Stochastic Optimisation

March 16, 2011

1 Introduction

In this coursework, the problem of global optimisation of Keane's Bump was solved with Biased Monte Carlo Sampling (BMCS) and with Simulated Annealing (SA) algorithms. The performance of the algorithms was investigated, as was the effect of different parameters on the performance.

2 Problem Description

$$\begin{aligned} \text{Maximise} \quad f(\boldsymbol{x}) &= \left| \frac{(\cos x_1)^4 + (\cos x_2)^4 - 2\,(\cos x_1)^2\,(\cos x_2)^2}{\sqrt{x_1^2 + 2x_2^2}} \right| \\ \text{subject to} \quad x_1, x_2 &\in [0, 10] \\ x_1 x_2 &> 0.75 \\ x_1 + x_2 &< 15 \end{aligned}$$

The surface of the feasible region is shown in Figure 1. Since we can see that the global optimum is on the $x_1x_2 > 0.75$ boundary, we can locate it by searching along that curve. This tells us that the optimum is located at $[1.6009\ 0.4685]^{\mathrm{T}}$, where the objective function is 0.36498. There are many other local optima, both on and off the constraint boundaries.

3 Techniques

3.1 Quantifying Performance

To investigate the performance of an algorithm with given parameters, it is run n times with the random seeds $1, 2, \ldots, n$, and each time the best solution found is recorded. To get an impression of the performance, we can consider the mean μ and standard deviation σ of the results. This can be likened to approximating the distribution of the results as a Gaussian, and matching the first and second moments. This approximation can be very bad, as shown in Figure 2, but it serves to indicate the performance of the algorithm. In the cases shown, the Gaussians predict 6% and 38% chances of a result exceeding the global optimum.

If we trust the Gaussian approximation, μ and σ are sufficient to calculate probabilistic bounds for the performance of the algorithm. For example, in 95% of cases the result would be above $\mu - 1.645\sigma$. It would be more accurate to directly estimate the 5th percentile from the data and perhaps this should have been used instead. However, one feature of the Gaussian approach is that it penalises rare, catastrophic failures heavily. For the SA results in Figure 2, the approximated and actual 5th percentiles are 0.338 and 0.358, respectively.

Another advantage of calculating the standard deviation is that it can be used to determine the accuracy of the calculated mean. By the Central Limit Theorem, as $n \to \infty$ the distribution of μ tends to a normal distribution, $N(\mu_{\text{true}},\ n^{-1}\sigma^2_{\text{true}})$. Thus, with a probability of around 95%, $\mu_{\text{true}} \in [\mu - 2\,n^{-1/2}\sigma, \mu + 2\,n^{-1/2}\sigma]$. This gives us a statistical confidence bound when we are drawing conclusions about performance.

Two concerns are ignored in the confidence bound above. Firstly, we assume $\sigma_{\text{true}} \approx \sigma$—this error will hopefully become negligible for large n. Secondly, if we are selecting the results with the highest mean, this will tend to select evaluations with $\mu > \mu_{\text{true}}$ and will skew the distributions of the estimated μ.

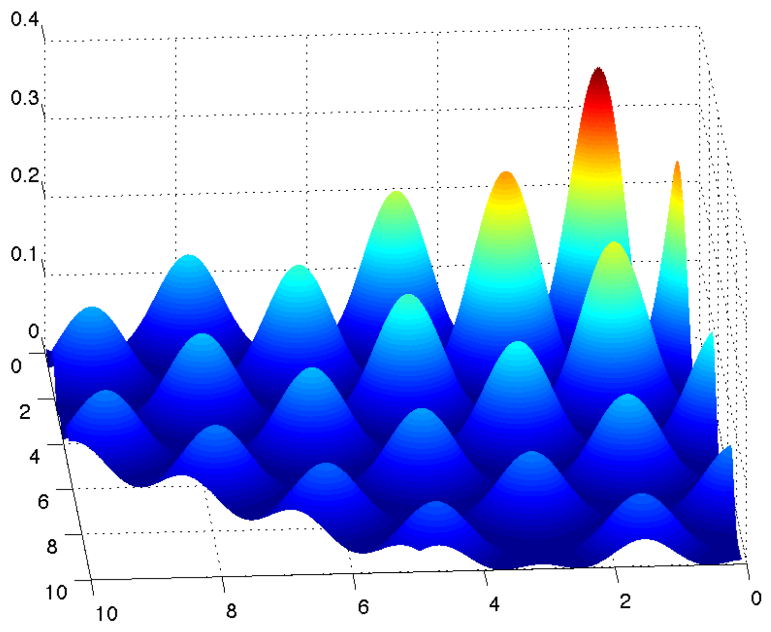

Figure 1: Keane's 2-D Bump

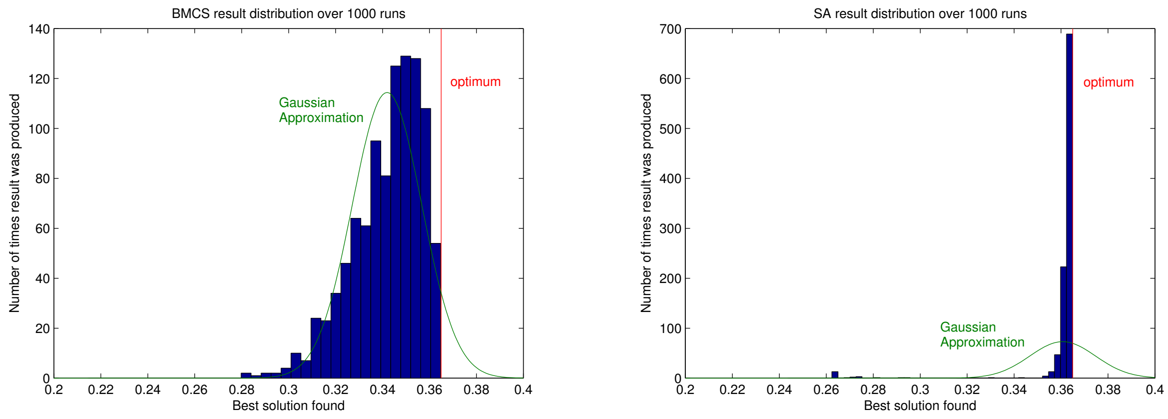

Figure 2: Result distributions and Gaussian approximations

3.2 Visualising Performance

By estimating μ and σ for different parameter settings, we can see how the parameter settings affect the performance, and we can determine the parameters that result in the best performance. This is effectively a multidimensional optimisation of a noisy objective. However, the purpose of this report is not to find the best parameters for this problem, but to gain an understanding of how they affect the performance. For this purpose, we need to be able to see how a single parameter changes the performance, but this is complicated by the settings of the other parameters—Figure 3 shows how complicated this is. To take this into account, for a given setting of one parameter we can choose the other parameters to optimise the performance. This allows us to view the performance as a function of a single variable, which makes it easier to visualise and explain.

More formally, given a performance measure $o(\theta_1, \theta_2, \ldots, \theta_k)$, we consider the k different functions:

$$o_i(\theta_i) = \max_{\theta_1 \ldots \theta_{i-1}, \theta_{i+1} \ldots \theta_k} o(\theta_1 \ldots \theta_k)$$

3.3 Penalty Function

The same penalty function was used in both the BMCS and SA algorithms:

$$\begin{aligned} c(\boldsymbol{x}) = {} & \min(x_1 - 0, && 0) + \\ & \min(10 - x_1, && 0) + \\ & \min(x_2 - 0, && 0) + \\ & \min(10 - x_2, && 0) + \\ & \min(15 - x_1 - x_2, && 0) + \\ & \min(x_1 x_2 - 0.75, && 0) \end{aligned}$$

It is scaled by a configurable weight, given by the `penalty_factor` variable. In general, we could use separate weights for every constraint, but since the variable scales are so similar in this problem, this possibility was not investigated.

3.4 Archiving

Archiving was implemented exactly as described in the lecture notes [Parks, 2011]. The same archiving code was used for the BMCS and SA implementations, and the algorithm implementations themselves are independent of the number and type of archives used. One downside of this low-coupling approach is that an implementation might want to use a dissimilarity archive when restarting the search. This was avoided by having the SA implementation keep track of the single best solution seen so far, and restart from there if necessary.

4 Biased Monte Carlo Sampling

The BMCS algorithm is much like that shown in lectures [Parks, 2011]. Each axis range is divided into m equally sized ranges, for a total of m^2 regions. To determine initial region probabilities it samples a certain number of times per region, then uses the average objective observed in each region to rank them. The linear selection probability relationship is used to translate the ranks to probabilities. 1000 random samples are made according to these probabilities, then the probabilities are recalculated.

The penalty function is used in two ways: firstly to detect invalid solutions and to avoid archiving them, and secondly when calculating the average objective in a region (for determining selection probability).

The following parameters of the algorithm were investigated:

Parameter	Description
`penalty_factor`	A constant weight used for the penalty function
`m`	The number of divisions of each axis
`initial_samples`	The number of samples to take, per region, before calculating region probabilities
`pressure`	The selection pressure, S, used when calculating region probabilities.

