking test via a browser. The Huygens Server then runs the user algorithm using the two RPC calls mentioned above. Once this is complete it updates its record of results based on the user algorithm's performance, and displays the results to the user through the browser.

This achieves the design principles and overcomes the difficulties outlined in Section 2.

5. Conclusion

While arguments such as the No Free Lunch Theorem mean that it will never be possible to identify the "best" general purpose optimisation algorithm, it is nevertheless vital to develop some definitive and widely accessible means of comparing algorithms. This paper presents one attempt at such a system.

We have argued that the benchmarking problems chosen are intuitively simple and appealing, while overcoming technical issues to provide efficient, high resolution fitness functions. We have also presented an architecture for accessing the system that overcomes problems of language and environment incompatibilities. The system allows the user to "plug in" their algorithm and initiate automated benchmarking, while leaving access to the test problems and accumulation of results data under the control of the server.

We hope that this system will provide a valuable resource to the evolutionary algorithms research community.

6. REFERENCES

[1] De Jong, K. A. Analysis of the behaviour of a class of genetic adaptive systems. PhD dissertation, (1975), Dept. Comput. Commun. Sci., Univsity of Michigan, Ann Arbor, MI,.

[2] Igel, C. and Toussaint, M. A no-free-lunch theorem for non-uniform distributions of target functions. *Journal of Mathematical Modelling and Algorithms, (2004),* in print. See also Recent Results on No-Free-Lunch Theorems for Optimization (2003), Los Alamos pre-print.

[3] MacNish, C. Huygens Benchmarking Suite, http://karri.csse.uwa.edu.au/cara/huygens/, accessible as at May 6, 2005.

[4] Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P., *Numerical Recipes in C*, 2nd Ed (1992), CUP.

[5] Schaffer, J. D., Caruana, L. J., Eshelman, L. J. and Das, R. A study of control parameters affecting online performance of genetic algorithms for function optimisation, Phillips Lab., North America Philips Corp., Briarcliff Manor, NY, 1989.

[6] Schumacher, C., Vose, M. D. and Whitley, L. D. The No Free Lunch Theorem and Description Length. In *Genetic and Evolutionary Computation Conference (GECCO 2001), (2001),* Morgan Kaufmann, CA, USA, 565–570.

[7] Spears, W. M. Genetic Algorithms (Evolutionary Algorithms): Repository of Test Functions, http://www.cs.uwyo.edu/~wspears/functs.html. accessible as at May 6, 2005.

[8] Spears, W. M. and Potter, M. A. Genetic Algorithms (Evolutionary Algorithms): Repository of Test Problem Generators, http://www.cs.uwyo.edu/~wspears/generators.html, accessible as at May 6, 2005.

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

[10] XML-RPC, UserLand Software, http://www.xmlrpc.com/, accessible as at May 6, 2005.