# Two methods for optimising cognitive model parameters

**David Peebles**

Department of Behavioural and Social Sciences, University of Huddersfield, UK.

Summary

Searching for the best set of parameter values is a key component of cognitive modelling and one in which a great deal of uncertainty lies. Parameter search can be a slow, laborious process when done by hand, particularly when a model has several interacting parameters, and can be challenging when models are non-differentiable, non-continuous, non-linear, stochastic, or have many local optima.

Two powerful methods for searching parameter spaces for such models by generating populations of models are *differential evolution* (DE) and using a *High Throughput Computing* (HTC) environment managed by *HTCondor*. Below I describe both and provide an example of each using a simple ACT-R model.

## Differential evolution

- An evolutionary strategy for real numbers used for multidimensional numerical optimisation [4, 5].
- **Attractions:**
  - Simplicity of the algorithm
  - Only three control parameters
  - Fast, accurate, and robust performance
  - Wide applicability
- **Control parameters:**

  **NP** the population size
  **F** a scale factor applied to the mutation process
  **Cr** a constant that regulates the crossover process

- To use DE for optimising cognitive model parameters, the model is conceptualised as an objective function of the parameters being optimised that produces a single fitness value (e.g., $R^2$) to be maximised.

### The DE algorithm

**Process in a nutshell.** In common with many evolutionary algorithms, DE applies repeated cycles of *mutation*, *recombination*, and *selection* (Figure 1) on an initial, randomly generated population of vectors to create a single vector that produces the best solution to a problem.

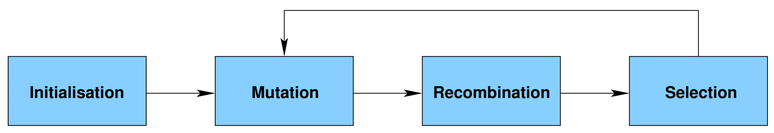


**Figure 1:** Differential evolution algorithm

#### Initialisation

The population is initialised by creating an $NP$ sized population of real-valued vectors of $D$ dimensions, one dimension for each of the model parameters to be optimised. The vectors are initialised with uniformly distributed random numbers within maximum and minimum bounds set for each dimension (Figure 2).

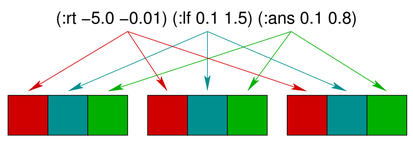


**Figure 2:** Initialisation of population vectors

To create the next population of vectors, each vector **i** in the current population is selected in sequence, designated as the *target* vector, and subjected to a competitive process. The competition involves the three mutation, recombination, and selection steps described below.

#### Mutation

Mutation randomises the search process by computing the weighted difference between two vectors in the current population. This ensures that differences in the scale and sensitivity of different vector parameters are taken into account and that the search space is explored equally on all dimensions.

A mutated *donor* vector is created by randomly selecting three unique vectors, **j**, **k** and **l**, which are not equal to **i**, from the population and adding the difference between **j** and **k** (scaled by the $F$ parameter) to **l** (Figure 3).

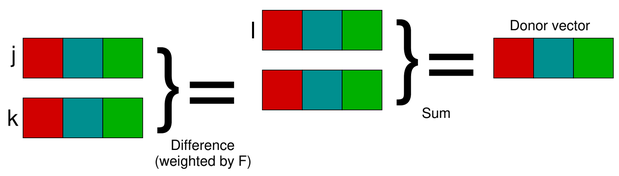


**Figure 3:** Creation of a *donor* vector using three vectors randomly selected from the population

#### Recombination

Once the donor vector has been created it is crossed with the target vector to create the *trial* vector. This recombination allows successful solutions from the previous generation to be incorporated into the trial vector (Figure 4).

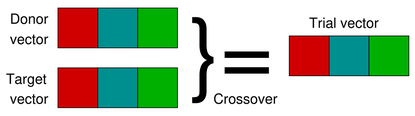


**Figure 4:** Crossing donor and target vectors to create a trial vector

Crossover is achieved by a series of Bernoulli trials which determine for each of $D - 1$ dimensions which parent will donate its value. The process is moderated by the crossover rate parameter $Cr$ (where $0 \leq Cr \leq 1$.). For each dimension, a uniformly distributed random number, $x$ between 0 and 1 is generated and compared to $Cr$. If $x \leq Cr$, the donor vector's parameter is passed on to the trial vector, otherwise the parameter comes from the target vector. To ensure that the trial vector does not emerge identical to the target vector, one dimension is selected at random to inherit its value from the donor vector.

#### Selection

The model is then run with the parameter values from the trial vector and if the resulting fitness value is better than or equal to that of the target vector, the trial vector replaces it in the next generation, or else the target vector is retained in the next generation. This process is carried out for each vector in the current population until the next population is created and the evolutionary process continues for a user-defined number of cycles. The vector with the highest fitness is recorded for each population and the winning vector in the final population is considered the best solution to the problem.

### Setting DE parameters

The performance of the DE algorithm is sensitive to its three control parameters.

**F** In the mutation process the $F$ constant scales all of the vector parameters equally and determines the size of the distance between the target and trial vectors. Effective values for $F$ are generally regarded to fall between 0.4 and 1.0 with a good initial value being 0.5 [1, 4].