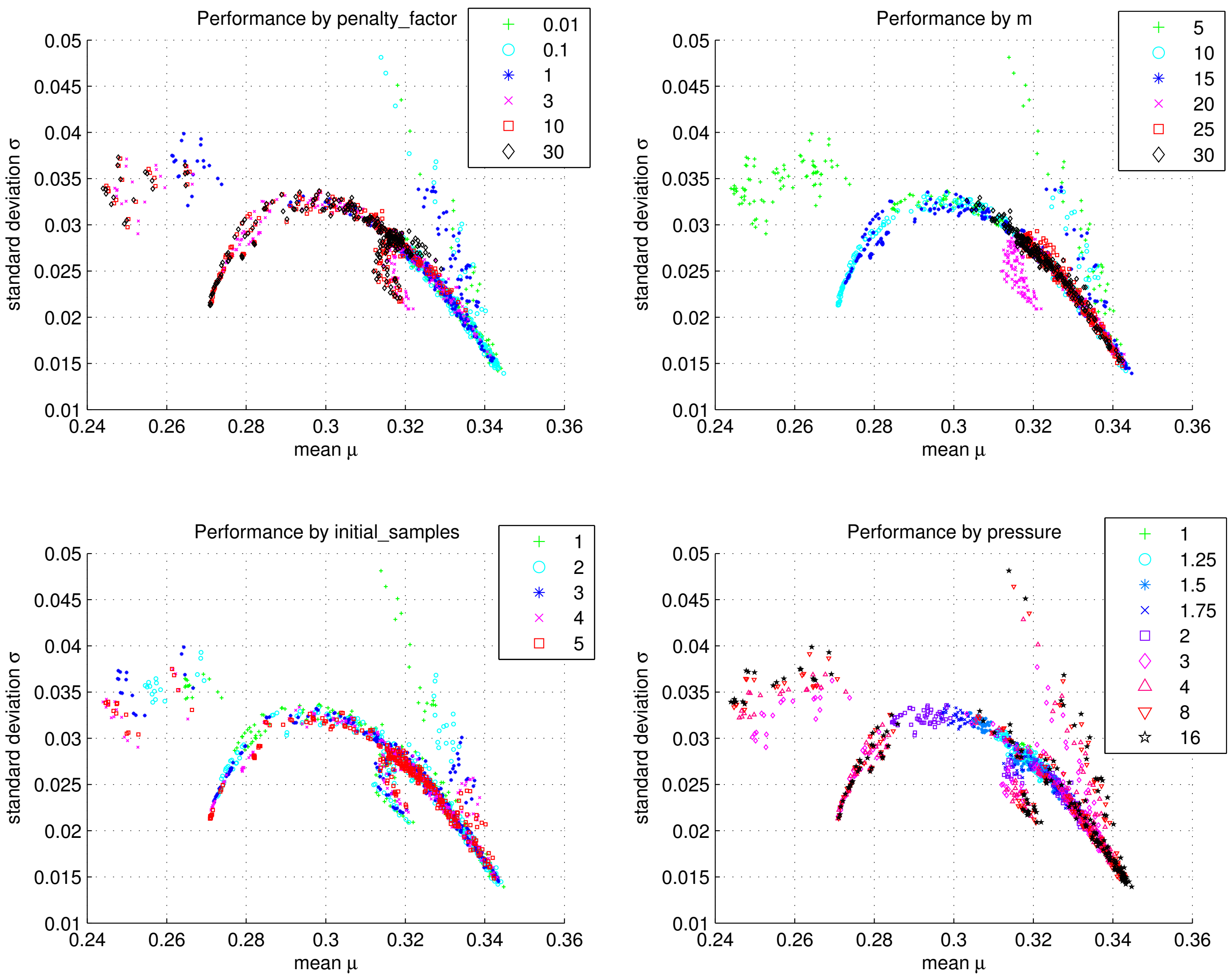

Figure 3: BMCS performance with varying parameters

4.1 Searching the Parameter Space

To investigate the effect of various parameters, the following parameter values were chosen for investigation:

Parameter	Values								
`penalty_factor`	0.01	0.1	1	3	10	30			
`m`	5	10	15	20	25	30			
`initial_samples`	1	2	3	4	5				
`pressure`	1	1.25	1.5	1.75	2	3	4	8	16

The algorithm was run for every combination of these values: a total of 1,620 combinations. For each combination, it was run $n = 1000$ times, as described in Section 3.1. Figure 3 shows the results, and Figure 5 summarises a few of the patterns visible.

These plots are rather complicated. They show some striking patterns (especially the variation with m) but it is not clear what is causing these. Some insights that can be drawn from these plots:

- The large bump visible can be explained by analogy to a Bernoulli random variable, with $\mu = p$ and $\sigma = \sqrt{p(1-p)}$, where the two values of the variable correspond to the two largest optima of the

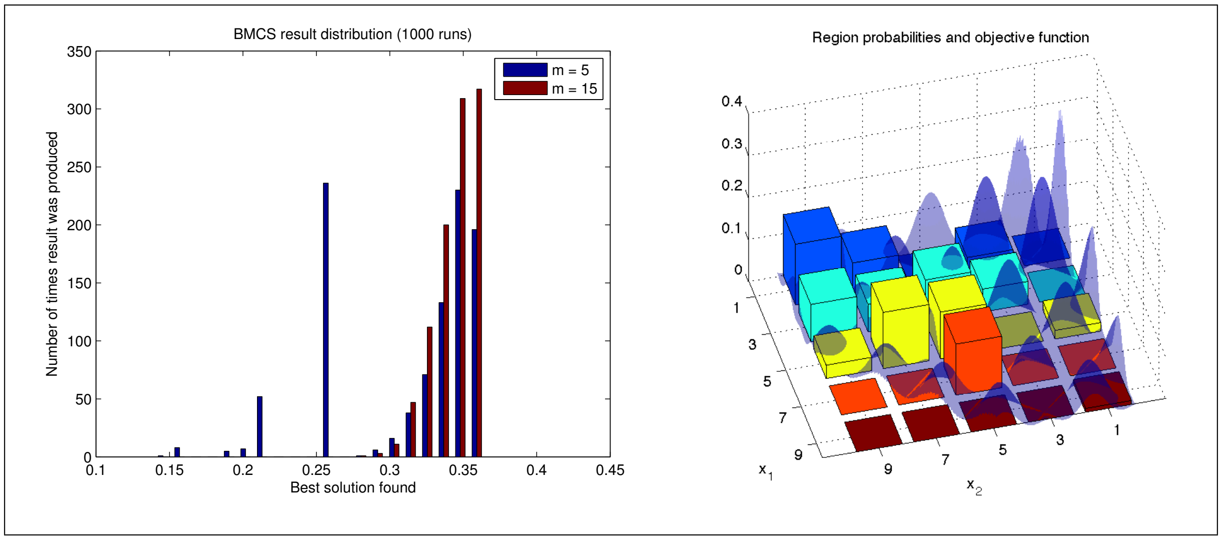

Figure 4: Small regions leading to an unreliable BMCS

function, peaking at 0.263 and 0.365.

- The graph of variation with selection pressure shows that at low pressures the performance varies very little when changing other parameters (the green and cyan dots are all hidden in the densely packed blob).

  At larger selection pressures, (pink, red and black) we start to see interesting patterns like the patch on the left, the main curve, curl coming off it and the "jet" of high variance results above the main curve.

- Similarly, results for smaller regions (larger m) are more tightly clustered on the graph. In this case, however, the regions can be made small without sacrificing performance.

  For larger regions (especially $m = 5$) the performance is very variable, and most of the strange features of the graphs are composed of results for small m.

- Smaller penalty factors appear to give better performance. However, the combination of:

  – Small penalty factor
  – Small m (and thus large regions)
  – Few initial samples
  – High selection pressure

  leads to the very high result variances seen in the "jet". Figure 4 shows why this is: with large regions and few initial samples, there is a high probability that the initial samples represent the objective function in the region badly—the regions contain both optima and zeroes of the function. The high selection pressure means that regions with low average objective functions are completely ignored, and the search never discovers the optimum in the region.

- Interestingly, when the penalty factor is higher, instead of seeing very high variances with high means, we see the lowest means observed but not such high variances. This is the patch of results on the left.

  This is because the penalty reduces the chance that sampling will occur near the constraints, and concentrates the search on the local optima near the centre of the space. Fewer, lower optima lead to a smaller, lower range of possible results, and as such smaller μ and σ.

