From DEoptim to RcppDE: A case study in porting from C to C++ using Rcpp and RcppArmadillo

Dirk Eddelbuettel
Debian Project

Abstract

DEoptim (Mullen *et˜al.* 2009; Ardia *et˜al.* 2010a,b) provides differential evolution optimisation for R. It is based on an implementation by Storn (Price *et˜al.* 2006) and was originally implemented as an interpreted R script. It was then rewritten in ANSI C which resulted in a much improved performance.

The present paper introduces another implementation. This version is written in C++ based on the **Rcpp** package (Eddelbuettel and François 2010) which provides tools for a more direct integration of R objects at the C++ level—and vice versa. It also uses the **RcppArmadillo** package (François *et˜al.* 2010) which provides an interface from R to the **Armadillo** linear algebra package written in C++ by Sanderson (Sanderson 2010).

We find that by rewriting the differential evolution optimisation algorithm in C++, we achieve three usually exclusive goals: a) shorter code, b) easier maintainability as well as improved ability to enhance and extend, and c) consistent performance gains.

Keywords:˜**Rcpp**, **RcppArmadillo**, **DEoptim**, differential evolution, genetic algorithm.

1. Introduction

DEoptim (Mullen *et˜al.* 2009; Ardia *et˜al.* 2010a,b) provides differential evolution optimisation for the R language and statistical environement. Differential optimisation is one of several evolutionary computing approaches; genetic algorithns and simulated annealing are two other ones. Differential optimisation is reasonably close to genetic algorithms but differs in one key aspect: parameter values are encoded as floating point values (rather than sequences of binary digits) which makes it particular suitable for real-valued optimisation problems.

DEoptim is based on an implementation by Storn (Price *et˜al.* 2006). It was originally implemented as an (interpreted) R script before being rewritten in (compiled) C which resulted in a much improved performance. **DEoptim** has been used to optimise problems from a wide range of problem domains ranging from crystallography (Mullen *et˜al.* 2010) to agricultural economics (Börner *et˜al.* 2007) and computational finance (Boudt *et˜al.* 2008). It is also being used by two other CRAN packages for R: **micEconCES** (Henningsen and Henningsen 2010) and **selectMeta** (Rufibach 2010).

The present paper introduces the R package **RcppDE**. It provides another iteration as far as implementations of differential evolution go. This new version is based very closely on **DEoptim** but written in C++. The implementation employs the **Rcpp** package (Eddelbuettel

and François 2010) which provides tools for a more direct integration of R objects at the C++ level—and vice versa. It also uses the **RcppArmadillo** package (François *et˜al.* 2010) which provides an interface from R to the **Armadillo** linear algebra package written in C++ by Sanderson (Sanderson 2010).

The code structure descends directly from the current **DEoptim** by Ardia *et˜al.* (2010b). The conversion to C++ was undertaken to see whether one or more of the goals *shorter*, *easier* and *faster* could be achieved by switching the implementation language. These goals were loosely defined as follows:

shorter replacing code that is by necessity somewhat verbose when written in C with more compact code written in C++: an example would be copying of a matrix which is implemented as a dual loop copying each element—whereas C++ allows us to use a single (overloaded) + operator and hence a single statement;

easier this may appear as a corollary to the previous point but really covers other aspects such as the automatic type conversion offered by **Rcpp** as well as the automatic memory management: by replacing allocation and freeing of heap-based dynamic memory, a consistent source of programmer error would be elimnated—plus we are not trying 'short and incomprehensible' in the APL-sense but aim for possible improvements on *both* the length and the ease of comprehension without trading one off against the other;

faster this may be a bit more of a conjecture as ultimately, C++ and C can be expected to be roughly equivalent given matching compiler versions etc; however gains maybe be expected from replacing a copying operation of a block of adjacent memory cells with a single `memcpy()` call done behind the scenes; **RcppArmadillo** also offers further possible gains from template metaprogramming which can result in the elimination of temporary object in complex expression where, loosely speaking, compile-time effort is substituted to gain later run-time performance.

