DIFFEV

Users Guide

Version 5.14

written by

Reinhard Neder
Email: reinhard.neder@fau.de

http://tproffen.github.io/DiffuseCode

Document created: November 27, 2017

Preface

Disclaimer

The DIFFEV software described in this guide is provided without warranty of any kind. No liability is taken for any loss or damages, direct or indirect, that may result through the use of *DIFFEV*. No warranty is made with respect to this manual, or the program and functions therein. There are no warranties that the programs are free of error, or that they are consistent with any standard, or that they will meet the requirement for a particular application. The programs and the manual have been thoroughly checked. Nevertheless, it can not be guaranteed that the manual is correct and up to date in every detail. This manual and the DIFFEV program may be changed without notice.
DIFFEV is intended as a public domain program. It may be used free of charge. Any commercial use is, however, not allowed without permission of the authors.

Using DIFFEV

Publications of results totally or partially obtained using the program DIFFEV should state that DIFFEV was used and contain the following reference:

> NEDER, R.B. in preparation - check website.

More information

This users guide can only provide program specific details. A broader discussion of simulation techniques and some DIFFEV examples and macro files can be found in our book

> NEDER, R.B. & PROFFEN, TH. "Diffuse Scattering and Defect Structure Simulations - A cook book using the programs DISCUS", *IUCr Texts on Crystallography*, Oxford University Press, 2007.

Contents

Chapter 1

Introduction

1.1 What is DIFFEV ?

DIFFEVis a generic evolutionary refinement program that implements the differential evolutionary algorithm **?**. Evolutionary or genetic refinement algorithms allow the refinement of models, functions, or more generally speaking the parameters of a cost function to obtain a good solution.
A least squares based refinement of a function $y = F(p_0, p_1, ..., p_n)$ requires the calculation of all partial derivatives $\partial y/\partial p_i$, either from an analytical or a numeric solution. If these derivatives cannot be calculated, either because they cannot be derived analytically or because the numeric computation is too time consuming, evolutionary algorithms offer a possibility to refine these parameters. All algorithms are population based, i.e. several different parameter sets $P_I : [p_0, p_1, ..., p_n]$ are created simultaneously. For ech of these parameter sets, the value of the cost function is computed. In the next step, a new group of parameter sets is generated and the cost function calculated anew. The respective values of the cost function are compared and those parameter sets that yield the better cost function will in turn be taken to generate the next generation of parameter sets. By carefully designed modification of the parameter values from generation to generation and by weeding out those parameter sets that lead to a bad fit, the algorithm will eventually find parameter sets that provide a good fit to the experimentally determined function.
DIFFEV provides the refinement part of such an evolutionary algorithm. It creates the group of parameter sets, compares the cost function values between two successive generations and creates the next generation based on a comparison between the old and new cost function values. It does, however, not calculate the cost function itself. This task is handed over to a slave program. Since this slave program could calculate any cost function, DIFFEV is not limited to the refinement of a particular physical problem.

1.2 What is new ?

DIFFEV is avaliable as a stand alone programme and may also be used within the DISCUS SUITE. The DISCUS SUITE is optimized with respect to the performance on a large scale computing facility. Several new features are available within the DISCUS SUITE. These are explained in the

DISCUS SUITE manual.

1.3 Getting started

After the program *DIFFEV* is installed properly and the environment variables are set, the program can be started by typing 'diffev' at the operating systems prompt.

Symbol	Description
"text"	Text given in double quotes is to be understood as typed.
<text>	Text given in angled brackets is to be replaced by an appropriate value, if the corresponding line is used in DIFFEV. It could, for example be the actual name of a file, or a numerical value.
`text`	Text in single quotes exclusively refers to DIFFEV commands.
[text]	Text in square brackets describes an optional parameter or command. If omitted, a default value is used, else the complete text given in the square brackets is to be typed.
{text | text}	Text given in curly brackets is a list of alternative parameters. A vertical line separates two alternative, mutually exclusive parameters.

Table 1.1: Used symbols

The program uses a command language to interact with the user. The command `exit` terminates the program and returns control to the shell. All commands of DIFFEV consist of a command verb, optionally followed by one or more parameters. All parameters must be separated from one another by a comma ",". There is no predefined need for any specific sequence of commands. DIFFEV is case sensitive, all commands and alphabetic parameters MUST be typed in lower case letters. If DIFFEV has been compiled using the `-DREADLINE` option (see installation files) basic line editing and recall of commands is possible. For more information refer to the reference manual or check the online help using (`help command input`). Names of input or output files are to be typed as they will be expected by the shell. If necessary include a path to the file. All commands may be abbreviated to the shortest unique possibility. At least a single space is needed between the command verb and the first parameter. No comma is to precede the first parameter. A line can be marked as comment by inserting a "`#`" as first character in the line.
The symbols used throughout this manual to describe commands, command parameters, or explicit text used by the program DIFFEV are listed in Table 1.1. There are several sources of information, first DIFFEV has a build in online help, which can be accessed by entering the command `help` or if help for a particular command <cmd> is wanted by `help <cmd>`. This manual describes background and principle functions of DIFFEV and should give some insight in the ways to use this program.
DIFFEV is distributed as part of the diffuse scattering simulation software DISCUS. However, DIFFEV can be used as general refinement program separate from the DISCUS program package.

Variable	Description
pop_gen[1]	Current population number
pop_n[1]	Number of members in the population
pop_c[1]	Number of children in the population
pop_dimx[1]	Number of parameters to be refined
diff_cr[1]	Cross over probability
diff_f[1]	Scale factor for the difference vectors
diff_k[1]	point along line between parent and donor base
diff_lo[1]	Local search probability
diff_sel[1] *	Selection mode: 0 compare to parent; 1 use best of (members and children); 2 use best of (children)
pop_xmin[i]	Minimum allowed value for parameter no. i
pop_xmax[i]	Maximum allowed value for parameter no. i
pop_smin[i]	Minimum allowed starting value for parameter no. i
pop_smax[i]	Maximum allowed starting value for parameter no. i
pop_sig[i]	Global sigma for parameter no. i
pop_lsig[i]	Local search sigma for parameter no. i
pop_v[i,j] *	Value of parameter no. i for member no. j
pop_t[i,j] *	Value of current trial parameter no. i for child no. j
rvalue[i] *	R-value for member no. i
child_val[i]	R-value for child no. i
bestm[1] *	Number of member with best R-value
bestr[1] *	Best R-value
worstm[1] *	Number of member with worst R-value
worstr[1] *	Worst R-value

Table 1.2: DIFFEV variables. Variables marked with * are read-only and cannot be altered.

1.4 Command language

The program includes a FORTRAN style interpreter that allows the user to program complex modifications. A detailed discussion about the command language which is common to all DISCUS package programs can be found in the separate DISCUS package reference guide which is included with the package. Table 1.2 shows a summary of DIFFEV specific variables. Some of these variables can not be modified, others like can be altered, thus allowing to modify the refinement strategy.

Chapter 2

Differential Evolutionary Algorithm

2.1 Refinement via evolutionary algorithms

Every time we measure some physical effect and wish to understand how this effect works, we want to determine the parameters of a model function that will replicate the observations. The term refinement refers to the process by which the parameters of the function are tuned such as to give the best agreement between the observed and calculated values. The term *best agreement* merits careful definition, for right now it is sufficient to say that the sum over all squared differences between the observations and the calculations shall be minimized. Thus, refinement is but a special case of general optimization. A very different example for an optimization could be the task to place as many integrated circuits into a chip and simultaneously achieve the fastest computations. Quite well known is the traveling salesman problem. Here the optimization task requires to find the shortest route that visits a number of spots distributed in space.
By far the fastest refinement technique is a least squares algorithm. Such an algorithm can always be applied if we can describe the physical effect as a function of parameters:

$$y \; = \; F(p_0, p_1, ..., p_n), \tag{2.1}$$

and all the partial derivatives $\partial y / \partial p_i$ can be calculated, either analytically or numerically. For each observed value y_{obs}, we calculate a value y_{calc} and minimize the value of a weighted residual wR:

$$wR = \sqrt{\frac{\sum_i w_i (y_{obs}(i) - y_{calc}(i))^2}{\sum_i w_i y_{obs}(i)^2}} \tag{2.2}$$

Here each difference is multiplied by a weight w_i that reflects the uncertainties of the experimental values. In case of crystal structure analysis, the observed values would be the observed intensities in a diffraction pattern and the calculated values those intensities that were calculated based on a structural model. Model parameters will be the lattice parameters, the positions of the atoms in the unit cell, atomic displacement parameters etc. as well as experimental parameters, such as the background. Under the assumption that we have a periodic crystal, the partial derivatives of the intensity with respect to lattice parameters, atom positions etc., can all be derived analytically.

For disordered structures, the situation becomes more complicated. Except for a few special cases like stacking faults or short-range order problems, no general analytical function straightforwardly links the disorder parameter to the intensity. The intensity can still be calculated from structural models. The simulation, however, usually involves the application of random choices to generate part or all of the atom positions, and the analytical derivative of the intensity with respect to the order parameter is no longer available. A numeric calculation of the derivatives involves the repeated simulation of a new model for each parameter and is very time consuming.
Other, general optimization problems also do not have an analytical derivative. For the traveling salesman problem, for example, no derivative exists between the sequence in which the spots are visited and the resulting length of the trip.
Under these circumstances, optimization algorithms are required that can find the best solution without calculation of the partial derivatives. Evolutionary algorithms are such an alternative to least squares refinement algorithms.
These algorithms loosely mimic the evolution of plants and animals under an environmental pressure. In contrast to a least squares algorithm they usually do not operate with a single parameter set but a group of parameter sets.

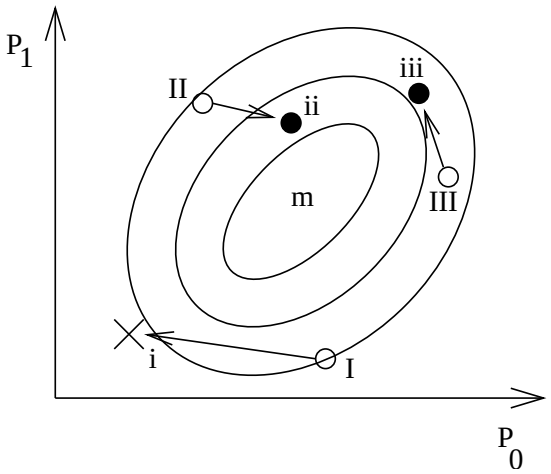

Figure 2.1: Schematic sketch of an evolutionary algorithm. Two parameters P_0 and P_1 are refined in the search for the optimum solution. The ellipsoidal lines represent lines of equal R-values. The global minimum is at point m. Members I, II, and III generate the new members i, ii, and iii by modification of the original parameter values. The new members ii and iii have an improved R-value, while member i has a worse R-value compared to the original member I.