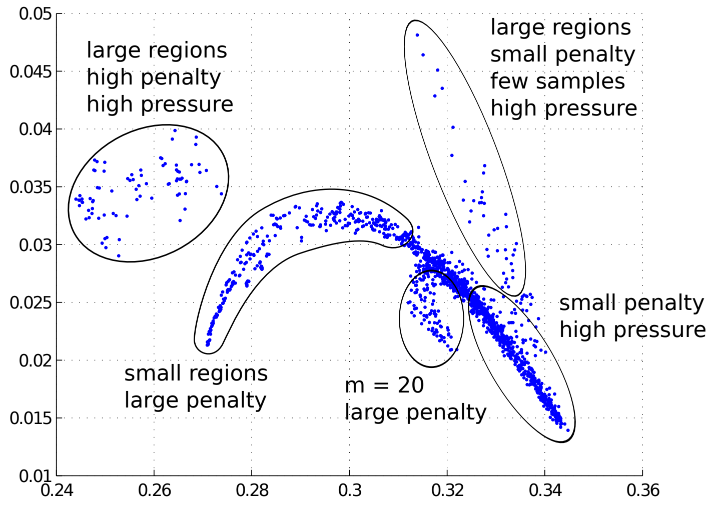

Figure 5: Summary of patterns in BMCS performance

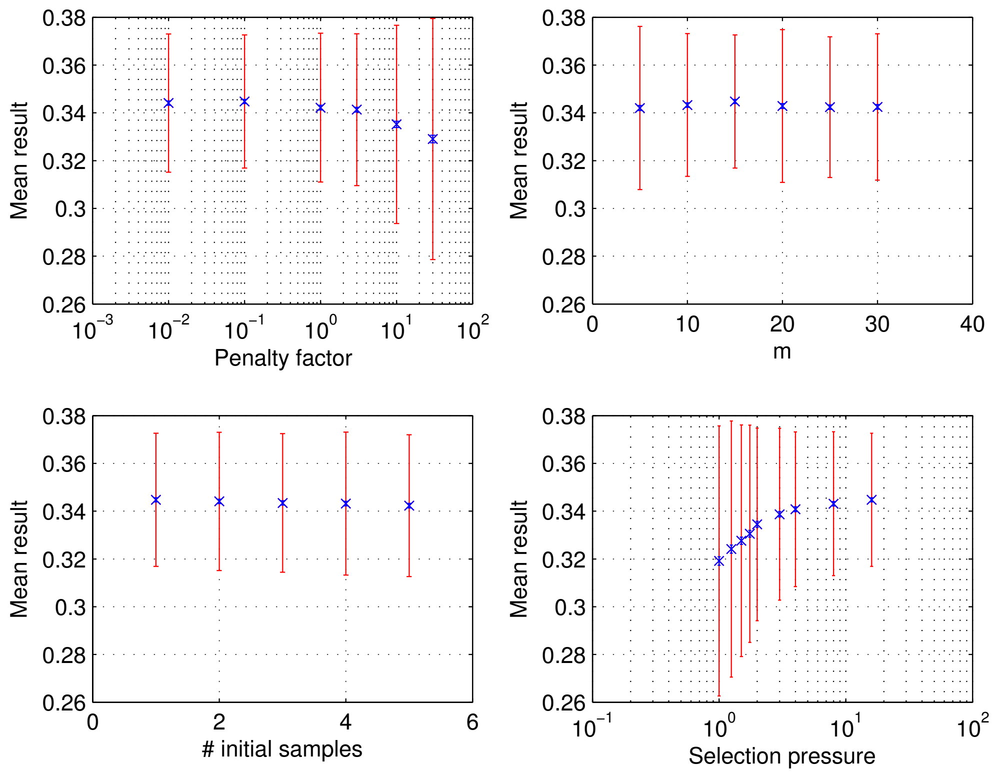

Figure 6: The effects of constraining individual parameters on best performance

4.2 Effects of Individual Parameters

Figure 6 shows how the performance is affected by difference parameters. These plots use the method described in Section 3.2—for each plot, the named parameter is varied and the others are chosen separately for each point to maximise the mean result. The blue points are shown with very small blue error bars that indicate the uncertainty in the estimation of μ. The red error bars stretch 2σ in both directions: they are meant to indicate the variability of the algorithm's performance. Because the result is not normally distributed, they stretch beyond the maximum possibly result.

4.3 Summary of BMCS Parameters

As shown in Figures 5 & 6, the parameters that are critical to good performance on this problem are the penalty weighting factor and the selection pressure. However, appropriate region sizes and initial samples make the algorithm more robust against changes in other parameters.

Penalty factor

If this is much too small, the algorithm may waste time sampling in infeasible space. However, if it is too high and the selection pressure is high, the algorithm may completely fail to sample optima near constraints.

Region size

If this is much too small (large m), the algorithm may waste time in the initial survey. If regions are too large, the algorithm becomes very sensitive to bad settings of other parameters. Figure 3 shows how small regions ($m \geq 20$) guarantee reasonable performance with the tested parameter ranges.

Number of initial samples

This does not have a particularly great effect on the algorithms performance. If the other parameters are set well then just one sample per region suffices for good performance, and allows more samples to be directed at promising regions. However, setting this ≥ 4 provides protection against the failure shown in Figure 4.

Selection pressure

This plays an important part in making the algorithm efficient. Figure 6 shows that with $S \leq 3$, the algorithm's performance is significantly worse, as it doesn't focus its efforts near the optima. Setting it very high allows the best performance with well-chosen parameters, but causes problems when the regions are too small.

The best performing parameter settings tested:

penalty_factor	m	initial_samples	pressure
0.1	15	1	16

4.4 Search Pattern

Figure 7 shows the solutions tested by one of the best performing BMCS algorithms. The grid is included, showing that the global optimum is in a region mostly occupied by invalid solutions. This demonstrates why lower penalty functions provided better performance on this problem. We can also see that the high selection pressure means that in the valleys between functions, only the single initial evaluation is performed before the region is completely abandoned.

4.5 Dissimilarity Archiving

Since it is harder to quantify the quality of a dissimilarity archive, the above approach focused on the best solution found. Figure 8 shows the contents of the dissimilarity archive after a run of the BMCS algorithm with the parameters in Section 4.3. It also shows the results with a smaller region size, which appears to do a better job of locating the 2nd highest peak (although not much can be drawn from a single run of the algorithm). Both results appear reasonable.

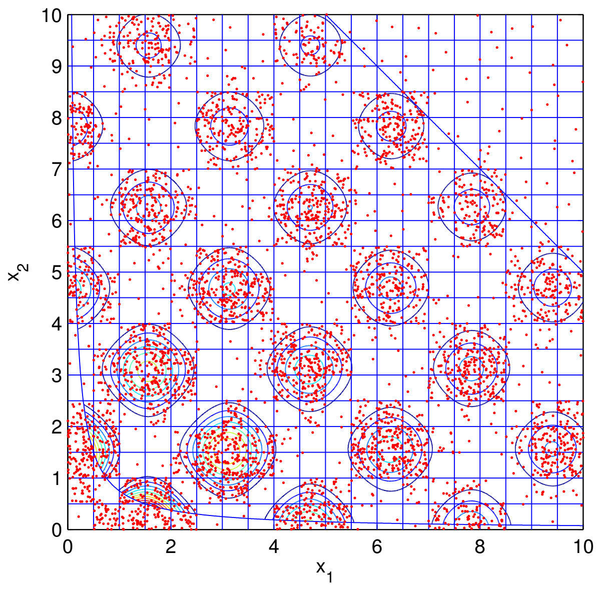

Figure 7: Search pattern of a BMCS search

A more full investigation of the problem could develop objective functions to characterise the quality of the dissimilarity archive, such as the sum of the values, and then investigate how this varies with the parameters.

4.6 Computational Cost

In order to run such an intensive evaluation of the algorithm, the implementation needed to be very efficient. 1620 parameter combinations were tested, each requiring 1000 runs of the algorithm to reduce the impact of noise, resulting in over 8 billion function evaluations.

By fully vectorising the MATLAB code for the BMCS implementation, the time required for a single run of the BMCS algorithm was reduced to 5ms, and the evaluation ran in about 2 hours on a notebook computer.

5 Simulated Annealing

Since the SA algorithm is more complicated, there were more design decisions made in the implementation, and more parameters could be varied.

Initial Temperature Setting

The algorithm can set the initial temperature one of three ways:

- T_0 given as a parameter to the algorithm.
- $T_0 = \dfrac{\delta f^-}{\ln 0.8}$ where δf^- is the mean value of objective reductions encountered in an initial survery [Kirkpatrick, 1984].
- $T_0 = \sigma_0$ where σ_0 is the standard deviation of objective changes encountered in the initial survey [White, 1984].

In both adaptive cases the length of the initial survey was chosen as 500 function evaluations.

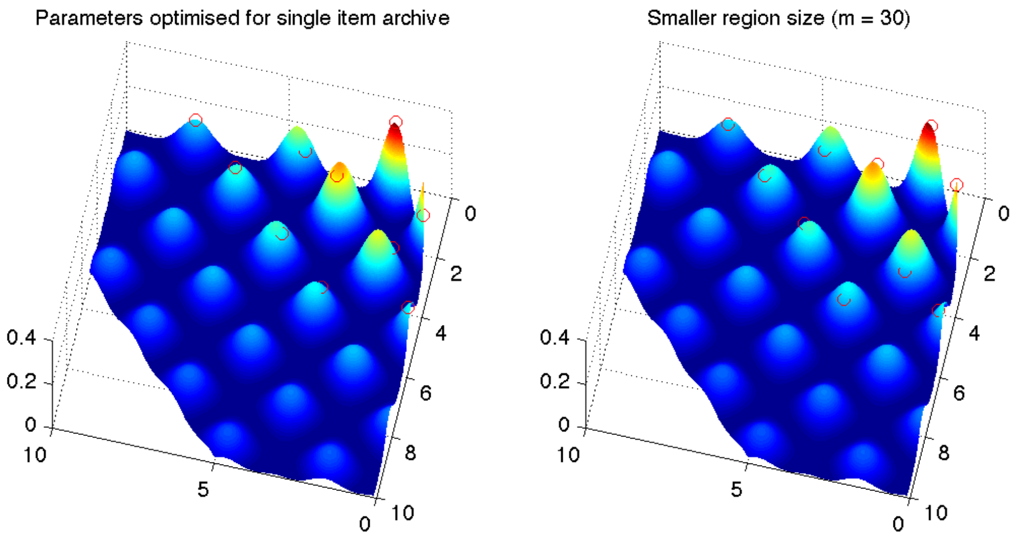

Figure 8: Dissimilarity archives from the BMCS algorithm

Solution Generation

The algorithm generates new candidate solutions by adding a random offset to the current position. The random offset can be chosen in one of three ways. Each way takes a step size parameter, S.