This paper is organised as follows. The next sections describes the structure of **DEoptim** which **RcppDE** shadows closesly. The following two section compare differences at the R and C++ level, respectively. Next, changes in auxiliarry files are discussed before we review changes in perfomance. A summary concludes. The appendix contains a list of figures contrasting the two implementations.

2. DEoptim structure

DEoptim is a straightforward and well-implemented package. Its core functionality is provided by three R files, as well as three C files.

In the transition **DEoptim** from to **RcppDE** many more changes were made to the C files: besides the obvious porting from C to C++, several internal code changes were made. We discuss these changes below. An important point to note is that the overall architecture and API remain as unchanged as possible. On the other hand, very few changes were required at the R level. The user-facing side of **DEoptim** persists virtually unchanged (with one or two changes discussed below).

Because of the dominant number of changes at the level of the compiled languages, we discuss the structure, and later on changes, of this part first before turning to the R side.

DEoptim		**RcppDE**	
File	Functions	File	Functions
`de4_0.c`	`DEoptimC()`	`deoptim.cpp`	`DEoptim()`
	`devol()`	`devol.cpp`	`devol()`
	`permute()`	`permute.cpp`	`permute()`
`evaluate.c`	`evaluate()`		
		`evaluate.h`	`EvalBase class`
`get_element.c`	`getListElement()`		

Table 1: Source file organisation for C files in **DEoptim**

2.1. / C++ structure and changes

Table˜1 lists the C and C++ files in **DEoptim** and **RcppDE**, respectively. The large file `de4_0.c` has been split into three files: one each for the core functions `DEoptim()` (which is called from R), `devol()` (which is the core differential evolution optimisation routine) and `permute()` (which is a helper function used to shuffle indices).

The evalution function has been replaced by a base class and two virtual classes. These can now make use of objective functions written in R (as in **DEoptim**) as well as ones written in C++. Using compiled objective functions can lead to substantial speed improvements, particularly when the evaluation of the objective is 'expensive' relative to overall computation in the optimization algorithm. Section˜3 discusses these changes in more detail.

2.2. R structure and changes

Table˜2 lists the files and corresponding key functions. Very few changes had to be made for **RcppDE**. Keeping the interface compatible between both implementations was an important goal. As can be seen from table˜2, no files or functions were added. A more detailed comparison follow below in section˜4.

File	Functions
`DEoptim.R`	`DEoptim()`
	`DEoptim.control()`
`methods.R`	`summary.DEoptim()`
	`plot.DEoptim()`
`zzz.R`	`.onLoad()`

Table 2: Source file organisation for R files in **DEoptim** and **RcppDE**

3. C / C++ changes

In this section, we will look at the changes at the C / C++ level. Figures˜4 to 6 contain the code the highest-level C++ function: `DEoptim()` (which we renamed from `DEoptim_C()` as there is no need for a different name at the C level relative to R). This is followed by figures˜7 to 14 on the main worker function `devol()` before figure˜15 compares the objective function evaluation of as the last element at the C / C++ level.

3.1. `de4_0.c` and `deoptim.cpp`

The `DEoptim()` function (renamed from `DEoptim_C()` as there is no need for a different name at the C level relative to R) is the entry point from R. It receives parameters, sets up the call of `devol()` and then prepares the return values.

Part 1: Start of `DEoptim()` The first part concerns itself with receiving parameters from R; figure˜4 displays this. The pure mechanics of passing and receiving parameters from R are easier thanks to logic provided by the **Rcpp** package:

1. Figure˜4 illustrates this point: Panel B (with code using C++) appears to be about half the size of panel A but this due in part to bringing comments on the same line as code. On the other hand, we save for example the declaration of ten `SEXP` variables as **Rcpp** objects can be converted directly to `SEXP` type.