The algorithm begins by creating a group of parameter sets, see Fig. 2.1. Each group member is a list of parameter values M:[$p_0, p_1, ..., p_n$], and for each member the R-value is calculated. Next, a new group of parameter sets is generated. Several different algorithms for this step exist. One possibility is to change each parameter by a Gaussian distributed random number with mean zero, which is added to the original parameter value. The variance of this distribution will vary from parameter to parameter. The lattice constants, for example will need another variance than the angles. In Fig. 2.1 the members I, II, and III create in turn the new members i, ii, and iii. This modification of a single parent member is comparable to the genetic mutation in

biological systems. Most systems apply a second modification, that mixes the parameter values of two different parent members, a process that roughly resembles the sexual replication.
Once the new group of members has been generated, their R-values are calculated in turn. In the example child members ii, and iii have a smaller R-value compared to their respective parent members, while child i has a higher R-value than its parent. The next task to decide which members of the old parents and children shall be retained and form the next parent group from which the next children group is to be generated. A number of different approaches exist for this task. One option is to compare a child with its immediate parent and to keep the better of these two. In the situation depicted in Fig. 2.1, this would choose the parameter sets I, ii, and iii as parents for the next generation. Another approach is to combine all parents and all children into one group. From this group the N best are chosen as parents for the next generation. In Fig. 2.1 this would give the parameter sets ii, III, and iii as parents for the next generation, since these three sets have the three lowest R-values.
As the parameter values change from generation to generations, those members survive that have better R-values. As consequence, the parameter value evolve towards the minimum R-value. If the landscape of R-values is more rugged compared to the simple situation of Fig. 2.1, one has to be careful not to refine into a local minimum instead of the global minimum. The literature on evolutionary algorithms extensively deals with this issue.

2.2 The differential evolutionary algorithm

The essential feature of the differential evolutionary algorithm, introduced by Price and Storn **?**, is the process, by which new children are generated.

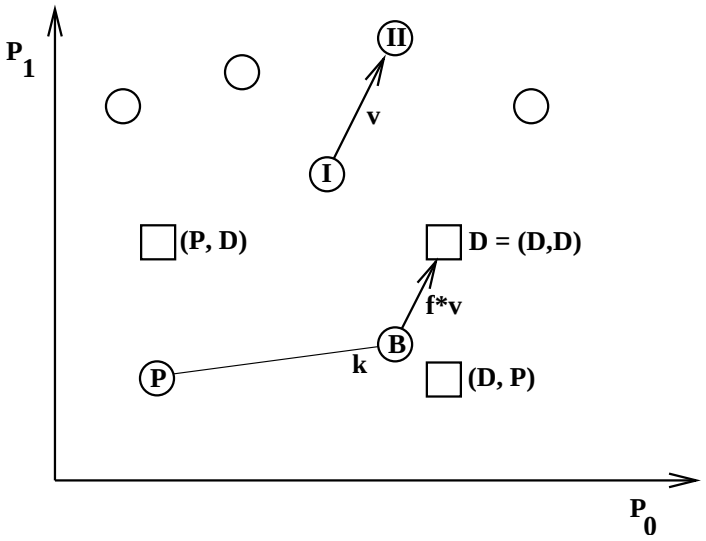

Figure 2.2: Schematic diagram of the differential evolution algorithm.

The algorithm picks in random sequence all members of the current generation, shown as circles in Fig. 2.2. The current parent has been marked by P in the figure. Another member, the base B is chosen at random. Next, two other members are also chosen at random, members I

and II in Fig. 2.2. The differential evolutionary algorithm then calculates the difference vector $\vec{v} = \vec{II} - \vec{I}$ between these two parents. This difference is the part of the algorithm that coined the name. The difference vector is multiplied by a factor f and added to the base member. All four members are different members of the population. The scale factor f is a variable that is used to control the refinement properties of the differential evolutionary algorithm. In general it should be somewhat smaller than one. The sum of the base member and the difference vector is a general point in parameter space, called the donor D. The donor D is the basic modification of th original parent vector. In contrast to other evolutionary algorithms, there is no direct connection between the parameter values of the parent P and its modification, the donor D.
After the determination of the donor D, the differential evolutionary algorithm allows for a mixing of the parameter values between the parent P and the donor D. By random choice, parameter values are taken either from the donor or from the parent. To ensure that the parent is not replicated, one parameter value is always taken from the donor D. The probability, by which the other parameters are taken from the donor D is called the cross over probability. Fig. 2.2 also illustrates the cross over process. If both parameters happen to be taken from the donor, the final child is the donor itself. If parameter p_0 is taken from the donor and parameter p_1 taken from the parent, the child will be the position labeled (D,P). Alternatively, if parameter p_0 is taken from the parent and parameter p_1 from the donor, the child will be the position labeled (P,D). Thus the cross over leads to a mixing of the parameter values of the donor D and the parent P. It depends on the refinement problem at hand, whether the cross over probability should favor parameters of the donor or the parent in order to ensure convergence.
Once children have been created for all parents, their R-values are computed. The differential evolutionary algorithm compares the R-values of each parent and its immediate child. Whoever has the lower R-value survives and is treated as parent for the next generation.
Several modifications to this basic differential evolutionary algorithm exist. The first modification concerns the choice of the donor base B. In the standard algorithm, the donor base is chosen randomly among the members of the population. As an alternative one can decide to take the current best member as donor base for all children. This will search predominantly in the neighborhood of the current best member and thus speed up the convergence into the minimum close to the current best member. If, however, this minimum is a local instead of the global minimum, chances are higher that all children will be within this local minimum as well.
Another alternative allows to add the scaled difference vector to any point along a straight line between parent and donor base, the line marked k in Fig. 2.2. A control variable *k* chooses the point. I the usual definition, the donor base is chosen if k=1 and the parent if k=0, and any point in between for intermediate values of k. In principle k is not limited to the interval [0:1], and DIFFEV does not limit your choice.
Choosing the surviving members allows for another modification of the original algorithm. Instead of a pairwise comparison of parent and child, one can also group all M parents and all N children into one group. Those M member of this combined group that have the lowest R-values survive and are used as parents for the next generation. The status of original parent or child is not taken into account. This approach will usually lead to a faster convergence, albeit at the risk of convergence into a local minimum. If the number of children is increased beyond the number of parents, the *evolutionary pressure* increases and the convergence is generally enhanced.

2.3 Termination criteria

Several different criteria may be used to terminate an evolutionary refinement.

- **Global minimum has been reached**
  If the values that correspond to the global minimum is known, the refinement can stop, once the lowest trial value falls within a defined threshold above this value. Unfortunately, in case of structure refinements, the lowest R-value cannot be known before hand and this criterium is not well suited.

- **Predefined number of refinement cycles**
  If the best R-value does not decrease for a given number of generations, chances are that we are very close to the global minimum. Unfortunately, one can never know whether the refinement may not improve after just a another few generations. This criterium may, however, be used to determine a good refinement strategy. The main variables to the differential evolutionary algorithm are the population size, the scale factor f by which the difference vector is multiplied, and the cross over probability. If refinements with different settings for these control parameters are allowed to run for a given number of generations. Those control parameters that lead to the lowest R-values after these generations can then be taken as good parameters for simular refinement problems.

- **Population statistics**
  For diffraction data, it is straightforward to calculate an expected R-value. The refinement can be stopped, once the best R-value reaches this value, or at least comes close. Another choice could be to wait until all R-values have dropped to within a defined range of R-values above the expected R-value. The corresponding parameter range may then be inspected to determine the corresponding parameter uncertainties. If the model is insufficient, one may never reach this situation. Instead one could terminate the refinement once all R-values have become very simular to each other. If the lowest R-value is significantly above the R-expected one should run the refinement again with different starting parameters, of different control parameter settings to exclude convergence into a local minimum. If the parameters refine into the same minimum, the model should be analyzed and hopefully be improved.

- **User intervention** The last choice is to run the refinement indefinetly and to terminate the refinement manually be the user. Given the many different disorder problems that DIFFEV may face, this is the main termination criterium offered at present.

2.4 Optimizing the performance

The examples in this section use the example from chapter 3. Details given here are sketchy, refer to chapter 3 for full details.
Choosing the best setup for the refinement is in itself an abstract optimization task. One should choose those values for the control parameters that will cause the refinement to find the global minium with the least amount of function calls. With regards to the differential evolutionary algorithm one needs to select the best values for:

- **Population size**
- **Scale factor f**
- **Cross over probability**
- **Choice of donor base**
- **Selection mode**
- **Local search probability**

The actual values that give the best performance will depend on the refinement problem at hand. See the discussion in chapters 2 and 3 of **?** for a further details. In the following we show a few examples that illustrate how to find good parameter settings.

2.4.1 Population size

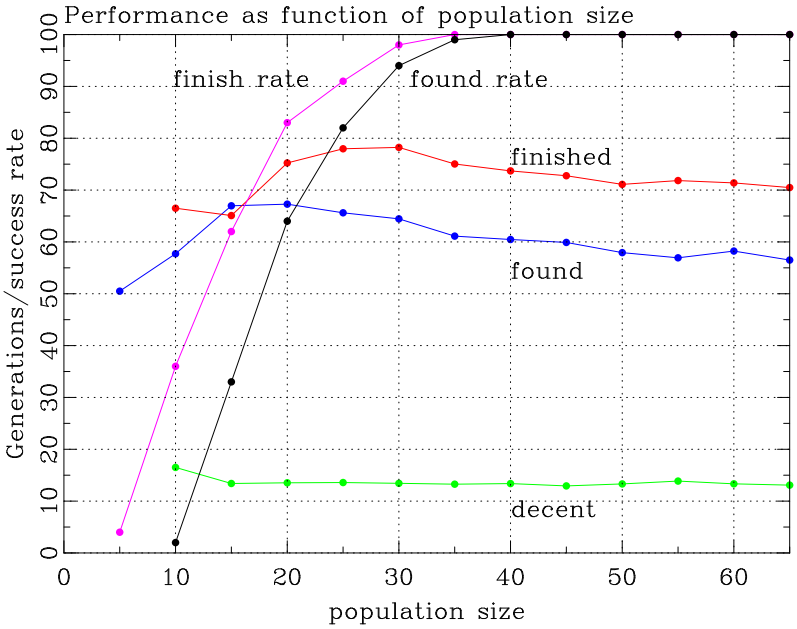

Figure 2.3: Success rate as function of population size.

If the population size is small, few permutations exist between the members of the population and there will be few locations that are searched in the parameter space.
Fig. 2.3 shows the effect the population size has on the refinement of the modified arctan function. This function requires three parameters, and parameters 2 and 3 pose a challenge due to the high noise present in the data. The true parameters for the example function were $P_1 = 100$; $P_2 = 100.23$; $P_3 = 0.1$. The refinement was run for population sizes from 5 to 65 members. The donor base was chosen at random, and the selection mode took the best members from the combined group of parents and children. The local search mode was switched off. The refinement was considered successful, if the R-value fell below a given threshold. Due

to the special function, this meant that the parameters are close to the true parameters and that the refinement will from here on converge into the global minimum. The refinement was allowed to run for 100 generations. Refinements that did not reach the global minimum or did not finish to refine to the global minimum were considered failures. At each population size, the refinement was repeated 100 times. Fig. 2.3 shows the performance as function of population size. The blue and red curves show the number of generations required to get close to the global minimum, respectively to finish refining into the global minimum. The black and purple curves show the percentage of refinements that came close to the minimum, respectively finished refining into the global minimum within the allowed 100 generations.
A population size of about 40 members is needed to ensure a 100% success rate. At smaller population sizes, a larger fraction of the refinements does not find the minimum and thus the refinement cannot be considered satisfactory. If the population size is increased beyond 40, the number of generations required to get close to the minimum, respectively to finish refining into the global minimum decreases slightly. The increase in function calls due to the population size increase is, however, higher than the gain obtained by faster refinements.