- Each dimension change is uniformly selected from the interval $(-S, S)$.
- Each dimension change is selected from a normal distribution with zero mean and variance S^2.
- Adaptive step sizes [Parks, 1990], as described in the lecture notes. Here, S is the initial maximum step size.

Constraint Handling

As with BMCS, the penalty function described in Section 3.3 is used, with a constant weight parameter, and invalid solutions are not archived. The penalty is also multiplied by T^{-1} as recommended in the lecture notes.

Annealing Schedule

After an optional initial survey of 500 evaluations, the algorithm will perform up to L_k evaluations and up to $\lfloor 0.6L_k \rfloor$ accepted solutions before reducing the temperature. The temperature is then reduced either by a constant factor [Kirkpatrick et al., 1982] or using a scheme [Huang et al., 1986], as given in the lecture notes.

Restarts

The algorithm was found to frequently get stuck in local optima, especially with adaptive step sizes. To work around this, if the algorithm runs 500 evaluations without improving the best value seen so far, it restarts from the best value, and resets the maximum step size to the initial value.

5.1 Searching the Parameter Space

A similar search to that for BMCS was performed, although considerably more intensive:

Solution generation

Uniform, Gaussian or Adaptive (Parks)

Initial maximum step size

0.5, 0.75, 1.0, 1.5 or 2.0

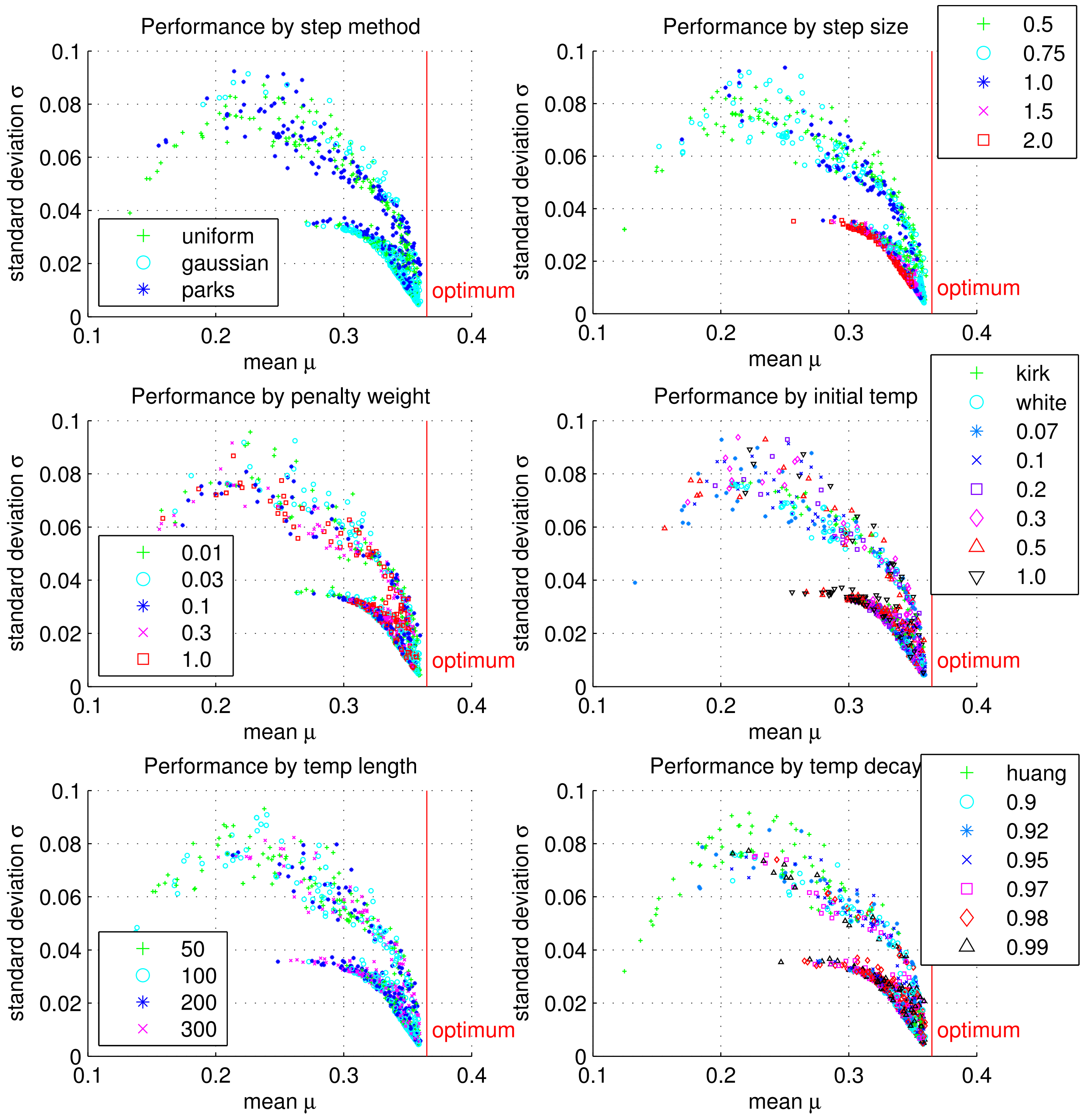

Figure 9: SA performance with varying parameters

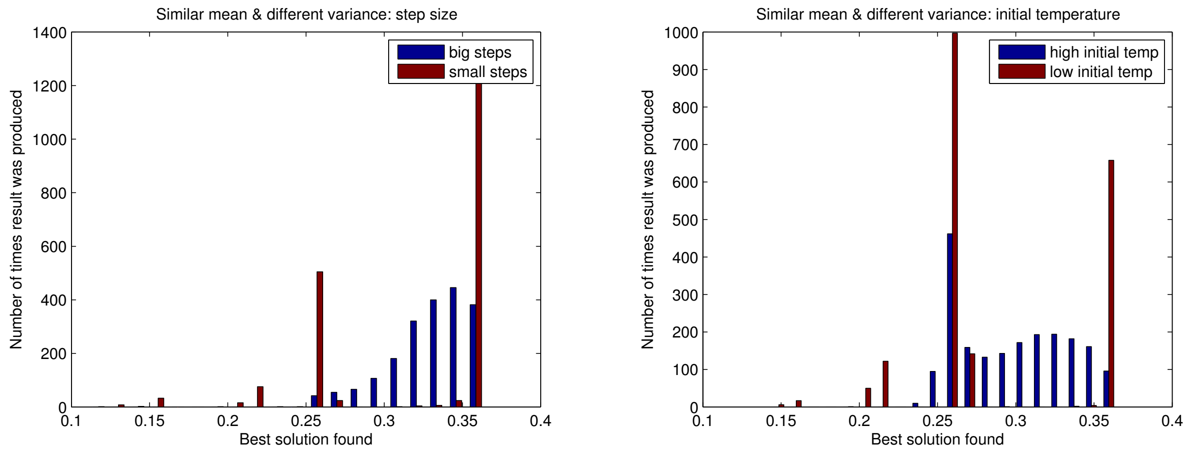

Figure 10: Comparison of result distributions with similar means

Penalty weight
0.01 0.03 0.1 0.3 1.0

Initial temperature
Adaptive (Kirkpatrick), Adaptive (White), 0.07, 0.1, 0.2, 0.3, 0.5 or 1.0

Temperature length L_k
50, 100, 200, or 300

Temperature decay
Adaptive (Huang), 0.9, 0.92, 0.95, 0.97, 0.98 and 0.99

This time, there is a total of 16,800 combinations. For each combination, it was run $n = 3000$ times, as described in Section 3.1. Figure 9 shows the results. Note that because so many more datapoints were collected, these plots show random subsets of the data.

Unfortunately, there are no patterns of the same clarity in this format, and so it is presented mainly for completeness. A few things can be seen, though:

- Gaussian solution generation seems to give better performance in more cases than the other techniques. However, top performance can be achieved with all 3 methods.
- Larger maximum step sizes (1.5 or 2) seem to give more consistent results. It is possible that this is because they get stuck in local optima less frequently, as they are able to step between the peaks of the function.

  Figure 10 suggests that this intuition is correct, but is of limited use since the two parameter sets used differ greatly.
- Similarly, higher initial temperatures seem to give more consistent results than lower temperatures (or the adaptive methods). Again, this is probably due to lower probability of getting caught in local optima, as suggested by Figure 10.
- Smaller penalty weights once again give the best performance, it is not such an important parameter as for BMCS. This is possibly because the penalty is multiplied by T^{-1}.
- The adaptive temperature reduction seems to perform worse in general than the constant exponential approach. However, it is still capable of achieving good performance.