2. Instead of using a mix of macros like `NUMERIC_VALUE`, `INTEGER_VALUE`, `NUMERIC_POINTER` and so on, we have a consistent use of the **Rcpp** template function `as` with template types corresponding to base typed `int`, `double` etc. Also of note is how one matrix object (`initialpom` for seeding a first population of parameter values) is initialized directly from a parameter.

3. Paremeter lookup is by a string value but done using the **Rcpp** lookup of elements in the `list` type (which corresponds to the R list passed in) rather than via a (functionally similar but ad-hoc) function `getListElement` that hence is not longer needed in **RcppDE**.

4. Here as in later code examples, care was taken to ensure that variable names and types correpond closely between both variants.

Part 2: Middle of `DEoptim()` The second part, displayed in figure˜5, allocates dynamic memory for both parameters returned to R as well as for temporary objects required to store the results of intermediate computations. Again, panel A shows the C code from **DEoptim** whereas panel B displays the C++ code from **RcppDE**. One difference becomes immediately apparent: the lack of proper matrix or vector types in C. We use the classes from the **Armadillo** C++ library written by Sanderson (2010) and provided via the R package **Armadillo** by François *et˜al.* (2010).

1. Matrix objects are created in C by first allocating a vector of pointers to pointers, which is followed by a loop in which each each column is allocated as vector of approrpriate length.

2. In C++, allocating a matrix is a single statement. Memory is managed by reference counting and is freed when objects go out of scope. This removes a *significant* portion of programmer errors.

3. Another subtle difference is in the allocations of the container holding different population snapshots, here called `d_storepop`: **Rcpp** lets us create a list object in which we store matrices, just as would in R whereas the C construct is much more complicated as we will see below.

4. A subtle point discussed more below is that **RcppDE** stores population members column-wise rather than row-wise. Whereas matrices on the left in panel A have dimension $n \times k$, we allocate them as $k \times n$ matrices in panel B.

Part 3: End of `DEoptim()` The third and last part of `DEoptim()` covers the actual call of the worker function `devol()` and the preparation of return values for R. As figure˜6 shows, this section realized a significant reduction in source code size.

1. The `devol()` function is called: as we aim to maintain interfaces, the call is unchanged between both approaches shown in figure˜6.

2. The code following the function call is very different. The new version is shorter for a number of reasons:

   (a) No need to create new temporary variables just to convert to `SEXP` types for return to R as the **Rcpp** package takes care of this: seamless conversion back to R is a key feature.

   (b) No need to allocate memory for new temporary variables (as we do not need these variables, and even if we did memory allocation would be implicit).

   (c) No need to `PROTECT` and later `UNPROTECT` such dynamic memory allocations (because this is handled automatically behind the scenes).

   (d) No need for an explicit new list object to hold the eight return variables.

   (e) No need to explicitly assign names for these eight return variables; this done implicitly while we create the returned list object.

3. Rather, a mere two statements are executed: the call to `devol()` followed by single call to create a return object as a list with named elements which are simply inserted—just like we would in R itself.

4. The remaining code takes care of exception handling by providing to `catch()` branches. These either forward a recognised exception to R, or (in the case of an unrecognised exception) signal a generic error.

In sum, we see how a number of (possibly small) enhancements taken together permit us to write a function which is considerably shorter and easier to read, yet fully equivalent in terms of its functionality.

3.2. `de4_0.c` and `devol.cpp`

The `devol()` function is the key part of the **DEoptim** implementation. It is also by far the largest function. We will discuss it again in different sections, each corresponding to one figure ranging from figure˜7 to figure˜14.

Part 1: Start of `devol()` The first part concerns the beginning of the `devol()`. The display (in figure˜7) of panels A and B differs mostly in minor ascpects:

1. The C version contains a declaration of a number of loop variable that are either not needed at all in the C++ version, or declared locally.
2. The urn depth is defined as a C macro and a constant variable, respectively.
3. The C++ version has an additional short block to set up the proper evaluation class for the user supplied function, depending on whether an external pointer object is passed (in which case we expect a compiled functin) or not in which case an R routine is used, just like in **DEoptim**.
4. The `sortIndex` vector is filled with index only in case strategy six has been selected as it is not used otherwise.