2.4.2 Scale factor and cross over probability

To test good values for these two parameters, the population size of 40 was adapted according to the observations on the population size. Both factors were chosen randomly in the interval [0:1]. As in the previous investigation, the refinement was considered successful, if the parameters reached the true values within 100 generations.

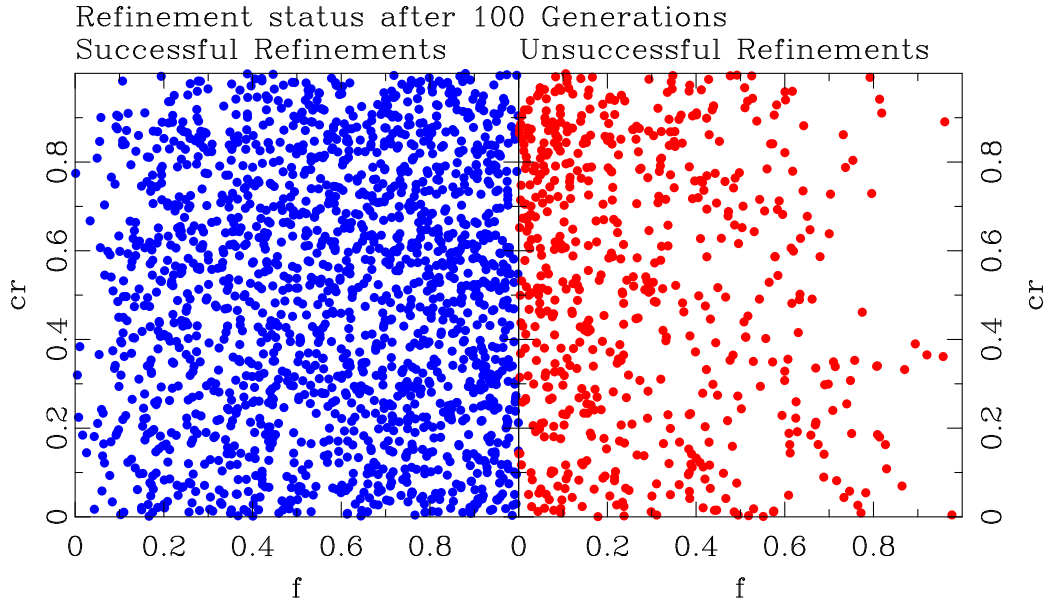

Figure 2.4: Success rate as function of scale factor f and cross over probability cr.

The figure shows that successful refinements of this example function require a scale factor that should be closer to 1. For very small scale factors, the algorithm effectively searches in the local environment of the donor base instead of a wide parameter range. In this example, this increases the chance of refining into a local minimum.

2.4.3 Local search probability

A cut through R-value space along P_3 at P_1 and P_2 at their respective optimum values shows a narrow minimum around the optimum value of 0.1. This behavior might indicate that a local search around the members might enhance the convergence into this narrow minimum. To test this, two different test were performed, in which the local search probability was systematically varied from 0 to 1.
For both tests with different local search options, otherwise identical control variables were used. The population size was set to 40 members, the scale factor was 0.81 and the cross over probability 0.8. The donor base was chosen randomly and the best members of the combined parent/children group were selected. For each local search probability the refinement was repeated 100 times.
Figure 2.5 shows the number of generations required to find a parameter combination close to the global minimum (blue), the total number of generations required to refine very close to the global minimum (red), the number of generations required from the time the first parameter set is close to the global minimum until the global minimum is reached (green), the percentage of refinements that came close to the minimum (black, and the percentage of refinements that finished refining into the global minimum (purple).

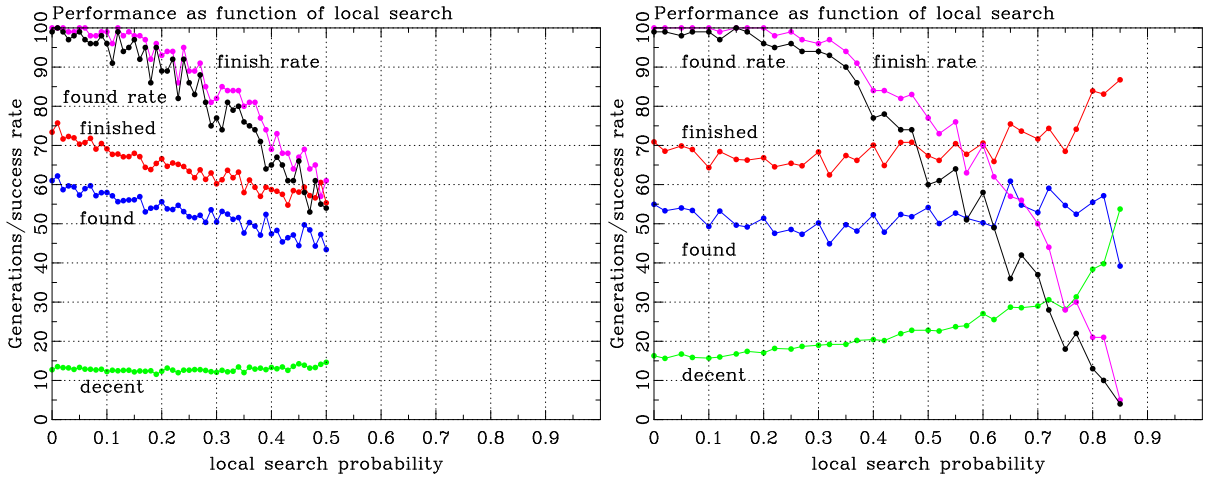

Figure 2.5: Refinement behavior as function of local search probability. Left, local search sigma adapted to 0.02 of the parameter spread, Right, local search sigma at fixed values (0.1; 0.1; 0.002).

The refinement with adaptable local search sigma showed that the number of generations required to find the global minimum decreases slightly with increasing local search probability. Beyond a probability of roughly 15%, the percentage of successful refinements starts to decline quickly. For the refinements with fixed local search sigma, the effect on the number of cycles is less pronounced. All in all one can see that the local search does not provide a clear effect on the refinement efficiency. In general, the original differential refinement algorithm seems to work better.

2.5 Invoking the slave program

The typical refinement requires the following steps:

- definition of the problem
- initialization
- A loop over the required generations within each loop the simulation of all crystal structures and the calculation of all cost functions / R-Values
- A comparison of old and new cost function values / R-values and generation of new trial parameters.

In the first step the number of population members and the number of refineable parameters and their allowed range must be specified. DIFFEV furthermore expects the definition of log files to keep track of the refinement. See the example in 3 for details on the commands.
The initialisation step will assig starting values to all parameters that you want to refine. DIFFEV expects you to provide for each parameter a range within which the parameters are allowed, and a (narrower) range for the starting distribution. The initialisation will place the starting parameter values with an even random distribution with the starting window. See 3 for further details.
As of version 5.3.0 the trial parameters need not be written to a file on the disk. If DIFFEV is used within the DISCUS SUITE, the trial parameters can be transfered directly to the slave program via the command: 'init silent'.
Within the loop the slave program that will simulate the crystal and evaluate the cost function / R-Value must be started. As of version 5.4.0 a unified command 'run_mpi' should be used for this purpose. The command will recognize whether DIFFEV runs as stand alone program or as part of the discus_suite (The reccommended style) and if the program uses MPI to process the slave program in parallel. An identical macro for the slave program can be used in all cases.
The slave program is actually started by DIFFEV as:

```
discus -macro discus_main.mac PWD kid indiv > discus_log.xxxx.yyyy
```

If DIFFEV is part ot the DISCUS SUITE, the suite will internally switch to the discus section to execute the macro 'discus_main.mac'. If DIFFEV is run as a stand alone program, it uses a system call to start the DISCUS program with the command line option to execute the macro. In both cases identical DISCUS macros can be used. The one exception is the 'silent' option for the 'init' and 'compare' commands. This option is available within the DISCUS SUITE only. For the stand alone version of DIFFEV no direct communication exists between DIFFEV and DISCUS. The trial parameters and the R-values need to be written to disk to communicate between the programs. It is strongly reccommended to use the DISCUS SUITE.

2.5.1 Parallel Refinement via evolutionary algorithms

As the refinement in DIFFEV is based on a large population, it naturally lends itself to parallel performance. This version of DIFFEV has a build in support for a MPI based parallel distribution. Within this model, each of the children is simulated / calculated in parallel to each other.

On a large scale computing facility you will typically submit your job to a queue and you will have to request a specific number of nodes and over all wall time. As example we will use the queue system at the high performance center in Erlangen. Take this as a general guide and refer to your system for changes that you might have to do.
Within this general set up the simulations of all crystal structures can be performed independently and thus in parallel. If the calculation of the r-value takes considerable time this can be performed in parallel as well. This parallel calculation must only be carried out once the simulation is finished.
If the simulation involves small crystal structures, or a distribution of defectes and/or sizes, you might need to simulate several individual structures for each member of the population. Again these simulations can be performed in parallel.
MPI is a widely distributed system to run jobs in parallel. You can even use it effectively on a multi core computer. To use DIFFEV with MPI you need to turn on the MPI option during compiliation or run the MPI version from the DISCUS download server.
A set of command to run the refinement in parallel will be like:

```
set prompt, redirect
@diffev_setup.mac       ! all the set up commands
init                    ! Initialize the population
do i[0]=1,10            ! Make a loop over 10 refinement cycles
   run_mpi discus, discus_main.mac, 1, discus_log
   run_mpi kuplot, kuplot_main.mac, 1, kuplot_log
   compare
enddo
exit
```

The `run_mpi` command instructs DIFFEV to run the program specified as first parameter in parallel. The second parameter is the macro that the slave program will execute. The third parameter is the number of individual repetitions that the slave program shall perform for each child. The last parameter is the base file name for a log file into which the slave program writes all of its standard output. On a Unix system you can avoid the log file by specifying it as `/dev/null`.
The slave program is started by DIFFEV as:

```
discus -macro discus_main.mac PWD kid indiv > discus_log.xxxx.yyyy
```

Here PWD is a place holder for the current working directory in which DISCUS shall work. KID is the number of the child within the population, and indiv is the number of the individual repetition. The log file name is extended by two four digit numbers xxxx and yyyy, which hold the values of kid and indiv.
If DISCUS must perform several simulations for each child, increase the value of the `nindiv` variable. Make sure that DIFFEV and DISCUS share the identical value of `nindiv`. Each instance of DISCUS will simulate exaclty one individual calculation for each member.
The number of parallel instances that will run depends on your MPI system. On a stand alone computer with several cores you will start DIFFEV as

```
mpiexec -n 8 diffev < diffev_main.mac
```

In this example, MPI will reserve 8 cores for your job, execute diffev, which must be located in a directory where the standard PATH environment variable will find it. The macro `diffev_main.mac`

must contain all instructions for DIFFEV, including the `set prompt, redirect` and final `exit` commands.
The MPI scheduler does not know how long a calculation by the slave program may take. This might cause an idle state for some or almost all CPU's once almost all children in a given generation have finished. This will cause your system not to run as effeciently as possible. Some of this idle state cannot be avoided. You can reduce the idle state if you keep the number of requested CPU's much smaller than the number of simulations required for one generation. Under these settings, each CPU will have to run several simulations and you can hope that the work load will average out. To average the calculations DIFFEV sends the CHILDREN times NINDIV calculations in a double loop. The inner, faster index is over all CHILDREN, the slower over all individual calculations.
Future releases of DIFFEV will use an optimized scheduling program currently under development at Argonne National Laboratory.