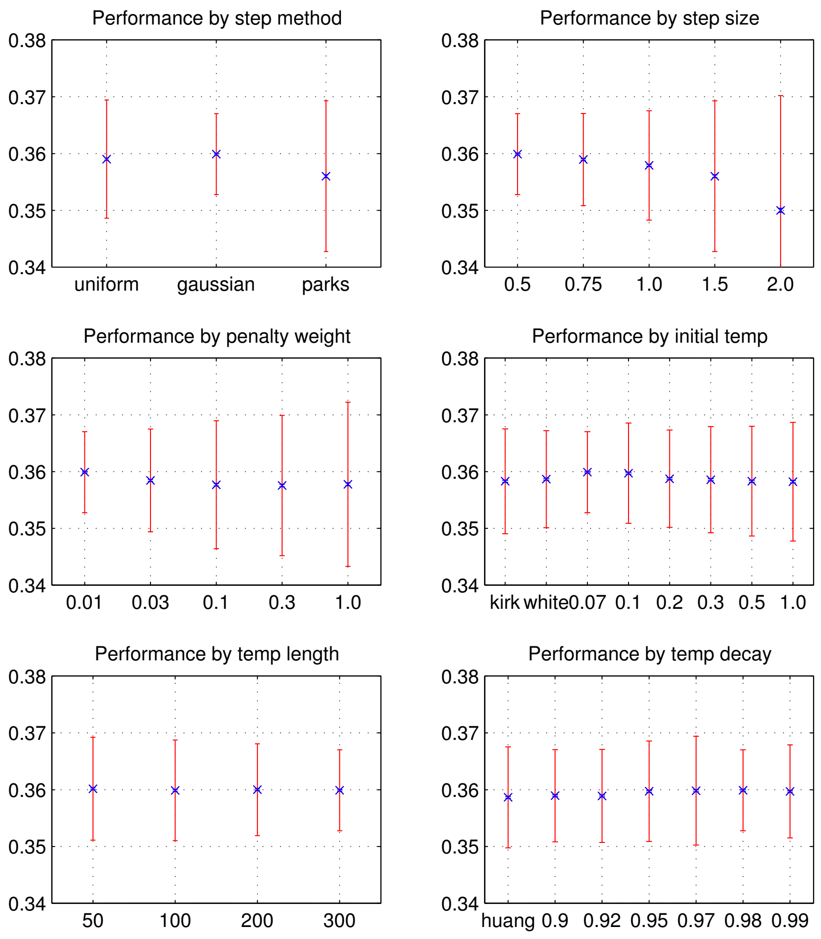

Figure 11: The effects of constraining individual parameters on best performance

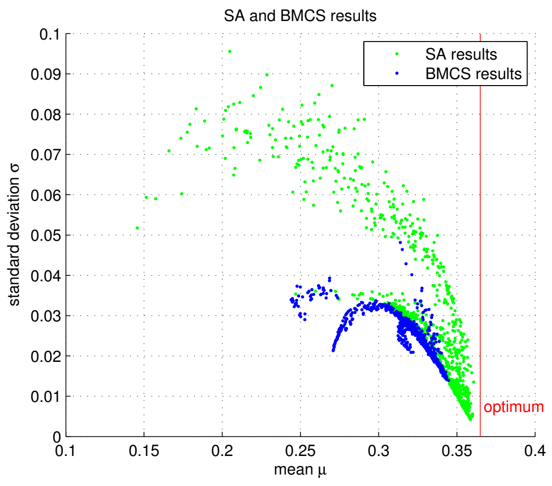

Figure 12: Comparison of SA and BMCS results

5.2 Effects of Individual Parameters

Figure 11 shows the results of the "best performing" parameter sets, as described in Section 3.2—here, the performance metric used to choose which result to display is $\mu - 1.645\sigma$. These plots appear to back up some trends noticed in Figure 9. In particular, Gaussian solution generation and small penalty weightings seem to perform better. However, it also appears that low initial temperatures and small step sizes enable the best performance, which disagrees with what we observed earlier.

It is possible that this approach to the problem is overfitting the parameters: that when the best performing selection out of thousands of different parameter combinations is chosen, an algorithm that regularly finds the optimum in this problem but might perform poorly in a similar problem can be considered best.

5.3 Summary of SA Parameters

Figure 12 makes it clear that SA has the potential to perform better and more reliably than BMCS, if the parameters are chosen well. However, it also has the potential to perform very badly - much worse than the simplest random sampling approach. Unfortunately, it is much harder to see patterns in the performance, or to give general advice on maximising it.

Solution generation

As Figure 13 shows, adaptive solution generation can get closer to the peak more often, because it can reduced the step size and search area near the end of the search. However, it is more prone to only finding the lower local optima, whereas non-adaptive solution generation is less likely to fall into this trap.

Initial maximum step size

A trade-off must be made here: small steps allow pinpointing the optimum, but large steps avoid becoming trapped in a local maximum. This parameter has much less importance when adaptive step sizing is used, but that brings its own problems.

Penalty weight

This parameter is not so important, perhaps since the penalty weight changes as the temperature decreases, but as before lower weights seem to perform slightly better.

Initial temperature

The two methods of adaptive temperature selection appear to perform similarly. Better performance can be achieved by tuning the initial temperature to fit with the other parameters.

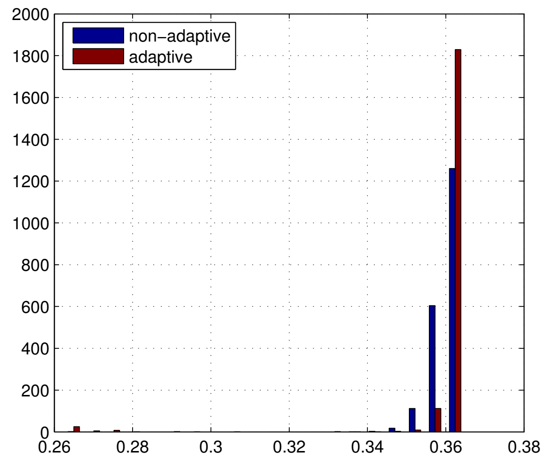

Figure 13: Comparison of adaptive and non-adaptive solution generation

Temperature length L_k

No clear effect is visible here—it appears to be highly dependent on other parameters.

Temperature decay

The adaptive temperature reduction method appears to perform poorly on this problem (unless other parameters are carefully chosen). The exponential cooling scheme [Kirkpatrick et al., 1982] with $\alpha = 0.95$ appears to perform satisfactorily.

5.4 Search Pattern

Figure 14 shows the progress of a successful Simulated Annealing search. Early on, the temperature is high and the algorithm accepts all but the most highly penalised solutions. Next, it appears to have adapted to a smaller step size and is moving between a few local optima. Later on, the search has progressed towards a less promising earlier. However, at this point it triggers the reset criterion, and moves to the best solution tested so far. By the end of the search, it has heavily tested the area around the global optimum.

5.5 Dissimilarity Archiving

The comments of Section 4.5 apply here too: all that will be done to evaluate the quality of the dissimilarity archive is a visual examination. Figure 15 shows the dissimilarity archive for the best performing SA algorithm, and that for the best performing BMCS algorithm for comparison.

The SA algorithm leaves a worse dissimilarity archive: some local optima are completely ignored, and others are less well maximised. This is not surprising, since it focuses on finding the global optimum, rather than exploring any promising parts of the solution space.

5.6 Computational Cost

The SA algorithm was first implemented in MATLAB. This made it very easy to stop the execution part-way through with the debugger, and to visualise intermediate stages. A debugging tool was written that allows the user to easily scroll through the different stages of the annealing schedule, to give an idea of how the search progresses as the temperature reduces.

However, the performance of the MATLAB implementation is very poor. A single run of the algorithm takes around 0.6 seconds—the examination of the parameter space would have taken almost a year to execute with this implementation. To enable the investigation, the implementation was translated to C++, decreasing