Part 2: Initializations in `devol()` The second part of `devol()` deals with the creation and initialization of a number of variables. The C language code in panel A is clearly more verbose and longer than the C++ code in panel B. As shown in figure˜8, key differences are:

1. Initialization of matrices to zero values uses two explicit loops in the C version.[1] In C++, we simply use the member function `zeros()` provided by the **Armadillo** library.
2. In panel B for the C++ case, the initial population in variable `initialpopm` is transposed in the C++ example. We keep each population as a *column* rather than a *row* as memory can generally be accessed faster column-wise.
3. The actual initialization of the first population is very comparable; in particular the R random number generator is called in the exact same sequence all throughout **RcppDE** so that results are in fact identical to those obtained from **DEoptim**.
4. The initial population evaluation occurs with a call to `evaluate()` in the original version, and a call of the member function of the evaluation class which will call either the supplied compiled function, or the supplied R functions.

Part 3: Iteration loop setup and start of population loop in `devol()` The next part of `devol()`, shown in figure˜9, starts both the main outer loop over all iterations as well as the main inner loop over all population elements. Similar to the discussion in the preceding paragraph, the new code is shorter in large part of more compact matrix expressions. Other differences are:

[1] The `memset()` function could be used in the C version to avoid the loops for a minor performance gain.

1. Intermediate populations are stored directly in a list, after being transposed to account for our design choice of operating column-wise. In the C code, the matrices are somewhat awkwardly 'serialised' into a single vector using the counter `popcnt` that incremened position by position.

2. Several other vector copies are each excecuted in a single statement rather than in an explicit loop.

3. At the beginning of the population loop, a vector is once more stored in a temporary variable and the permuation algoritm is called to pick suitable indices which will be used next.

Part 4 and 5: Population strategies in `devol()` Evaluating each population member based on the user-selected strategies is detailed in both figures˜10 and 11 covering the six available strategies as well as the default case. There are only fairly minor differences between both version as shown by panels A and B of both figures:

1. Instead of `if/else` branches, the new version uses a `switch` statement. This change can be beneficial as it may lead to fewer comparison, depending on the chosen strategy, and though the inner loop is executed many times, the overall benefit is still likely to be small.

2. The case-invariant initialization of `k` has been moved before the block.

3. The code for the different strategies differs very little between the initial C implementation and the newer C++ code.'4

Part 6: End of population loop in `devol()` Figure˜12 contains two fairly short segments that are entered once within each outer iteration after the loop over all population elements has finished. The two code segments in panels A and B of figure˜12 are fairly close, with the one difference once again the element-by-element copy of vector elements (in C) versus the single statement using C++ objects.

Part 7: Special case of `bs` flag in `devol()` Similarly, figure˜13 once more shows differences chiefly due to the way interim solutions are copied.

1. Panel A has a full nine loops for copying vector or matrix elements which are not needed in panel B.

2. Panel A has a somewhat elaborate segment to use a loop to copy a first population vector to a temporary vector, copy a second into the place of the first before then copying the content of the temporary vector into the second (and likewise for the evaluation score of these vectors). In Panel B, we simply use a single call of `swap()` member function for both the population vectors and their fitness.

We should note that this code is executed only when the user has changed the default value of false for the `bs` option in the control list for `DEoptim()`.

Part 8: End of `devol()` Finallt, figure˜14 contains the final portion of the `devol()` function. The population and its fitness value are saved. If the `checkWinner` option of the control structure has been changed by the user from the default value of false, a possible re-evaluation of the best population occurs and values are updated.

Next, if tracing is enabling and the iteration counter has a value which signals that tracing display should occur, then updates are printed before a few state variables are updated. The `devol()` then finishes right after restoring the state of the random number generator.