**Cr** The crossover rate parameter $Cr$ affects the search process by regulating the probability that noisy random values enter the trial vector (raising $Cr$ increases the likelihood that dimension values will come from the donor vector). Although views differ, $Cr$ values between 0.3 and 0.9 are generally considered reasonable for the majority of functions.

**NP** Recommendations for the optimum population size, $NP$ are generally specified as a function of the number of vector parameters, $D$, and also vary but typically range between 3D and 10D [e.g., 2, 4].

## High throughput computing with HTCondor

While DE is useful for optimising models with relatively few parameters or short run times on a single computer, if models are large, complex, or are simulating the behaviour of many participants, then the computational requirements may be such that this option becomes impractical. In these circumstances, an alternative is to search the parameter space by running a population of models over a computer network and one relatively accessible and increasingly popular way to do this is by using *HTCondor* (**https://research.cs.wisc.edu/htcondor**).

HTCondor is an open source, cross-platform software system for managing and scheduling computationally intensive tasks across computer networks developed over many years at the University of Wisconsin-Madison [3]. It can be employed on dedicated server clusters or to schedule tasks over idle desktop computers on a network (Figure 5).

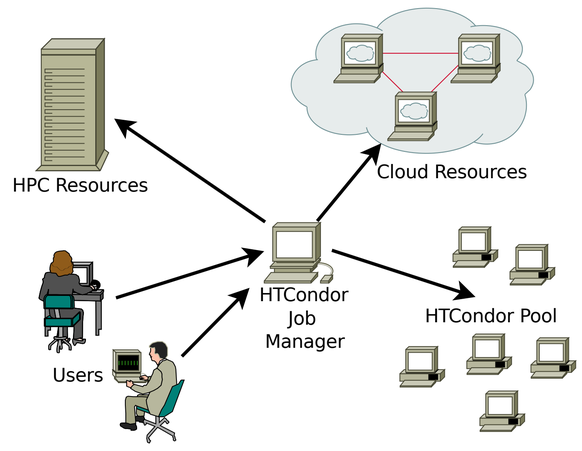


**Figure 5:** HTCondor architecture

Using HTCondor for exploring parameter spaces for cognitive models can be achieved by submitting multiple versions of the model, each with a different set of randomly generated parameter values, analysing the returned outputs, and then iterating. The process for doing so is relatively straightforward. All that is required is the creation of a *submit description file* which specifies details about the job such as the executable to be run and upon which platform, the model files to be loaded by the executable, the command to start the program running, and the number of times to run the program. As each program may also use the standard streams, files must be defined that will substitute for stdin, stdout and stderr.

HTCondor code for the ACT-R *paired associate* model

```
requirements = (OpSys == "WINNT61" && Arch == "INTEL") ||
               (OpSys == "WINDOWS" && Arch == "INTEL") ||
               (OpSys == "WINDOWS" && Arch == "X86_64"))

executable = actr-s-64.exe
arguments = "-l 'paired.lisp' -e '(collect-data 20)' -e '(quit)'"

transfer_executable = ALWAYS
transfer_input_files = paired.lisp

output = out.stdout.$(Cluster).$(Process)
error = out.err.$(Cluster).$(Process)
log = out.clog.$(Cluster).$(Process)

queue 100
```

For example, the code extract above is from a submit description file for a job to run 100 instances of an ACT-R model defined in the file *paired.lisp*. It specifies:

1. That only 64-bit Windows machines in the network should be used.
2. That the executable is ACT-R for 64-bit Windows.
3. A string of arguments to the executable: load the model file, run it for 20 participants, and then quit.
4. The executable and the model file to be transferred to each machine.
5. The output, error, and log files to be created for each instance of the model.
6. Finally, the last command sets the job to run 100 instances of the model.

When all of the model instances have been run, their outputs are saved in numbered output files which can then be collected together and analysed.

## Try them out

To enable further investigation of these methods, code to optimise an ACT-R model of paired associate learning taken from Unit 4 of the ACT-R tutorials (available from the ACT-R website: **http://act-r.psy.cmu.edu/software/**), together with full instructions is available on GitHub. The repository for differential evolution can be found at **https://github.com/peebz/actr-paired-de** while that for running the model on HTCondor is available at **https://github.com/peebz/actr-paired-htc**.

## References

[1] S. Das and P. N. Suganthan. 'Differential evolution: A survey of the state-of-the-art'. In: *IEEE Transactions on Evolutionary Computation* 15.1 (2011), pp. 4–31.

[2] R. Gämperle, S. D. Müller, and P. Koumoutsakos. 'A parameter study for differential evolution'. In: *Advances in intelligent systems, fuzzy systems, evolutionary computation* 10 (2002), pp. 293–298.

[3] M. J. Litzkow, M. Livny, and M. W. Mutka. 'Condor—A Hunter of Idle Workstations'. In: *Proceedings of the 8th International Conference on Distributed Computing Systems*. IEEE. June 1988, pp. 104–111.

[4] R. Storn and K. Price. *Differential evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces*. Tech. rep. TR-95-012. Berkeley, CA: ICSI Berkeley, 1995.

[5] R. Storn and K. Price. 'Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces'. In: *Journal of global optimization* 11.4 (1997), pp. 341–359.