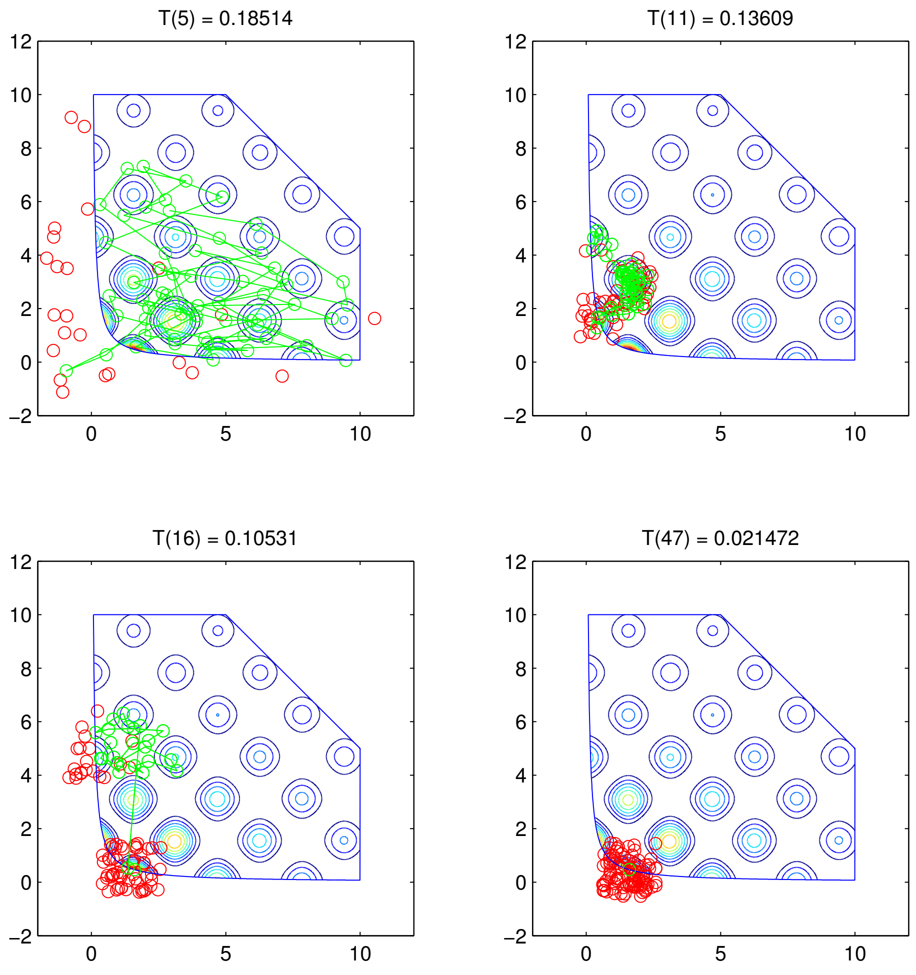

Figure 14: SA Search Pattern

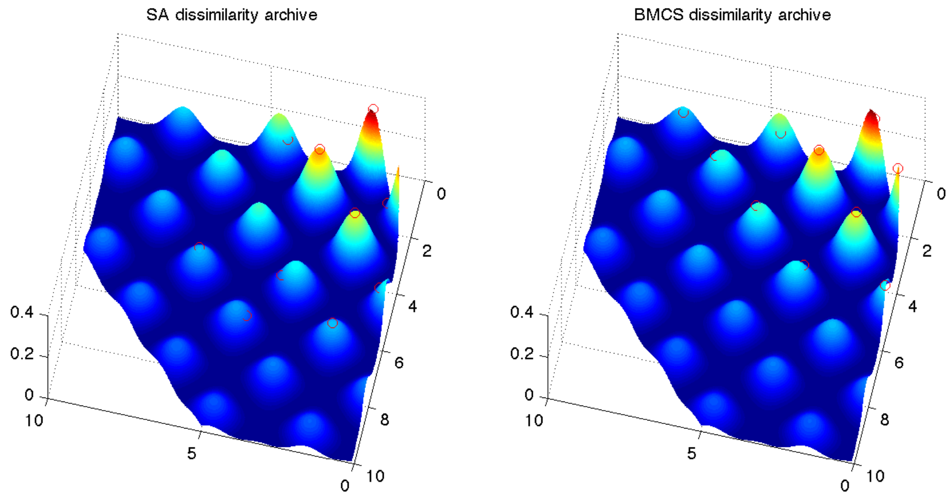

Figure 15: Dissimilarity archives for best performing SA and BMCS algorithms

the time for a single optimisation to 0.6 milliseconds. With this implementation, the 252 billion function evaluations were conducted in about 10 hours on a notebook computer.

6 Conclusions

The Biased Monte Carlo Sampling algorithm was investigated, and found to perform best with small regions and high selection pressures. However, it was still unable to reliably get results close to the optimum. The Simulated Annealing algorithm was found to be able achieve much better performance, and with the right parameters, it could get within a small distance of the global optimum in a large majority of cases. Adaptive step sizing [Parks, 1990] and adaptive temperature initialisation [Kirkpatrick, 1984] [White, 1984] were found to perform well, although not quite as reliably as the best-tuned initialisations.

References

[Huang et al., 1986] Huang, M. D., Romeo, F., and Sangiovanni-Vincentelli, A. (1986). An efficient general cooling schedule for simulated annealing. In *International Conference on Computer Aided Design*, pages 381–384.

[Kirkpatrick, 1984] Kirkpatrick, S. (1984). Optimization by simulated annealing: Quantitative studes. *Journal of Statistical Physics*, 34(5-6):975–986.

[Kirkpatrick et al., 1982] Kirkpatrick, S., C.D. Gerlatt, J., and Vecchi, M. (1982). Optimization by simulated annealing. *IBM Research Report*, RC 9355.

[Parks, 2011] Parks, G. (2011). 5R1 lecture notes.

[Parks, 1990] Parks, G. T. (1990). An intelligent stochastic optimization routine for nuclear fuel cycle design. *Nucl. Technol*, (89):233–246.

[White, 1984] White, S. R. (1984). Concepts of scale in simulated annealing. *AIP Conference Proceedings*, 122(1):261–270.

A Keane's Bump Code

```
function ans = bump(x, y)
    if nargin == 1
        y = x(:,2);
        x = x(:,1);
    end

    cxs = cos(x).^2;
    cys = cos(y).^2;
    ans = abs((cxs.^2 + cys.^2 - 2*cxs.*cys)./sqrt(x.^2+2*y.^2));
```

```
function penalty = bump_penalty(x, y)
% bump_penalty(x, y)
% Returns 0 for solutions that do not violate constraints, and < 0 for
% solutions that do.
    if nargin == 1
        y = x(:,2);
        x = x(:,1);
    end

    % transform x & y to column vectors
    old_size = size(x);
    x = reshape(x, [], 1);
    y = reshape(y, [], 1);

    terms = [(x-0) (10-x) (y-0) (10-y) (15-x-y) (x.*y - 0.75)];
    terms(terms > 0) = 0;
    % sum along the rows of terms
    penalty = sum(terms, 2);
    size(penalty);

    % reshape to original shape
    penalty = reshape(penalty, old_size);
```

B BMCS Code

```
function [ark, xs, ys] = bsmc(f, x_range, y_range, penalty, ... % trick lexer: )
        m, initial_samples, pressure, ark)
% bsmc(f, x_range, y_range, penalty, ...
%       m, initial_samples, pressure, ark) performs a Biased
% Selection Monte Carlo maximisation of the 2-D function f.
% f(x, y): the function to be minimised
% x_range, y_range: 2-entry vectors indicating a bounding-box of the
%                    search space
% penalty(x, y): a function == 0 in allowable space, and < 0 in disallowed
%                 space
% m: the number of intervals to divide each axis into (=> m^2 regions)
% initial_samples: the number of samples per region before biasing
%       (if m^2*initial_samples > 5000, the algorithm fails)
% pressure: a number >= 1 controlling how heavily we bias the selection
% ark: a list of empty archives
%
% returns ark: a list of full archives

    region_sum = zeros(m, m);
    region_n = zeros(m, m);
    region_p = [];
    M = m^2; % number of regions

    samples_remaining = 5000; % Enforces maximum number of samples

    % check for invalid parameters
    if M * initial_samples > samples_remaining
        ark = archive_add(ark, [0 0], 0);
        return
    end

    xs = [];
    ys = [];

    while samples_remaining > 0
        if isempty(region_p)
            % A constant number of initial samples in each square are
            % used to get initial estimate of region probabilities.
            [r,c] = meshgrid(1:m);
            r = reshape(r, [], 1);
            c = reshape(c, [], 1);
            r = repmat(r, initial_samples, 1);
            c = repmat(c, initial_samples, 1);
            samples_remaining = samples_remaining - initial_samples*M;
        else
            % Choose the number of samples to take before re-evaluating
            % region probabilities.
            samples = min(1000, samples_remaining);
            [r,c] = random_square(region_p, samples);
            samples_remaining = samples_remaining - samples;
        end

        % Sample within each chosen region.
        [x,y] = sample_in(x_range, y_range, m, r, c);

        % Calculate the objective function value at chosen sample points.
        assert(samples_remaining >= 0);
        val = f(x, y);
        pen = penalty(x, y);

        % Update region records.
        d_region_sum = accumarray({r, c}, val + pen);
        d_region_n = accumarray({r, c}, 1);
        % Make sure they're the same size as region_sum
        if any(size(d_region_sum) ~= size(region_sum))
            % Assign to lower-right corner to extend to full size
            d_region_sum(m, m) = 0;
            d_region_n(m, m) = 0;
        end
        region_sum = region_sum + d_region_sum;
        region_n = region_n + d_region_n;

        % Rank regions on average objective
        region_avg = region_sum ./ region_n;
        [~, ix] = sort(reshape(region_avg, [], 1), 'descend');
        rank = zeros(size(region_n));
        rank(ix) = 1:length(ix);

        % Calculate new probabilities from ranks
        region_p = pressure*(M+1-2*rank) + 2*(rank-1);
        region_p(region_p < 0) = 0;
        region_p = region_p / sum(sum(region_p));

        % Only archive valid solutions
        ark = archive_add(ark, [x(pen == 0) y(pen == 0)], val(pen == 0));
        xs = [xs; x];
        ys = [ys; y];
    end
end

function [rs,cs] = random_square(region_p, n)
% Returns a row,col indices of an entry in p, randomly chosen with
% probability equal to the contents of the cell (probabilities should
% sum to 1).
    cums = cumsum(reshape(region_p, [], 1));
    cums = cums/cums(end); % just in case they don't sum to 1

    [~, loc] = histc(rand(1,n),[0;cums]);

    k = size(region_p, 1);
    rs = rem(loc'-1, k)+1;
    cs = (loc'-rs)/k + 1;
end

function [x,y] = sample_in(x_range, y_range, m, r, c)
% Returns vectors of points chosen uniformly from within the square region
% r,c, itself a subset of the region x_range*y_range, each axis divided
% up m times.
    x_min = x_range(1);
    x_wid = x_range(2)-x_range(1);
    y_min = y_range(1);
    y_wid = y_range(2)-y_range(1);

    x = x_min + x_wid * (c-1 + rand(size(c)))/m;
    y = y_min + y_wid * (r-1 + rand(size(r)))/m;
end