3.3. Evaluation functions in R and C++

Figure˜15 details the code used to evaluate the user-supplied objective function. This figure is an exception: the code from **RcppDE** is much longer than the code in **DEoptim**. This is due to a key main extension in **RcppDE**: the ability to use not only an R function to describe the objective function to be minimmized—but also a compiled function.

This is implemented by means of common C++ idiom: an abstract base class, here called `EvalBase`. This is an empty class which contains no code, but providing an interface containing of two public functions `eval()` and `getNbEvals()` which are *virtual*: the declare the interface, but provide no implementation. This is provided by two classes deriving from the abstract base class: one each for evaluating the R and the C++ function.

The class `EvalStandard` in panel B correspond most closely to the normal `evaluate()` in panel A. A function call with a set of parameters is prepared and the evaluated in an environment. Here, the function and the environment are supplied once at the beginning—and hence used to instantiate the class. Each evaluation then brings a new parameter vector.

The class `EvalCompiled` does the same, but not for the compiled function that we access via an external pointer. The support for external pointer types via type `XPtr` class in **Rcpp** was instrumental in implementing this. Similar to the standard case, the function is supplied at the beginning to instantiate the class. Later, on each evaluation call a new parameter vector is supplied.

4. R changes

Figures˜16 and 17 display the main R function `DEoptim()` which provides the interface the user of these packages employs. A few changes have been made:

1. **DEoptim** supports variable arguments in the R function, which follows the standard set by other optimisation functions. For symmetry with the compiled function, we support just a standard vector. However, the environment in which the function and parameters are evaluated can also be supplied by the user (whereas **DEoptim** always creates a new environment). The use of the environment then permits us to pass auxiliary arguments to the function in the same way the variable arguments would.

2. **RcppDE** therefore has an additional argument `env` for the user-supplied environement, as well as an additional creation of a default environment if none was supplied.

3. Population matrices are passed from C++ to R as matrix objects; no copy or rearrangement has to be undertaken. This saves a block of code at the top of panel B in figure˜17. Similarly, we do not have cast the population matrix as we already obtain a matrix.

None of the other functions from the files listed in table˜2 were changed (apart from a trivial startup message in the `.onLoad()` function in file `zzz.R`). In other words, the control options for `DEoptim()` are unchanged between between both versions, as are the additional method for summarizing, printing and plotting.

5. Auxiliary files

5.1. Regression tests

A a directory `tests/` has been added. It contains the file `compTest.R` which provides a first means of both *comparing* results between **RcppDE** and **DEoptim** and also timing them.

Three standard test functions (Wild, Rastrigin, Genrose) are run for four sets of parameter vector sizes—for both **RcppDE** and **DEoptim**. This ensures that results are identical between both implenations.

Adding full regression testing is left for a future version of **RcppDE**.

5.2. Demo files

Several demos have been added for **RcppDE** to the existing demo file already present in **DEoptim**. These new files are

- `SmallBenchmark` which runs the three standard test functions in both implementations for three small parameters sizes. As these small optimisation problems are relatively inexpensive, they are repeated a number of times and timings are obtained as trimmed means.
- `LargeBenchmark` which runs the three standard test functions in both implementations for three larger parameters sizes, this time without replication.
- `CompiledBenchmark` which runs the three standard test functions—but this time as compiled C++ functions demonstrating a significant performance gain relative to the R version.
- `environment` which runs a single small example showing how to pass an auxiliary parameter to the user-supplied function using an environment.

5.3. Benchmarking Scripts

The demos file from the preceding section are also being used for performance comparisons (as detailed in the next section).

The files are organised as thin wrapper scripts around the demo files described in the preceding section.