Chapter 3

Example refinements

The example in this chapter will be used to explain in detail the commands and control variables that DIFFEV uses. The example is a simple function y = F(x). While this restriction eases the display of the data, and refinement process, it does not impose a restriction on the settings. The macros and input data needed to run this example are found along with the DIFFEV source code, in the directory *diffev/TESTS*.

3.1 Noisy example function

The function in Fig. 3.1 shall be the data set for the refinement in this section. The function from which these data were calculated is:

$$y = P_1 atan\left(\frac{|x - P_2|}{P_3}\right) \tag{3.1}$$

where atan is the arc tangens or inverse tangens function, and P_i are free parameters that determine the behavior of the function. The function has a minimum y=0 at $x = P_2$. The width of this minimum depends on P_3. For small values of P_3 the minimum is smaller. Far away from the minimum, the function is almost constant at $y = \pi/2P_1$. The data were generated by adding a Gaussian distributed noise to each data point.
This data set proves to be a very hard challenge for any least squares algorithm. Unless the starting values are very close to the actual values, the least squares algorithm will not find the global minimum. The very high noise level effectively prevents the algorithm from finding the position of the minimum. Instead the R-value is minimized by setting P_3 to the smallest positive value. At this level, the width of the function is reduced to one point, if any. Furthermore, the slope of the function is essentially zero for all x, and the position of the minimum does not influence the R-value. The R-value is, however, significantly above the R-value that is achieved for the ideal parameter values, show in Fig. 3.2.
To refine the function with DIFFEV, we need to define a couple of items. These include:

- Population size
- Limits, starting range etc. for the parameters P_i
- Values for refinement control variables

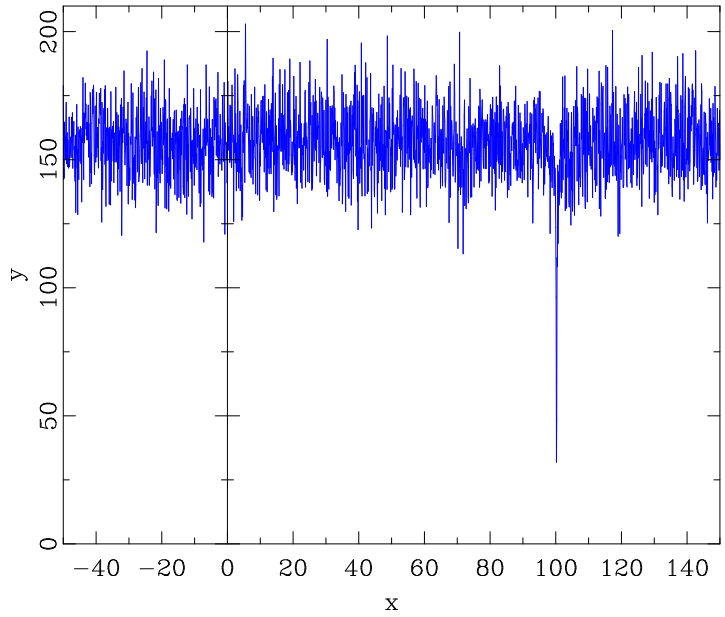

Figure 3.1: Example data

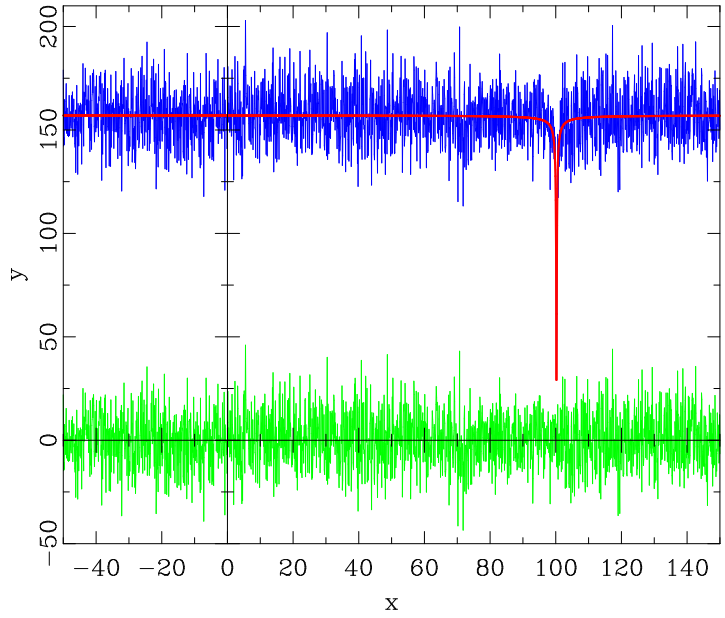

Figure 3.2: Example data in comparison to the ideal function. The R-value between the observed and calculated function is 8.18%.

- location and names for files

The optimum size of a population is not always straightforward to estimate. Since the minimum at P_1 is very narrow, a fairly large population size is advisable. A test shows that the population should consist of approximately 40 members to ensure that the minimum is found. The corresponding commands would be:

```
pop_n[1]   = 40
pop_c[1]   = 40
pop_gen[1] =  0
```

Here the number of members and children was set to the same value, there is no fixed need to do so, unless the selection mode is set to compare a child with its immediate parent. With the last statement the current generation number is set to zero. If a refinement is to be continued, the generation number can be set to the corresponding value.
The next group of definitions includes the number of parameters, and then for each parameter a suitable name, hard boundaries, a starting range and definitions how to handle parameter behavior at the hard boundary and in a local search.
In the current example, we have three parameters, all of which are floating numbers. None of the parameters has any absolute lower or upper boundary imposed by external rules. Parameter P_3 must not be equal to zero, this we will handle later as a constraint. Sensible hard limits for the parameters could be:

parameter	lower boundary	upper boundary	comment
1	0	200	All data points are larger than zero.
2	-50	150	Minimum must be somewhere within the observed x-range
3	-1	1	Include zero to use constraints

If one has good estimates for the parameter values, these can be used to limit the initial spread of the population. A narrow spread of the initial population around the expected final value will speed up the convergence. Should these estimates prove to be wrong, however, the refinement will take extra long or may fail altogether. Here we will not impose any prior knowledge on any of the parameters and use the full range set by the hard boundaries.
During the initial refinement stages, the differences between the members will be large and chances are that the donor falls outside the allowed hard boundary interval. DIFFEV corrects this situation by setting the violating parameter to a Gaussian distributed value with mean at the hard boundary. The respective half of the Gaussian distribution that falls inside the boundary range is taken as valid region. The user can set the sigma for this distribution and define whether this sigma shall remain constant or be adapted during the refinement. For parameters 1 and 2 we will set the initial sigma to 1, and for parameter 3 to 0.02. In later refinement cycles the value of sigma can be adapted to a fraction of the total parameter spread, in our example to 0.2 of the parameter spread.
Similarly, the sigma for a local search is fixed to starting values and adapted to a fraction of 0.01 of the total parameter spread.
For parameter 1 this would be set by the commands:

```
pop_name    1,height
type real,1
```

```
pop_xmin[1] =   0.0
pop_xmax[1] = 200.0
pop_smin[1] =   0.0
pop_smax[1] = 200.0
pop_sig [1] =   1.0
pop_lsig[1] =   0.1
adapt sigma , 1,0.2
adapt lsigma, 1,0.01
```

Since parameter P_3 forms the denominator of Eq. 3.1, its value must not be equal to zero. Check the function with a couple of different parameters, or simple mathematical analysis would show that the product P_3 P_1 must also be larger than zero to produce the observed dip. A clear lower limit for P_3 can, however, not be given. Therefore, P_3 was allowed in the interval [-1:1], and we just have to exclude a value of zero.
If the problem is difficult to solve, or if one wants to get a quick estimate of one parameter, one can choose to refine just a subset. Since we only have three parameters, we will refine all at the same time.

```
constrain p[3].ne.0.0
refine     all
```

The next group of definitions concerns the control variables. DIFFEV offers the two basic variables, the scale factor, by which the difference vector is multiplied, `diff_f[1]`, and the cross over probability `diff_cr[1]`. Both are limited to the interval [0:1]. For this refinement problem, the actual value of the cross over probability does not matter, the scale factor should be closer to one to ensure successful refinement.
A variation of the basic algorithm allows to add the difference vector to any point along the line between parent and donor base. In this problem, the location does not influence the refinement, and we choose the value of 1, which corresponds to the original algorithm. Also, the value of the local search probability does not affect the search efficiency in this example, as long as its value is smaller than a value of roughly 0.3, and here it is set to zero.

```
diff_cr[1]  = 0.8
diff_f[1]   = 0.81
diff_k[1]   = 1.0
diff_lo[1]  = 0.0
```

The next choice concerns selection of the donor. One can either choose the current best member as donor, or choose the donor at random. Here the donor is chosen at random.
If the dependency of the R-value on the parameters is a reasonably smooth distribution without too many false minima, one can accelerate the convergence by taking the combined group of parents and children and to choose the best members to form the next set of parents. Here this is a good choice.

```
donor      random
selection  best,all
```

Finally we need to define filenames for the trial files, the results and the two log files. To keep the output nicely sorted, these files are written into a subdirectory `DIFFEV`.

```
trialfile  DIFFEV/Trials
restrial   DIFFEV/Result
logfile    DIFFEV/Parameter
summary    DIFFEV/Summary
```

This concludes the basic setup. Prior to the main refinement loop, we just have to generate the initial parameter sets, which is done by the command `initialize`, which does not take any parameters. This command sets the current generation number to zero and writes a first version of all trial files.

```
init
```

The main refinement follows, usually in a loop over several generations. Within the loop, the `system` command must be used to execute the slave program or programs that calculate the R-value. Once these are done, the `compare` command reads all R-values, and generates the next generation of trial values. It also updates the log files. For a fixed number of refinement cycles this could be a construction like:

```
do i[0]=1,40
  system ./arctan
  compare
enddo
```

A loop with 40 cycles is executed, using the external program "./arctan" to calculate the R-values. In a UNIX environment the leading "./" specifies that the program `arctan` is found in the current directory. Without this specifier, the UNIX shell would search for the program in the directories specified by the value of your *PATH* variable.
An indefinite loop that requires manual intervention could be:

```
variable integer, terminate
terminate = -1
fopen 1, CONTINUATION
fput  1, terminate
fclose 1
do while(terminate.eq.-1)
   system ./arctan
  compare
  fopen 1, CONTINUATION
  fget  1, terminate
  fclose 1
enddo
```

The macro initially sets the variable `terminate` to "-1" and writes this to the file `CONTINUATION`. The loop is executed until the value of the variable is no longer "-1". At each cycle the content of file `CONTINUATION` is read into variable `terminate`. Thus, the macro will stop, if you edit this file and change the number stored within.
Finally, the following macro checks the R-value and reacts accordingly. The commands in this macro were used to generate the performance tests in chapter 2.

```
variable integer,cycle
cycle = 0
do
  system ./arctan
  compare
  cycle = cycle + 1
enddo until (bestr[1].lt.0.0817830 .or. cycle.gt.100)
```

The loop is run indefinetly, until either the R-value has fallen below a threshold, or until the number of cycles exceeds 100.