```

C Archiving Code

```
function ark = archive_new(type, size, varargin)
% archive_new(type, size)
% Create a new archive struct of the given size and type.
    ark.type = type;
    ark.size = size;
    ark.objs = [];
    ark.args = [];

    if strcmp(type, 'dissimilarity')
        ark.dmin = varargin{1};
        ark.dsim = varargin{2};
    end
```

```
function ark = archive_add(ark, args, objs)
% archive_add(ark, args, objs)
% Add data to an archive or cell array of archives.
% - objs is a column vector of objective function values
%   (this function assumes we are maximising)
% - each row of args should be the the parameters achieving the
%   corresponding value in objs

    if (length(ark) ~= 1)
        for i = 1:length(ark)
            ark{i} = archive_add(ark{i}, args, objs);
        end
        return;
    end

    if (size(objs, 2) ~= 1)
        error('objs should be a column vector');
    end
    if (size(objs, 1) ~= size(args, 1))
        error('objs should have the same number of rows as args');
    end
    if (~isempty(ark.args) && size(args, 2) ~= size(ark.args, 2))
        error('differing numbers of arguments in archive and data');
    end

    if strcmp(ark.type, 'best')
        unsorted_args = [ark.args; args];
        [sorted_objs, ix] = sort([ark.objs; objs], 'descend');

        new_size = min(ark.size, size(unsorted_args, 1));

        ark.objs = sorted_objs(1:new_size);
        ark.args = unsorted_args(ix(1:new_size), :);
    elseif strcmp(ark.type, 'single')
        unsorted_args = [ark.args; args];
        [best_obj, best_ix] = max([ark.objs; objs]);
        ark.objs = best_obj;
        ark.args = unsorted_args(best_ix, :);
    elseif strcmp(ark.type, 'complete')
        ark.objs = [ark.objs; objs];
        ark.args = [ark.args; args];
    elseif strcmp(ark.type, 'dissimilarity')

        for i = 1:length(objs)
            % Dissimilarity archiving as given in lecture notes.
            if (length(ark.objs) < ark.size)
                % archive not full: archive if dissimilar to entries so far
                if (all(row_dist(ark.args, args(i,:)) > ark.dmin))
                    ark.args = [ark.args; args(i,:)];
                    ark.objs = [ark.objs; objs(i)];
                end
            elseif (all(row_dist(ark.args, args(i,:)) > ark.dmin))
                % archive full, new entry dissimilar to all prev. entries
                % archive if better than worst
                [obj_worst, i_worst] = min(ark.objs);
                if (objs(i) > obj_worst)
                    ark.args(i_worst,:) = args(i,:);
                    ark.objs(i_worst) = objs(i);
                end
            elseif (all(ark.objs < objs(i)))
                % best so far: archive, replacing closest
                [~, i_closest] = min(row_dist(ark.args, args(i,:)));
                ark.args(i_closest,:) = args(i,:);
                ark.objs(i_closest) = objs(i);
            else
```

```
                % see if v. similar to a solution, and better than it
                similar = row_dist(ark.args, args(i,:)) < ark.dsim;
                can_replace = similar & (ark.objs < objs(i));
                if any(can_replace)
                    i_replace = find(can_replace, 1);
                    ark.args(i_replace,:) = args(i,:);
                    ark.objs(i_replace) = objs(i);
                end
            end
        end

    else
        error(['Unknown archive type: ' ark.type]);
    end
end

function ans = row_dist(a, b)
    if ~isempty(a)
        ans = sqrt(sum(bsxfun(@minus, a, b) .^ 2, 2));
    else
        ans = [];
    end

end
```

D BMCS Parameter Search Code

```
function ark = bsmc_wrapper(penalty_factor, m, initial_samples, pressure)
    ark = bsmc(@bump, [0 10], [0 10], ...
            @(x,y) (penalty_factor*bump_penalty(x,y)), ...
            m, initial_samples, pressure, archive_new('single', 1));
```

```
function results = evaluate(f, n_iters, varargin)
    n_args = length(varargin);
    arg_counts = cellfun(@length, varargin);
    n_combinations = prod(arg_counts);

    results = struct('args', cell(n_combinations, 1), 'mean', 0, 'std', 0);

    p = progressbar();

    for arg_ind = 1:n_combinations
        [arg_sub{1:n_args}] = ind2sub(arg_counts, arg_ind);

        args = arrayfun(@(i) varargin{i}{arg_sub{i}}, 1:n_args, 'UniformOutput', false);

        objs = zeros(n_iters, 1);
        for i = 1:n_iters
            stream = RandStream('mt19937ar', 'Seed', i);
            RandStream.setDefaultStream(stream);
            ark = f(args{:});
            objs(i) = ark.objs(1);
        end

        results(arg_ind).args = args;
        results(arg_ind).mean = mean(objs);
        results(arg_ind).std = std(objs);

        save('results.mat');

        p = setStatus(p, arg_ind/n_combinations);
        display(p);
    end
```

E SA Code

```
function [ark, diag] = sa(f, penalty, ark, ... % trick lexer: )
        step_method, init_step_size, ...
        penalty_weight, ...
        initial_temp, ...
        temp_length, ...
        temp_decay)
% sa(f, penalty, ark,
%       step_method, init_step_size,
%       penalty_weight,
%       initial_temp,
%       temp_length,
%       temp_decay)
% Performs a Simulated Annealing maximisation of the 2-D function f.
%
% f([x y]): the function to be minimised
% penalty([x y]): a function == 0 in allowable space, and < 0 in disallowed
%                 space
% ark: a list of empty archives
% step_method: 'uniform', 'gaussian' or 'parks' giving step size update method
% init_step_size: initial step size
% penalty_weight: constant factor on penalty function
% initial_temp: 'kirkpatrick', 'white' or a number giving initial temp
% temp_length: max number of steps before decreasing temp
% temp_decay: 'huang' or constant factor, controlling how temp decreases
%
% returns ark: a list of full archives
%         diag: information about progress of algorithm

    if any(strcmp(initial_temp, {'kirkpatrick', 'white'}))
        T = inf;
    else
        T = initial_temp;
    end

    step_size = init_step_size;
    position = [5 5];
    objective = f(position);
    % penalised objective
    objective_pen = objective;

    samples_remaining = 5000-1;

    % used to determine when & how to reduce temperature
    objective_changes = [];
    num_trials = 0;
    num_acceptances = 0;
    initial_trials = 500;
    max_trials = temp_length;
    max_acceptances = 0.6*temp_length;

    % parameters controlling step size adaptation
    alpha = 0.1;
    omega = 2.1;

    % used to detect when to restart
    best_obj = objective;
    best_pos = position;
    best_time = samples_remaining;

    % tracking the behaviour of the algorithm
    ctemp = 1;
    diag.temps = [T];
    diag.trials = {[position objective]};
    diag.accepts = {[position objective]};
    diag.rejects = {[]};

    while samples_remaining > 0
        % consider restarting if we've been too long since seeing
        % the best observation
        if (best_time - samples_remaining) > 500
            best_time = samples_remaining;
            position = best_pos;
            objective = best_obj;
            objective_pen = best_obj + penalty_weight * penalty(position) / T;

            % reset step size to avoid getting stuck with bad step size
            step_size = init_step_size;
        end

        if strcmp(step_method, 'gaussian')
            step = step_size .* randn(1,2);
        else
            step = step_size .* (2*rand(1,2)-1);
        end


        new_penalty = penalty_weight * penalty(position+step);

        % ignore all invalid solutions in the initial survey
        if T == inf && new_penalty ~= 0
            continue;
        end

        new_objective = f(position+step);
        new_objective_pen = new_objective + new_penalty / T;
        samples_remaining = samples_remaining - 1;

        diag.trials{ctemp} = [diag.trials{ctemp};
                              position+step new_objective_pen];

        num_trials = num_trials + 1;

        % only archive valid solutions
        if new_penalty == 0
            ark = archive_add(ark, position+step, new_objective);

            % update reset counters
            if new_objective > best_obj
                best_obj = new_objective;
                best_pos = position+step;
                best_time = samples_remaining;
            end
        end

        % calculate acceptance probability
        if strcmp(step_method, 'parks')
            p = exp(- (objective_pen - new_objective_pen) / (T * norm(step)));
        else
            p = exp(- (objective_pen - new_objective_pen) / T);
        end

        % accept change with probability 1-p
        if rand() < p
            diag.accepts{ctemp} = [diag.accepts{ctemp};
                                   position+step new_objective_pen];

            num_acceptances = num_acceptances + 1;
            objective_changes = [objective_changes new_objective_pen-objective_pen];

            position = position + step;
            objective = new_objective;
            objective_pen = new_objective_pen;

            % update step size info (if out of initial survey)
            if T ~= inf && strcmp(step_method, 'parks')
                step_size = (1 - alpha) * step_size + ...
                       alpha * omega * abs(step);
            end
        else
            diag.rejects{ctemp} = [diag.rejects{ctemp};
                                   position+step new_objective_pen];
        end

        % consider reducing temperature
        reduced_T = false;
        if T == inf
            if num_trials >= initial_trials
                % set initial temperature
                if strcmp(initial_temp, 'kirkpatrick')
                    df_neg = - mean(objective_changes(objective_changes < 0));
                    T = - df_neg / log(0.8);
                elseif strcmp(initial_temp, 'white')
                    T = std(objective_changes);
                else
                    error('unknown temp_method');
                end
                reduced_T = true;
            end
        elseif num_trials >= max_trials || num_acceptances >= max_acceptances
            if strcmp(temp_decay, 'huang')
                factor = exp(-0.7 * T / std(diag.accepts{ctemp}(:,3)));
                factor = max(0.5, factor);
                T = T * factor;
            else
                T = T * temp_decay;
            end
            reduced_T = true;
        end

        if reduced_T
            % reset counters
            objective_changes = [];
            num_trials = 0;
            num_acceptances = 0;

            ctemp = ctemp+1;
            diag.temps(ctemp) = T;
            diag.trials{ctemp} = [position objective_pen];
            diag.accepts{ctemp} = [position objective_pen];
            diag.rejects{ctemp} = [];
        end
    end
end
```