6. Performance

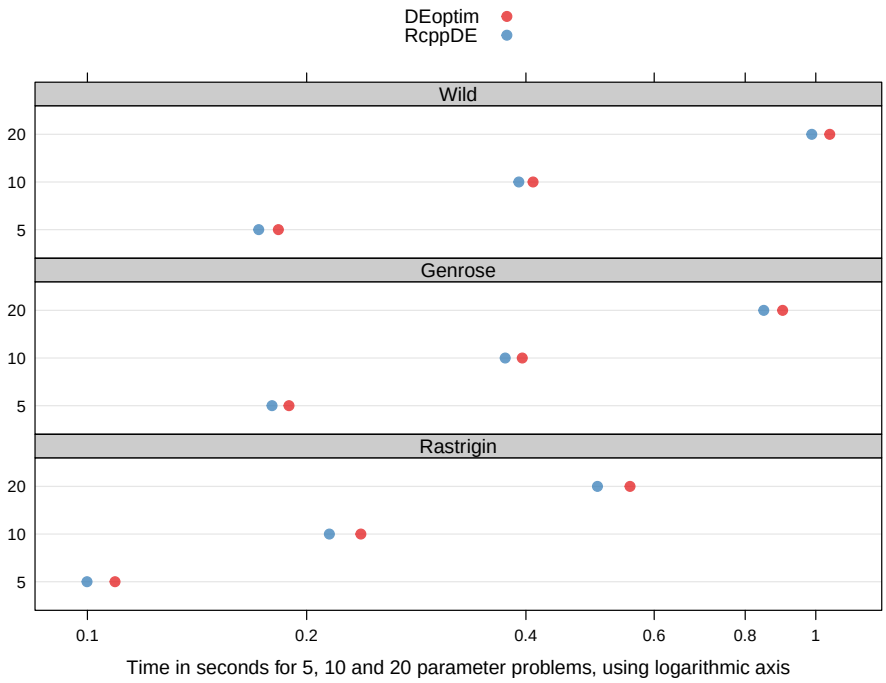

Figure 1: Performance comparison for small-scale optimisation problems.

Results from our calculations using scripts included in the **RcppDE** package; results are included in the source package. Tests were performed using Ubuntu Linux version 10.10 in 64-bit mode on an Intel i7 '920' CPU running at 2.6 GHz in hyperthreaded mode.

We will divide the performance comparison in three sections, corresponding to the same *small*, *large* and *compiled* split detailed above in section˜5.2.

Performance was measured between version 2.0-7 of **DEoptim** and the development versions of **RcppDE** preceding the 0.1.0 release of the latter.

6.1. Performance on small parameter vectors

Figure˜1 displays a performance comparison on the standard objective functions Wild, Genrose and Rastrigin. Each function is evaluated at five, ten and twenty parameters, respectively. As running time for the small problems is inconsequential, we report trimmed means (excluding 10% at each side) over a set of ten replications (as shown in the script and demo files in the package and discussed above).

From figure˜1, we can draw a number of conclusions:

- Performance between **DEoptim** and **RcppDE** is roughly comparable, though **RcppDE** has a small edge for which is consistent across functions and parameter sizes.
- Performance varies between objective functions: the Wild function with its two calls of trigonometric functions as well as five expressions of the vector x is roughly twice as expensive as the Rastrigin function which has just one trigonometric function and two