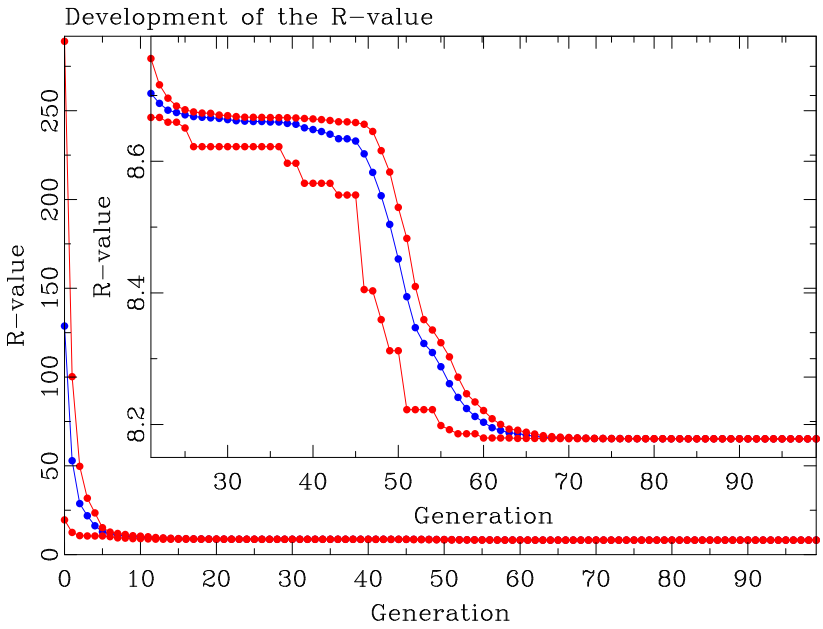

Figure 3.3: R-value as function of refinement generation. The Figure shows the best, worst R-value (red curves) and the average R-value (blue)

A typical refinement run is shown in Fig. 3.3. The refinement is set according to the control variables described in this section. Initially, the best and worst R-values quickly drop to values around 9 to 12 %. In these cycles, those parameter values that are really far off are eliminated. Thereafter, the real refinement starts. Up to generation 37 little change occurs. At this stage, the first members have found the global minimum, and after generation 45 all members begin to converge into the global minimum, which is reached after generation 70. Thereafter, no significant progress occurs.

The main effect on the R-value is caused by the value of the parameter P_1, Fig. 3.4, since this parameter lowers or raises the function over the whole x range. Accordingly, the refinement quickly finds a value close to the right value. Notice that around generation 50, the parameter values spread before the final value is found. It is around these generations that all members find the global minimum and during the adjustment of parameter P_3, the other parameters spread out for a few generations.

For the first roughly 15 generations, the value of parameter P_2, Fg. 3.5, hardly affects the R-value. As long as the position of the minimum is not very close to the true value, its position hardly matters. From Generation 20 to 45 more and more members find the correct position.

Parameter P_3, Fig. 3.6, shows the most unusual refinement behaviour. In the first few generation, all members with negative values of P_3 are eliminated. The parameter then refines to values between zero and 0.02, much lower than the true value. As long as the position of the minimum is not found, lower R-values result if parameter P_3 is as close to zero as possible. To get out of this local minimum, a large population size is required. Once the correct position is found, parameter P_3 also refines to its correct value.

At this stage, the refinement is finished and the calculated curve is that of Fig. 3.1. Admittedly, the difference between the R-values around generations 30 at 8.65% are not very significantly

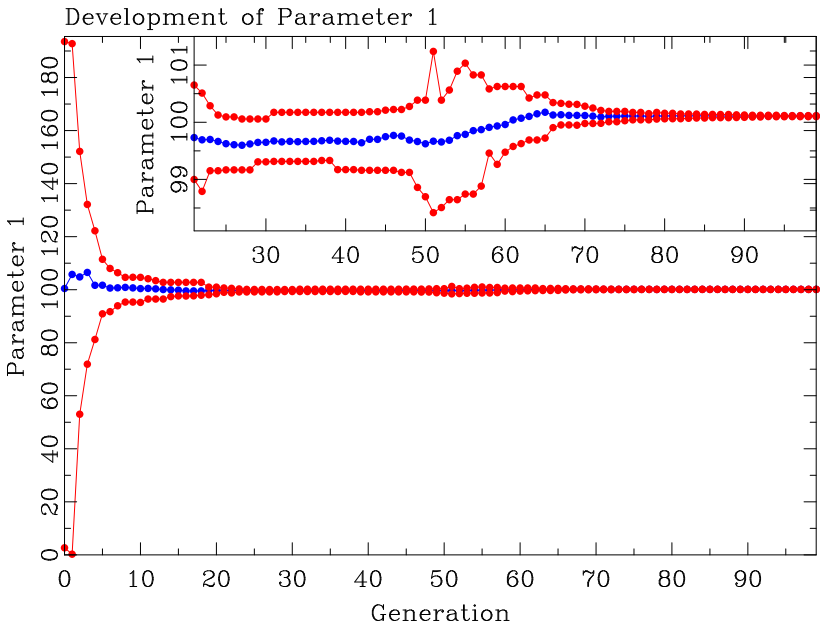

Figure 3.4: Parameter P1 as function of refinement generation. The Figure shows the best, worst R-value (red curves) and the average R-value (blue)

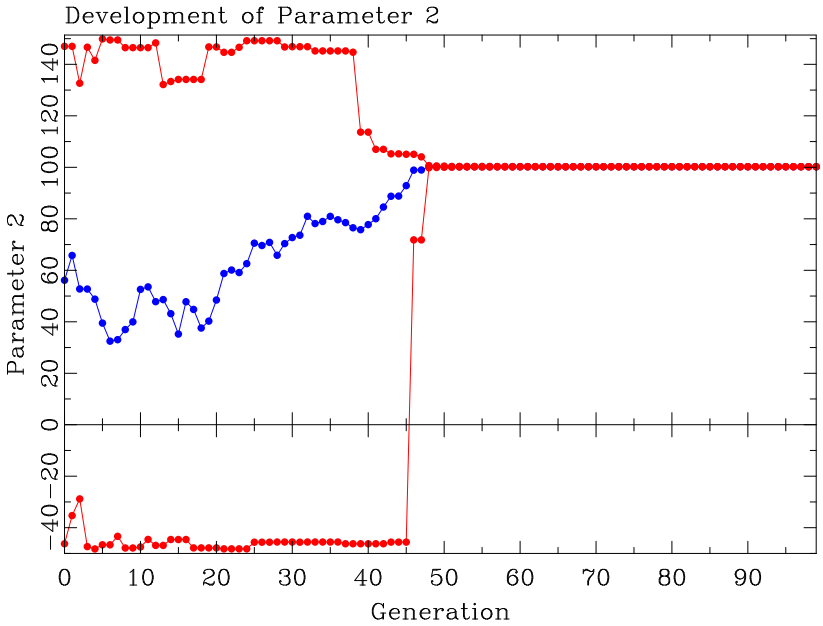

Figure 3.5: Parameter P2 as function of refinement generation. The Figure shows the best, worst R-value (red curves) and the average R-value (blue)

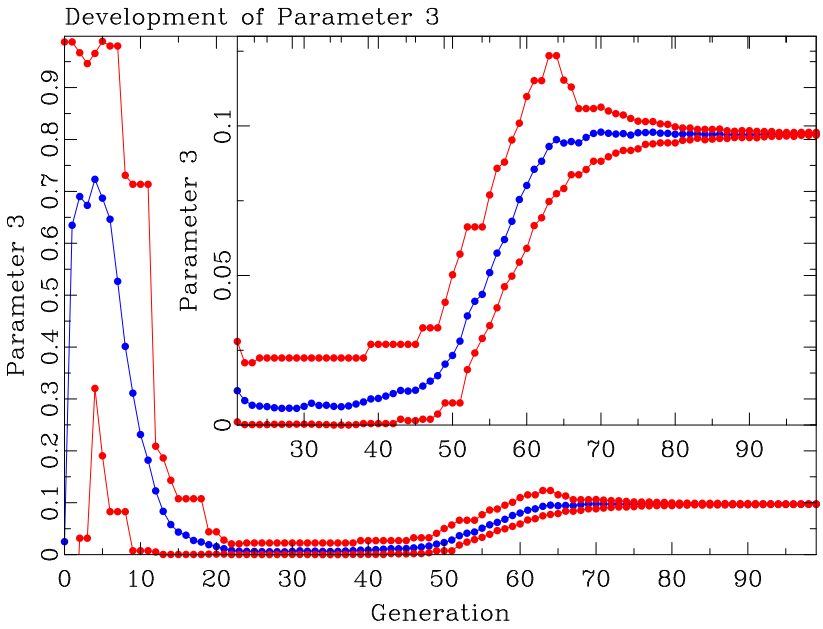

Figure 3.6: Parameter P3 as function of refinement generation. The Figure shows the best, worst R-value (red curves) and the average R-value (blue)

worse that the final R-value of 8.18%. This is due to the large amount of noise in the *experimental* data. These conditions were, however, chosen on purpose to illustrate the ability of the differential evolutionary algorithm to jump out of local minima, even under adverse conditions.

Appendix A

DIFFEV commands

A.1 News

2017_Sep

Throughout the program the internal calculation of random numbers was changed to the FORTRAN 90 intrinsic function.

2017_July

DIFFEV has been modified to log the status of the random number generator at the beginning of each slave calculation. This status is documented internally for the current best member of the population. Once an 'exit' command is executed, DIFFEV will write a macro called "diffev_best.mac" that can be used to recreate the current best solution.

2017_Jan

An unfortunate typing error in News/2016_Oct regaring the new refinement variable ref_para[1...] (was misspelled as ref_param[1...]) is corrected in the on-line help.
Another typing error caused a an error in the macro parameters transferred with a NON MPI command: run_mpi, making these not backward compatible. This has been fixed.

2016_Dec

At a few select points colors are introduced into the output. Currently these are just the error messages.