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include <string.h>

#if 0
    typedef float real;
#define powr powf
#define logr logf
#define cosr cosf
#define sinr sinf
#define fabsr fabsf
#define sqrtr sqrtf
#else
    typedef double real;
#define powr pow
#define logr log
#define cosr cos
#define sinr sin
#define fabsr fabs
#define sqrtr sqrt
#endif

enum {
    uniform,
    gaussian,
    parks,
} step_method;

real init_step_size;
real penalty_weight;
enum {
    kirkpatrick,
    white,
    constant,
} initial_temp_method;
real initial_temp;
int temp_length;
enum {
    huang,
    exponential,
} temp_decay_method;
real temp_decay;

real randf()
{
    return random() / powr(2, 31) + 1 / powr(2, 32);
}

real randn()
{
    static bool prepped = false;
    static real prepped_result = 0;

    if (prepped)
    {
        prepped = false;
        return prepped_result;
    }
    else
    {
        real u1 = randf(), u2 = randf();
        real len = sqrtr(-2 * logr(u1));
        prepped = true;
        prepped_result = len * sinr(2*M_PI*u2);

        return len * cosr(2*M_PI*u2);
    }
}

real mean(real x[], int n)
{
    real sx = 0;

    for (int i=0; i<n; i++)
    {
        sx += x[i];
    }

    return sx / n;
}

real std(real x[], int n)
{
    real sx = 0, sxx = 0;

    for (int i=0; i<n; i++)
    {
        sx += x[i];
        sxx += x[i]*x[i];
    }

    return sqrt((sxx - sx*sx/n) / (n-1));
}

real bump(real x, real y)
{
    real cxs = cosr(x); cxs *= cxs;
    real cys = cosr(y); cys *= cys;

    return fabsr((cxs*cxs + cys*cys - 2*cxs*cys) / sqrtr(x*x+2*y*y));
}

real penalty(real x, real y)
{
#define BOUND(x) ((x) < 0 ? x : 0)
    return BOUND(x-0)+BOUND(10-x)+BOUND(y-0)+BOUND(10-y)+BOUND(15-x-y)+BOUND(x*y-0.75);
}

real T_kirkpatrick(real x[], int n)
{
    int n_neg = 0;
    real sx_neg = 0;

    for (int i=0; i<n; i++)
    {
        if (x[i] < 0)
        {
            sx_neg += -x[i];
            n_neg++;
        }
    }

    return - (sx_neg / n_neg) / logr(0.8);
}

real sa(unsigned seed)
{
    srandom(seed);

    real T;
    if (initial_temp_method != constant)
        T = INFINITY;
    else
        T = initial_temp;

    real step_size_x = init_step_size, step_size_y = init_step_size;
    real pos_x = 5, pos_y = 5;
    real obj = bump(pos_x, pos_y);
    real obj_pen = obj;

    int samples_remaining = 5000-1;

    real obj_d[5000];
    int n_obj_d = 0;
    real accepts[5000];
    int n_accepts = 0;

    int num_trials = 0, num_acceptances = 0;
    int initial_trials = 500;
    int max_trials = temp_length;
    int max_acceptances = 0.6*temp_length;

    real alpha = 0.1, omega = 2.1;

    real best_obj = obj;
    real best_x = pos_x, best_y = pos_y;
    int best_time = samples_remaining;

    while (samples_remaining > 0)
    {
        if (best_time - samples_remaining > 500)
        {
            best_time = samples_remaining;
            pos_x = best_x;
            pos_y = best_y;
            obj = best_obj;
            obj_pen = best_obj + penalty_weight * penalty(pos_x, pos_y) / T;

            step_size_x = step_size_y = init_step_size;
        }

        real step_x, step_y;
        if (step_method == gaussian)
        {
            step_x = step_size_x * randn();
            step_y = step_size_y * randn();
        }
        else
        {
            step_x = step_size_x * (2*randf()-1);
            step_y = step_size_y * (2*randf()-1);
        }

        real new_x = pos_x+step_x, new_y = pos_y+step_y;

        real new_pen = penalty_weight * penalty(new_x, new_y);

        if (T == INFINITY && new_pen != 0)
            continue;

        real new_obj = bump(new_x, new_y);
        real new_obj_pen = new_obj + new_pen / T;
        samples_remaining--;

        num_trials++;

        if (new_pen == 0)
        {
            if (new_obj > best_obj)
            {
                best_obj = new_obj;
                best_x = new_x;
                best_y = new_y;
                best_time = samples_remaining;
            }
        }

        real p;
        if (step_method == parks)
        {
            real step_norm = sqrtr(step_x*step_x + step_y*step_y);
            p = exp(- (obj_pen - new_obj_pen) / (T * step_norm));
        }
        else
        {
            p = exp(- (obj_pen - new_obj_pen) / T);
        }

        if (randf() < p)
        {
            num_acceptances++;
            obj_d[n_obj_d++] = new_obj_pen - obj_pen;
            pos_x = new_x;
            pos_y = new_y;
            obj = new_obj;
            obj_pen = new_obj_pen;

            accepts[n_accepts++] = new_obj_pen;

            if (T != INFINITY && step_method == parks)
            {
                step_size_x = (1-alpha)*step_size_x + alpha*omega*fabsr(step_x);
                step_size_y = (1-alpha)*step_size_y + alpha*omega*fabsr(step_y);
            }
        }

        bool reduced_T = false;

        if (T == INFINITY)
        {
            if (num_trials >= initial_trials)
            {
                if (initial_temp_method == kirkpatrick)
                    T = T_kirkpatrick(obj_d, n_obj_d);
                else if (initial_temp_method == white)
                    T = std(obj_d, n_obj_d);
                else
                {
                    fprintf(stderr, "unknown temp method");
                    exit(1);
                }
                reduced_T = true;
            }
        }
        else if (num_trials >= max_trials || num_acceptances >= max_acceptances)
        {
            if (temp_decay_method == huang)
            {
                real factor;
                if (n_accepts < 2)
                    factor = 0.5;
                else
                {
                    factor = exp(-0.7*T/std(accepts, n_accepts));
                    if (factor < 0.5)
                        factor = 0.5;
                }
                T *= factor;
            }
            else
                T *= temp_decay;
            reduced_T = true;
        }

```

```c
        if (reduced_T)
        {
            //printf("%g\n", T);
            n_obj_d = 0;
            num_trials = 0;
            num_acceptances = 0;
            n_accepts = 1;
            accepts[0] = obj_pen;
        }
    }

    return best_obj;
}

int main(int argc, char **argv)
{
    if (argc != 8)
    {
        printf("needs 7 args\n");
        return 1;
    }

    int n_iters = atoi(argv[1]);

    if (strcmp(argv[2], "uniform") == 0)
        step_method = uniform;
    else if (strcmp(argv[2], "gaussian") == 0)
        step_method = gaussian;
    else if (strcmp(argv[2], "parks") == 0)
        step_method = parks;
    else
    {
        printf("unknown step method\n");
        return 1;
    }

    init_step_size = atof(argv[3]);
    penalty_weight = atof(argv[4]);

    if (strcmp(argv[5], "kirkpatrick") == 0)
        initial_temp_method = kirkpatrick;
    else if (strcmp(argv[5], "white") == 0)
        initial_temp_method = white;
    else
    {
        initial_temp_method = constant;
        initial_temp = atof(argv[5]);
    }

    temp_length = atoi(argv[6]);

    if (strcmp(argv[7], "huang") == 0)
        temp_decay_method = huang;
    else
    {
        temp_decay_method = exponential;
        temp_decay = atof(argv[7]);
    }

    real results[n_iters];

    for (int i=0; i<n_iters; i++)
    {
        results[i] = sa(i);
        //printf("%g\n", results[i]);
    }

    real m = mean(results, n_iters);
    real s = std(results, n_iters);

    printf("%g %g\n", m, s);

    return 0;
}
```

F SA Parameter Search Code

```
function results = cli_evaluate(n_iters, varargin)
    n_args = length(varargin);
    arg_counts = cellfun(@length, varargin);
    n_combinations = prod(arg_counts);

    results = struct('args', cell(n_combinations, 1), 'mean', 0, 'std', 0);

    p = progressbar();

    for arg_ind = 1:n_combinations
        [arg_sub{1:n_args}] = ind2sub(arg_counts, arg_ind);

        args = arrayfun(@(i) varargin{i}{arg_sub{i}}, 1:n_args, 'UniformOutput', false);

        cli = strjoin(' ', './sa', num2str(n_iters), args{:});

        [~, ans] = system(cli);
        ans = sscanf(ans, '%f');

        results(arg_ind).args = args;
        results(arg_ind).mean = ans(1);
        results(arg_ind).std = ans(2);

        save('results.mat');

        p = setStatus(p, arg_ind/n_combinations);
        display(p);
    end
```