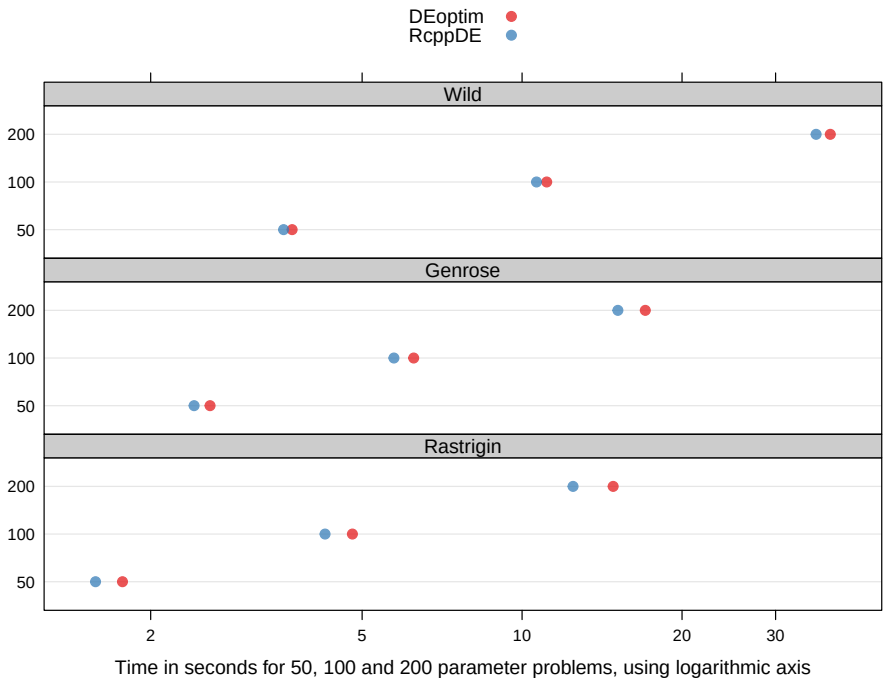

Figure 2: Performance comparison for large-scale optimisation problems.

Results from our calculations using scripts included in the **RcppDE** package; results are included in the source package. Tests were performed using Ubuntu Linux version 10.10 in 64-bit mode on an Intel i7 '920' CPU running at 2.6 GHz in hyperthreaded mode.

x terms.

- The cost of increasing parameter size is larger than just linear: for all functions, $n = 20$ takes more than twice as long than $n = 10$, and likewise for $n = 5$. Note that we plotted figures˜1 to 3 using a logarithmic x-axis which linearises the results.

6.2. Performance on large parameter vectors

Figure˜2 display results from the running the same three test functions for larger paramters vectors of size fifty, one hundred and two hundred, respetively.

As in the preceding figure˜1, using **RcppDE** rather than **DEoptim** on these optimization problems provides a consistent performance edge. This edge is now actually larger in both absolute and relative terms and ranges from just 3.5% (for the Wild function at $n = 50$) to almost 16% (for Rastrigin at $n = 200$). The performance gain also increases across all functions as n increases.

6.3. Performance with compiled objective function

Using a compiled objective function can yield dramatic performance gains. Figure˜3 compares results for **RcppDE** using a compiled objective function with **DEoptim** using the standard R