2016_Oct

Global variables have been introduced that use the same syntax as user defined variables. This include just "pi" and variables related to the refinement. DIFFEV sets the value to these variables: REF_GENERATION Current generation REF_MEMBER Current population size REF_CHILDREN Current children size REF_DIMENSION Number of parameters REF_KID

Current child Updated for DISCUS and KUPLOT only REF_INDIV Current individuum Updated for DISCUS and KUPLOT only REF_NINDIV Number of individual repetitions ref_para[1..] Current trial parameters for current child

2016_june

DIFFEV may now be interruted gracefully with a CTRL-c. This will cause DIFFEV to shut down MPI if active.
The 'run_mpi' command can now be used as a generic interface to identical slave macros, regardless of the MPI status.

2016_april

The initialise command was augmented by a second form to initialise just the logfiles, see ==> 'logfile', 'summary'. This might be helpful if you want to reset the refinent cycle to a smaller generation number. For very lengthy refinements with a few thousand refinement cycles and many parameters, a continuation will take appreciable time to read all previous cycles. If the actual development accross the cycles is not relevant to you you can reduces the file sizes drastically be a sequence like: pop_gen[1] = 1 initialise logfile
New command 'lastfile'
This new command creates a short copy of the ==> 'logfiles'. This short form contains the parameters just for the last refinement generation.

A.2 Synopsis

```
Description ! A description of the program
News        ! Information on recent changes
allocate    ! Allocate array sizes
adapt       ! Adaption of global/local search width
backup      ! Backup current best solutions
compare     ! Compares the results of the current population
constraint  ! Defines a constraint condition
deallocate  ! Deallocate array sizes
dismiss     ! Dismiss the worst parents; replace by new children
donor       ! Defines which donor to use
fix         ! Fixes a parameter
initialize  ! Initializes the generation zero
logfile     ! Defines the file name for the parameters for each generation
pop_name    ! Defines names for the individual parameters
refine      ! Defines which parameters are refined/fixed
restrial    ! Defines the file name for the current R-values
run_mpi     ! Run the cost function program in parallel through MPI
selection   ! Defines how children/parents survive into next generation
summary     ! Defines the file name for a summary of the parameter changes
trialfile   ! Defines the file name for the current parameters
type        ! Sets the numerical type for a parameter (integer or real)
write       ! Writes new Children or new GENERATION file
```

A.3 Description

The program DIFFEV uses the differential evolution algorithm to refine a set of parameters to a set of observations.
DIFFEV provides the handling of the parameters and their evolution. An external program must be used to calculate the cost function or R-value that corresponds to a given set of parameters.
The differential evolution algorithm compares simultaneously the resulting cost function for several sets of parameters. The number of parameter sets is called the population. For each member of the population a set of parameters is used. This is called a parameter vector. Its dimension depends on the model that you need to describe. In order to describe a parabola you would need three parameters. The whole set of parameters is called a generation.
For each generation, the cost function is evaluated for each member of the population. The next generation is determined from the current parent generation through the following procedure: Loop over all members - Choose a member (at random or at will), this will be the donor base. - Set a point along the line between current member and donor base to create the effective donor base ==> diff_k[1] - Select two other members by random choice - Loop over all parameters: - Take the difference between the corresponding parameters of the two other randomly chosen members. - Multiply this difference by a user provided value ==> diff_f[1]. - Add the difference vector to the effective donor base to create a parameter set called the donor. - One parameter of the donor is always chosen for the child. All other parameters are then randomly chosen from: Either: the parent Or : the donor - The probability for this choice is weighted by a user provided probability ==> diff_cr[1].
Once a new generation has been determined the corresponding cost function is calculated.
Next the selection process determines, which current children survive into the next generation. This is done either by: - Direct comparison between a parent member and its immediate child. Only those new members survive, whose cost function is less than that of the parent. Otherwise the parent is retained. - All parents and all children are pooled into one set. From this set the best members survive, irrespective whether they were a parent or a child.
Such an algorithm is able to search for parameters if a standard refinement algorithm fails or is difficult to adapt. This might be the case for: - Undefined parameter values within the possible range. The parameter P may for example NOT be equal to 1 for a function: 1/(1-P) but values larger AND smaller are allowed. - The calculation of the cost function involves existing extensive algorithms or several different programs. - The calculation involves the averaging of several calculations that rely on data created by (Gaussian-) randomly distributed parameters.
Files
DIFFEV uses several files to store the parameter values, the current status etc these are:
"GENERATION" Fixed file name in the current directory!
The file "GENERATION" contains ten lines with the current generation number, and all relevant file names.

```
# generation members children parameters
     158        45        90         4
# trial file
DIFFEV/Trials
# result file
DIFFEV/Results
```

```
# log file
DIFFEV/Parameter
# summary file
DIFFEV/Summary
```

Trial files
DIFFEV writes a short file for each member that contains the current parameters for one member. The user defined filename is appended by a four digit member number. The first lines contain the information on current generation number, the number of members in the population, the number of children and the number of parameters

```
# generation members children parameters
     181        45        90         4
# current member
    1
# parameter list
    0.8284027688E-02
    0.5815573883E+02
    0.2050311089E+01
    0.3000000000E+01
```

Result files
DIFFEV expects to read the R-values for each member from a short file. The user defined filename is appended by a four digit member number.
The file must contain the member number and the R-value in free format within the first line.
Parameter files
For the R-value and for each parameter a SPEC type file is written that contains all old R-values and parameters, respectively. Each generation makes up a scan. The first column is the member number, the second column is the R-value and the third column is the respective parameter value. The base name of the parameter files can be set by the user via command ==> 'logfile'. This base name is appended by a four digit number with leading zeros. Parameter file no 0000 has the same structure, except that both column two and three are the R-values.
Last file
A short copy of the parameter file that contains the parameters just for the last generation.
Summary files
For the R-value and for each parameter a SPEC type summary files contains a single scan. Each generation creates one line within the scan. Five values arew written to each line. The first column is the generation number For the R-value and each of the parameters four further columns are written. The first of these is the average value, the second the minium value, the third the maximum value and the fourth the sigma of the parameter distribution. The base name of the summary files can be set by the user via command ==> 'summary'. This base name is appended by a four digit number with leading zeros. Summary file no 0000 has the same structure, except that both column two and three are the R-values.
Further help topics are:

Basic_Example

This example illustrates the commands to refine the three parameters that describe a parabola: y = P1*x**2 + P2*x + P3

You will find the data and the macros in the diffev/Example directory within the program source directories.

```
======================diffev.mac=======================================
#
#
pop_gen[1]  = 0         # initialize the current generation to zero
#
pop_n[1]    = 15        # The population shall have 15 members
pop_c[1]    = 15        # The population shall have 15 children
pop_dimx[1] = 3         # We need three parameters
#
pop_name      1,square  # The parameter is called "square"
#
type          1,real    # Parameter 1 is a floating number
#
pop_xmin[1] = -1.0      # The first parameter is restricted to the
pop_xmax[1] =  1.0      # range -1 to +1
#
pop_smin[1] = -0.8      # The starting parameters of the first parameter
pop_smax[1] = +0.8      # are restricted to the range -0.8 to +0.8
#
pop_sig[1]  =  0.02     # The sigma of minimum parameter "noise".
pop_lsig[1] =  0.002    # The sigma for local searches
#
adapt  sigma, 1, 0.025  # After generation 0 the value of pop_sig[1]
#                       # is adjusted to 0.025*(largest parameter -
#                       #                       smallest parameter )
#
adapt lsigma, 1, 0.0025 # After generation 0 the value of pop_lsig[1]
#                       # is adjusted to 0.0025*(largest parameter -
#                       #                        smallest parameter )
#
pop_name      2,linear
pop_xmin[2] = -2.0
pop_xmax[2] =  2.0
pop_smin[2] = -1.8
pop_smax[2] = +1.8
#
pop_name      3,constant
pop_xmin[3] = -5.0
pop_xmax[3] =  5.0
pop_smin[3] = -4.8
pop_smax[3] = +4.8
#
constraint  p[1].lt.1.3 # Several constraint may be imposed on the
constraint  p[2]+p[3].gt.0.0
#                       # parameters. Here the first parameter must be
#                       # less than 1.3. In the second constraint condition,
#                       # The sum of the second and third parameter must be
#                       # greater than zero.
#
diff_cr[1]  = 0.9       # The cross over probability is 90%
diff_f[1]   = 0.81      # The difference vectors are multiplied by 0.81
diff_k[1]   = 1.0       # For diff_k = 1, the difference vector is added
#                       # to the donor, for diff_k = 0 to the parent.
diff_lo[1]  = 0.1       # In 10% of all cases, a member creates its child
#                       # not by diffev algorithm, but from a local Gaussian
#                       # distributed search.
#
trialfile  Trials       # The temporary file of the current parameter
#                       # values
restrial   Resultate    # The temporary files of the resulting cost
#                       # function. Must be written by the slave program.
```

```
logfile    Parameter    # The log file for the parameters
summary    Summary      # A shorter log file
#
init                    # Initializes diffev, Generation zero is written.
#
do i[1]=1,15            # A loop over 15 generations
  sys kuplot < kcompare.mac > /dev/null
#                       # diffev starts the kuplot program with input
#                       # from file "kcompare.mac". This macro instructs
#                       # kuplot to read the 'trialfiles', to calculate the
#                       # cost function for each member and to write the
#                       # results into the 'restrial' files.
#
  compare               # diffev reads the 'restrial' files, compares the
#                       # cost functions of children and parents and
#                       # creates the next generation.
#
enddo                   # End of the loop
```

Increase_Dimension

Here is an example for a macro that should be used to increase the number of parameters to be refined:

```
variable integer, ipar           # just a nice variable name
pop_dimx[1]    = pop_dimx[1] + 1 # increase dimension
ipar           = pop_dimx[1]     # copy into variable

pop_name         ipar,cube       # Define name etc for new parameter
pop_xmin[ipar] = -5.0
pop_xmax[ipar] =  5.0
pop_smin[ipar] = -4.8
pop_smax[ipar] = +4.8
type real,       ipar
refine           ipar            # Set refinement flag
init             ipar            # Initialize just this new parameter
dismiss          pop_n[1]/2      # Set R-value of half the population
                                 # to a very high value, thus they will
                                 # be replaced in the next generation
```

Hints

These are some hints regarding useful parameters. They are derived from the authors experience and are to be carefully adopted.
The population size MUST be at least 4!
The population size should be at least about ten times as large as the number of parameters, twenty times will give you a good sampling of the parameter space. A smaller population runs the risk of converging into a local minimum.
The cross over probability ==>diff_cr[1] should be about 0.8 A smaller value (lets say around 0.3), seems to prevent convergence, While a small value prevents the special properties of the differential algorithm to be applied at all. At diff_cr[1]=0, all children would always be identical to their parents!
The multiplier for the difference vector ==> diff_f[1] should be around 0.8 A small value prevents the children from being very different from their parents, while a large value > 1.5(?) seems to prevent convergence, since all children always jump too far away from their parents.

selection mode If the dependency of the cost function/R-value on the parameters is (or seems to be) fairly straightforward, the convergence is much faster if you choose the selection best,all scheme. By increasing the number of children in comparison to the parents, the selection pressure also increases and the algorithm will move faster into the minimum. This will, however, also happen if you happen to be close to a local minimum, instead of the global minimum!
I cannot give a hint regarding the number of generations required. This depends too much on the problem at hand, parameter correlation, the initial choice of parameters etc.

A.4 News

2013_May

The run_mpi command has been augmented by a "socket" qualifyer, which will cause DIFFEV to run the application program controlled via a socket. This should speed up the process on a multi-core/multi-cpu system.

2011_June

The program is now a fully functional fortran2003 program. There are no changes that must be followed by the user, but important differences exist in the memory allocation. The internal arrays that hold the population are now all allocated automatically, when you define the number of parameters to be refined, the size of the population and the constraint equations. Most of the time you will not have to bother with these computational details. If you whish, you can allocate appropriate array sizes, see ==> 'allocate', 'deallocate'.
With this version, the program allows the user to change the population size and the number of parameters to be refined during a refinement cylce.
If the number of parameters to be refined in increased during a cycle, the program will automatically write new child parameter sets and patch the logfile and summary file.
See the entry ==> "Description" ==> "Increase_Dimension" for an example.
Related to this new feature, the ==> "initialize" command now allows to (re-)initialise an individual parameter.

A.5 allocate

allocate
allocate "default"
allocate "constraint", <**max_constr**>
allocate "population", <**max_members**>, <**max_parameters**>
allocate "show"

DIFFEV allows to to allocate memory for the arrays needed to store the population and the constraints. DIFFEV will dynamically allocate the population size, the number of refinement parameters and the number of constraints. Thus one often may not have to use this command. If previously allocated arrays are reallocated, DIFFEV tries to save the old values and you can continue to use these. If the new array sizes are smaller than the previous ones, this can

obviously not be done. DIFFEV will perform the new allocation, but all old data are lost. A short warning will be printed.
allocate
allocate "show"

Without parameter or with the parameter "show", the allocate command shows the current memory allocations.
allocate "default"

Allocates all array sizes to default values.
allocate "constraint", <**max_constr**>

Allocates the maximum number of constraints that DIFFEV shall handle.
allocate "population", <**max_members**>**,** <**max_parameters**>

Allocate the maximum population size and the maximum number of parameters that can be refined.

A.6 adapt

adapt "sigma" ,<**parameter_number**>**,[,**{**"yes"**|**"no"**|<**value**>}**]**
adapt "lsigma",<**parameter_number**>**,[,**{**"yes"**|**"no"**|<**value**>}**]**

DIFFEV uses to "sigmas" to handle special situations.
The global sigma pop_sig[<i>] is used in the following two situations: - a new parameter falls outside the range allowed by pop_xmin[<i>] and pop_xmax[<i>]. In this case the new parameter is chosen by adding a Gaussian random distributed value with sigma pop_sig[<i>] next to the respective boundary. - the difference between two parameters is zero. This will usually occur for integer parameters only. In this case the new parameter is chosen by adding a Gaussian random distributed value with sigma pop_sig[<i>] to the value of the effective donor.
If sigma is allowed to adapt during the fit, its value is set to (maximum parameter value - minimum parameter value) * <value> Thus, as the population converges to a smaller parameter spread, sigma dynamically becomes smaller as well.
The adaptation can be set for each parameter <parameter_number> separately.
The local sigma is used to modify a member not by differential evolution, but by a adding a local shift to the member. The local shift is a Gaussian distributed random value with sigma = <lsigma>. Whether DIFFEV uses this mode, is determined by the ==> variable diff_lo[1]. This gives the probability, that a given member will be modified by the local change instead of differential evolution.

A.7 backup

backup "NONE"
backup <**input**> **[,**<**extension**]**,** <**output**>

This command allows you to back up the current best solutions. Diffev expects that the current trial solutions are called inputfile.???? or inputfile.????.extension where "????" is a four digit integer number with leading zeros. Note that the '.' before the number "????" and before

the extension are mandatory parts of the filename. You need to ensure that your version of "kup.diffev.mac" adheres to this standard. If the output file name includes a path, make sure that the output directory does exist. DIFFEV does not create the output directory.
Examples backup CALC/calc, FINAL/final This form will back up the files "CALC/calc.????" as "FINAL/final.????" backup TEMP/calc, tth, FINAL/final This form will back up the files "TEMP/calc.????.tth" as "FINAL/final.????.tth"
If multiple files need to be backed up, use the 'backup' command repeatedly:
backup TEMP/calc, tth, FINAL/final backup TEMP/calc, grcalc, FINAL/final These two lines will back up the files "TEMP/calc.????.tth" as "FINAL/final.????.tth" "TEMP/calc.????.grcalc" as "FINAL/final.????.grcalc"
backup NONE Turns off the back up option
The backup option is useful, if the calculation of the solutions takes a long time and involves random configurations. In these cases, the extra time required to copy the files may outway the time to calculates these again after the end of the refinement. If the calculation involves random configurations, a repeated calculation of the solutions on which the cost function depends may not yield exactly the same result. With the backup option you ensure that you always have those backed up solutions correspond to the actual cost function values that DIFFEV used.
If the calculation of the solutions is quick or if it does not involve random configurations it is faster not to run the backup during the refinements.

A.8 branch

branch kuplot [, "-macro" <**macro_name**> [<**par1**> [, <**par2**> **...]]]**
branch discus [, "-macro" <**macro_name**> [<**par1**> [, <**par2**> **...]]]**

Active within the discus suite only!
Branches to the "kuplot" or "discus" section.
Within this section any standard KUPLOT command can be given. The behaviour of "kuplot" is essentially the same as in the stand alone version. Likewise for DISCUS.
The main use will branch to KUPLOT while the discus section is run via run_mpi from a DIFFEV slave.
Optionally the "-macro" qualifier instructs the suite to run the macro <macro_name> (with its optional parameters) before the interactive session is started.

A.9 compare

compare ["silent"]

This is the main part of the differential evolution section. This command reads the current results that the slave program stored in files ==> 'restrial' and compares these to the results of the parent generation. A new set of children is calculated according to the differential evolution algorithm. The successful parents are written to the file ==> 'logfile', which stores their respective cost function, and the full parameter set. At the same time the short summary file ==> 'summary' is appended with abbreviated information about the last generation. The new children parameters are written to the temporary files ==> 'trialfile'. The current generation is increased in file 'GENERATION'.

If the optional "silent" qualifier is specified, DIFFEV will not read the result files. Instead, DIFFEV must have received the current results by explicitly setting the values of child_val[*] for all children. This option will work only within the discus_suite, which is a collection of DIFFEV, DISCUS, and KUPLOT into a common program.

A.10 constraint

constraint <**logical expression**>

The parameter range may be restricted by defining one or several constraint conditions. Each condition must be a valid logical expression. For details of the syntax see the manual entry under ==> Command language. The parameters within the condition are referred to by p[<i>], where <i> is parameter number, 1 up to the dimension of the problem at hand. The dimension is fixed through parameter ==> 'pop_dimx', see the variable entry.
Example

```
constraint  p[1].gt.2   # The first parameter must be larger than 2
constraint  4.le. p[2]**2 + p[3]**2
                        # The sum of parameters 2 and 3 squared must be
                        #  equal or greater than 4.
```

If a constraint equation is not met, DIFFEV will create a new parameter set. This process is repeated until a valid parameter set is found, or until DIFFEV has tried so for MAX_CONSTR_TRIAL times. In this latter case the program stops.

A.11 deallocate

deallocate {**"all"** | **"constraint"** | **"population"**}

This command allows to deallocate memory for the specified program segments. This helps to conserve memory, if program sections are no longer needed.
This deallocation applies to memory that you allocated yourself and also to memory that DIFFEV has allocated automatically during runtime. As DIFFEV does not necessarily know, when you do not need the results of certain calculations any longer, it does not deallocate the automatically allocated memory sections unless you tell DIFFEV to do so.

```
"constraint"
```

Free memory associated to the maximum number of constraints.

```
"population"
```

Free memory associated to the population size and maximum number of refineable parameters.

A.12 dismiss

dismiss {<**n**> | **"all"** }

Set the R-value of the worst <n> parents to a very high value. Thus <n> of the next children will definetly have a better R-value and replace these parents.
This command should be used after you changed the parameter dimension and initialised ==> 'init' some or all of the trial values. Otherwise, many or eval all of the new children might not have a better R-value than their parents, and as a consequence the (re-)initilization of the trial values may get lost.

A.13 donor

donor {**"best"** | **"random"**}

The donor vector may be chosen in two different ways. "best" chooses the parent member that has the best parameter set. "random" chooses at random one of the parent vectors as donor vector. Two other parent vectors are always chosen at random to form the difference vector that is added to the donor vector. DIFFEV chooses the effective donor base along the straight line from the current parent vector to the donor. The point along this line is determined by the value of "diff_k[1]" (==> variables). For diff_k[1] = 0 the effective donor is the parent vector, for diff_k[1] = 1 the effective donor is the donor vector itself.

A.14 fix

fix <**parameter_no**>**,** {<**value**> | **"best"**}

This command fixes the value of parameter <parameter_no> to <value> for all members of the population. The ==> refine flag is turned of for this parameter. If the second parameter is "best", the parameter is set to the corresponding value of the current member with the best R-value.

A.15 initialize

initialize [<**par_number1**> **[,** <**par_number2**>**]] [, "silent"]**
initialise "logfile"

This command initializes the differential evolution sequence.
Before using this command, you must have defined:

```
Number of parameters to be defined ==> 'pop_dimx'
Size of the parent population      ==> 'pop_n'
Size of the children population    ==> 'pop_c'
Boundaries for each parameter      ==> 'pop_xmin'
                                   ==> 'pop_xmax'
Starting intervals for parameters  ==> 'pop_smin'
                                   ==> 'pop_smax'
Sigmas for parameter adjustment    ==> 'pop_sig'
Local sigmas                       ==> 'pop_lsig'
Cross over probability             ==> 'diff_cr'
Fraction of the difference vector  ==> 'diff_f'
Point between parent an donor base ==> 'diff_k'
Local search probability           ==> 'diff_lo'
```

Without the optional parameters, 'initialize' is used to start the generation zero.
'Initialize' will use this information to generate the zero's generation. The file 'GENERATION' is set to generation zero, the population size and the number of parameters is written. The files ==> 'logfile' and 'summary' are initialized. Old versions with the same name are overwritten! The starting parameter values are written to files ==> 'trialfile' After the header, each line contains a one parameter, 'pop_n' (i.e. the size of the population) lines are written.
If you want to reinitialize one or several parameters, the 'initialize' command may be used with the optional parameter(s). In this case 'initialize' will simply set the corresponding parameter, or parameter range from <par_number1> to <par_number2> to the range defined by the values of pop_smin[*] and pop_smax[*]. A new set of children and the GENERATION file is written.
If the last parameter is the string "silent", the trial files are note written to your disk. DIFFEV expects to be part of a suite program and will transfer the trial parameters directly to the slave program. See also ==> 'trialfile', 'run_mpi', 'compare' This option will work only within the discus_suite, which is a collection of DIFFEV, DISCUS, and KUPLOT into a common program.
The second form of the command can be used to initialize just the logfiles, see ==> 'logfile', 'summary'. This might be helpful if you want to reset the refinent cycle to a smaller generation number. For very length refinements with a few thousand refinement cycles and many parameters, a continuation will take appreciable time to read all previous cycles. If the actual development accross the cycles is not relevant to you you can reduces the file sizes drastically be a sequence like: pop_gen[1] = 1 initialise logfile

A.16 lastfile

lastfile <**filename**>

This defines the short log file of the parameter evolution.
After each generation, the short lastfile <filename> is overwritten by the parameters of the current generation.
It is a SPEC type file that contains all old R-values and parameters. Only this last generation makes up a scan. The first column is the member number, the second the R-value and all further columns the respective parameter values.

A.17 logfile

logfile <**filename**>

This defines the log file of the parameter evolution.
After each generation, the logfile <filename> is appended by the parameters of the current generation.
It is a SPEC type file that contains all old R-values and parameters. Each generation makes up a scan. The first column is the member number, the second the R-value and all further columns the respective parameter values.

A.18 refine

refine {**"all"**|**"none"**|<**number**> **[,**<**number**>**...]**}

This command allows you to set, which of the variables are refined. If you give the parameter number as negative number, the corresponding parameter is not refined.
Currently you can only specify up to 20 variable numbers on one "refine" command line. If you need to specify the behaviour for more than 20 variables, please use several "refine" commands.

A.19 summary

summary <**filename**>

This defines the log file of the R-value/cost function evolution.
The SPEC type summary files contains a single scan. Each generation creates one line within the scan. The first columns is the generation number. For the R-value and each of the parameters four columns are written. The first of these is the average value, the second the minium value, the third the maximum value and the fourth the sigma of the parameter distribution.

A.20 restrial

restrial <**result**>

These temporary files are used to communicate the R-value from the slave program back to DIFFEV. DIFFEV expects a separate file for each member of the population. The filename <result> is automatically augmented by a four digit member number. Thus, if the filename is Results, DIFFEV expects to find the files:

```
Results.0001
Results.0002
etc.
```

The file contains one line with two numbers, the member number and the R-value obtained for the corresponding set of parameters in file ==> 'trialfile'.
If the file name is the string "silent", the result files are note read from your disk. DIFFEV expects to be part of a suite program and will expect that the result values have been transfered directly from the slave program. See also ==> 'trialfile', 'run_mpi', 'compare'

A.21 run_mpi

run_mpi <**program_name**>**,** <**macro_name**>**,** <**no_repetitions**>**,** <**output_base**> **[,"socket"]**

This command starts the processing of the slave program <program_name>. If the program has been compiled with MPI and started with mpiexec, a parallel computation is started. Otherwise the slave program is executed in a serial loop over all children and individual repatitions. This command starts parallel processing of program <program_name>. The program will be executed <pop_c>*<no_repetitions> times. <pop_c> is the number of children for the current refinement and each child corresponds to one ==> trialfile. The calculations can be repeated

<no_repetions> times, with the identical parameter set. This may be necessary, if you need to average several calculations.
MPI option: The standard output of the program will be directed into a file <output_base>. The current child number (and the current repeat number, if present) will be appended as four digit wide field. On a UNIX system you can redirect the output to /dev/null.
NON MPI option: Regardless of the value of <output_base>, the output is displayed at the screen. The usual options for the ==> 'set prompt,' command do hold.
Starting with version 5.7 the communication between the diffev section and the discus or kuplot section is done via internal variables. These are: REF_GENERATION : the current generation number REF_MEMBER : the number of members in the population REF_CHILDREN : the number of children in the population REF_DIMENSION : the number of parameters defined in DIFFEV REF_KID : the current child REF_INDIV : the current repetition for the current child REF_KID REF_NINDIV : the intended total number of repetitions
The actual parameters are transferred to DISCUS/KUPLOT via variables ref_para[...] : Indices from 1 to REF_DIMENSION
The capitalization of the refinement variables is intended to distinguish these from user variables. other than that they should be used just as any user defined variable, with no square brackets: kid = REF_KID
IN a similar fashion, the KUPLOT rvalue command transfers its result back to DIFFEV internally. See 'resfile silent'
Essentially, DIFFEV branches to the section, executes the macro file and returns to DIFFEV. As the communication is done via the internal variables, the need for the macro parameters included in previous version 5.5 and earlier has ceased. You are encouraged not to rely on macro parameters but to use the refinement variables.
If the last optional parameter is "socket", the program will be started via a socket connection, else via a single "system" call.
system == non-socket option:
Relevant for older verions (5.6 an earleir only):
The actual line that starts the program has syntax: "program_name" -macro <macro.name> <cwd> <child> <indiv> > <output>
<macro_name> name of the macro file to be executed <cwd> 1st macro parameter string with current directory <child> current child number (1 to pop_c) <indiv> current repetion number (1 to <no_repetions>) This parameter is omitted, if <no_repetions> is zero <output> File for standard output written by <program_name>
Example run_mpi discus, discus.mac, 5, LOGFILES/d Runs discus with macro "discus.mac". For each child the calculation is repeated 5 times. DISCUS output is written into files "LOGFILES/d.xxxx.yyyy", where "xxxx" is the child number and "yyyy" the repetion number. run_mpi discus, discus.mac, 5, /dev/null Same, except that the output is written to "/dev/null" i.e. it is thrown away. run_mpi discus, discus.mac, 0, LOGFILES/d Same, except that no repetions are requested. Only one calculation is performed per child and the output is written to "LOGFILES/d.xxxx".
socket option:
The program is startet via socket at the beginning of a refinement. It remains active until diffev is terminated. A second, third program may be started as well, such as a discus run followed by kuplot.

The socket option starts the program via the line:
"program_name" -remote -access=127.0.0.0 -port=<port> > <output>
Thereafter the vatiables generation, member, children, parameters, kid, indiv, nindiv are defined and placed at their proper values. The corresponding lines are variable integer,generation generation = <value> And so on for the other variables. For each calculation i.e. combination of kid, indiv the current values are sent for "kid" and "indiv". All corresponding trial values are placed into the array "r[*]", into entries 201 to 201 + <parameters>. Then the calculation macro is started, which does of course not have to read the GENERATION and the trial file.

A.22 selection

selection {**"compare"** | **"best","all"**}

This command governs the selection criterium that determines which children and parents survive into the next generation.

```
"compare"    Each child is compared to its immediate parent. The better
             of these two will survive into the next generation. It will
             serve as one of the parents from which the next children
             are derived using the Differential Evolution Algorithm.
             The number of children ==> 'variables' pop_c, should be
             identical to the number of members i.e. parents
             ==> 'variables' pop_n.
"best","all" All parents and all children are put into a single list.
             Of this list the best <pop_n[1]> members survive into
             the next generation. No restriction applies to keep any
             parent or any children.
             You are free to use any number of children. If the number of
             children is increased compared to the number of parents, the
             selection becomes "tougher" since a smaller percentage of the
             whole population survives into the next generation. This will
             speed up convergence, yet run a higher risk of getting stuck
             in a local minimum.
```

A.23 trialfile

trialfile <**filename**>|**"silent"**

These temporary files are used to communicate the current set of parameters between DIFFEV and the slave program.
DIFFEV writes a separate file for each member of the population. The filename <filename> is automatically augmented by a four digit member number. The if the filename is Trials, DIFFEV expects to find the files:

```
Trials.0001
Trials.0002
etc.
```

The file has format:

```
# generation members children parameters
     181        45        90         4
# current member
```

```
    1
# parameter list
    0.8284027688E-02
    0.5815573883E+02
    0.2050311089E+01
    0.3000000000E+01
```

The first states the current generation, the number of members in the population, the number of children in each population, and the number of parameters. The next two lines states the number of the current member. This is followed by a list of all parameter values, each in a separate line.
If the file name is the string "silent", the trial files are note written to your disk. DIFFEV expects to be part of a suite program and will transfer the trial parameters directly to the slave program. See also ==> 'trialfile', 'run_mpi', 'compare' This option will work only within the discus_suite, which is a collection of DIFFEV, DISCUS, and KUPLOT into a common program.

A.24 type

type {**"integer"** | **"real"**},<**number**>
Defines the number that the parameter <number> assumes. Valid options are:

```
"integer" The parameter is restricted to integer numbers.
"real"    The parameter may take on any real,floating number.
```

See the entry on 'variables' regarding options to limit the allowed range, and the entry ==> 'constraint' on possible constraints.

A.25 variables

Like all programs of the Diffuse suite, diffev offers integer and real variable for standard calculations == Command Language/variables
Unique diffev variables are:

```
pop_gen[1]    The number of the current generation
pop_n[1]      The size of the parent population
              The size of the population may be changed during a
              refinement. In this case the GENERATION file is updated
              automatically. If the population is increased, the R-values
              for the new members are set to ten times the maximum
              current R-value. Parameter values are copied from old
              member no. one.
pop_c[1]      The size of the children population
              The size of the children population may be changed during a
              refinement. In this case, a new set of trial files is
              automatically generated and the GENERATION file is updated
              as well.
pop_dimx[1]   The dimension, i.e. the number of parameters
              The dimension may be increased during a refinement. To do
              so you need to increase pop_dimx[1], set all parameter
              related values for tis new parameter and run the
              ==> 'init <par_number>' command. This will initialize the
              parameter to the starting range and update the trial files
              and the generation file.
```

```
pop_xmin[<i>] Minimum allowed value for parameter <i>
pop_xmax[<i>] Maximum allowed value for parameter <i>

pop_smin[<i>] Minimum starting value for parameter <i>
pop_smax[<i>] Maximum starting value for parameter <i>

pop_sig[<i>]  Sigma of Gaussian distribution for parameter <i>. If the
              difference between two parent parameters is zero, or if a
              child parameter is outside the limits defined by pop_xmin
              and pop_xmax, the corresponding parameter is modified by
              a Gaussian distributed random number.
              Set pop_sig to zero to switch off this option.

pop_lsig[<i>] Sigma of local Gaussian distribution for parameter <i>.
              The probability diff_lo[1] determines if a given member
              is changed locally only or takes part in the usual
              differential evolution algorithm. If it is changed only
              locally, the parent parameters are modified by adding
              a Gaussian distributed random number, with mean zero
              and sigma pop_lsig.

pop_v[<i>,<j>] Value of parameter <i> for member <j>
              This parameter is read only.

pop_t[<i>,<j>] Current trial value of parameter <i> for child <j>

rvalue[<i>]   R-value for member <j>. This is the R-value for the
              current parent generation.

bestm[1]      Parent member that currently has the lowest R-value

bestr[1]      Currently lowest R-value.

worstm[1]     Parent member that currently has the highest R-value

worstr[1]     Currently highest R-value.

p[<i>]        Parameter symbol used in the constraint conditions.

diff_cr[1]    Cross over probability
diff_f[1]     Multiplier for difference vector
diff_k[1]     Multiplier for vector between parent vector and donor
diff_lo[1]    Probability for local refinement of a population member
diff_sel[1]   Selection mode for compare command, READ_ONLY
```

A.26 write

write {**"children"** | **"generation"**}
write "children"

The write command will generate a new set of child values and update the corresponding trial files and the GENERATION file. The generation number is not changed.

write "generation"

Writes the GENERATION file. No changes are done to the file.

Bibliography