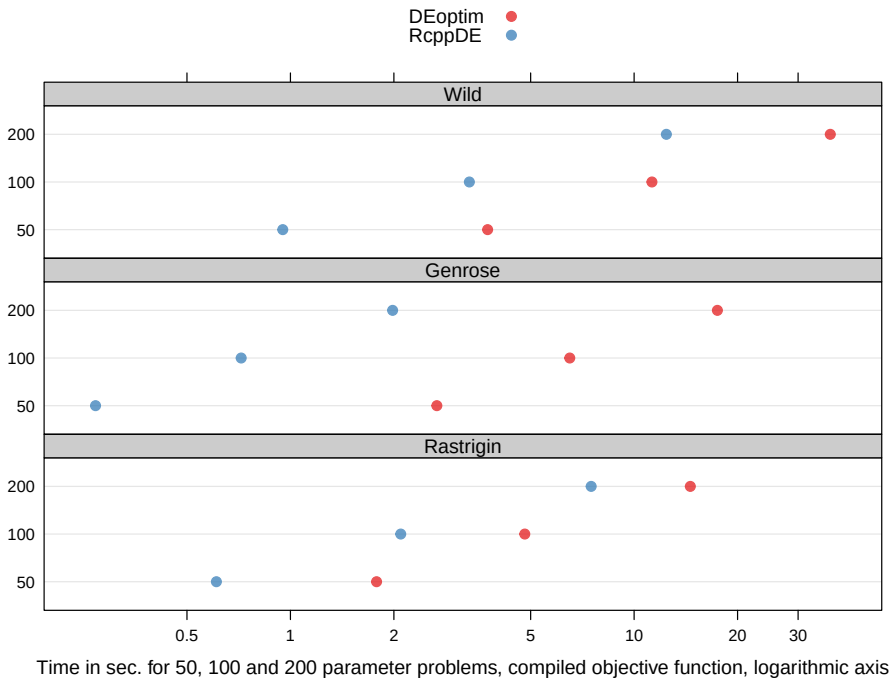

Figure 3: Performance comparison for compiled objective function in optimisation problems.

Results from our calculations using scripts included in the **RcppDE** package; results are included in the source package. Tests were performed using Ubuntu Linux version 10.10 in 64-bit mode on an Intel i7 '920' CPU running at 2.6 GHz in hyperthreaded mode.

implementations used before.

Gains can reach from (approximately) halving the observed time (for the Rastrigin function at $n = 200$) to reducing it to almost one-tenth (for the Genrose function at all sizes).

6.4. Discussion

This section has demonstrated performance gains for the **RcppDE** implementation of optimisation via differential evolution relative to the **DEoptim** implementation we parted from. The gains we observed were consistent and range from small gains on small problems to moderate gains in the ten-percent range for larger problems. In both these cases, the objective functions used were written in R.

This paper also introduces a performance gain with allows the analysts to deplay differential evolution optimisation within R, but via a compiled objective function. This approach can yield more dramatic gains as was seen in section˜6.3. Of course, the 'No Free Lunch' theorem still holds: writing such an objective function may well be more work, or may not always be feasible. However, if it is possible—and the **Rcpp** (Eddelbuettel and François 2010) for R and C++ integration makes it easier—then this approach could provide significant gains on a wide range of optimisation problems.

7. Summary

Differential evolution optimization has been available for R through the **DEoptim** package (Mullen *et˜al.* 2009; Ardia *et˜al.* 2010a,b). The **RcppDE** package presented in this paper started from a simple question. Could we start from **DEoptim** and, by relying on the **Rcpp** and **RcppArmadillo** packages, achieve what the the quip *Shorter, Faster, Easier: Pick Any Three* alludes to: simulataneous improvements in code length, expressiveness (while maintaining comprehensibility) and at the same time gain in performance?

Answering the first part is easiest. As section˜3 demonstrated, and as can be seen from figures˜4 to 14 in the appendix, the C++ source code in **RcppDE** is now measurably shorter that the C code in **DEoptim** that we built upon. While some of this change is caused by to editing style and comment preferences, a very significant portion is due to two key sources. First, the direct vector and matrix expressions in C++ free us from boilerplate code using loops just to copy vectors or matrices. Second, direct R object manipulation in C++ is possible thanks to the **Rcpp** package. Among other things, this makes it easier to access parameters passed from R, and to return results back from C++ to R.

Answering the second question in the affirmative is also possible. Section˜6 presented results of consistent performance gains of **Rcpp** over **DEoptim** across all test functions and all parameters vector sizes that were examined in this paper. Particularly noteworthy improvements in performancen were obtained with the compiled objective functions that are possible with **RcppDE**.

As for the third part and whether this makes using or extending the code *easier*: The proof may very well be in the pudding. We hope to now investigate how the use of multithreaded programming approaches, in particularly via the OpenMP framework, can further improve the performance of optimization via differential evolution. We think that having changed the code basis to the more compact C++ should facilitate this investigation. In the meantime, the relative ease with which the extension for compiled objective function has been added may be an indication of the possible benefits from using C++. So this is not yet fully proven, but some benefits have already been demostrated.

Concluding, we can score the approach presented here at a careful *2 1/2 out of 3 possible points*. Going from **DEoptim** to **RcppDE** has been a useful case study in applying **Rcpp** and **RcppArmadillo** to a well-established problem. We hope that **RcppDE** also proves useful to other R users.

References

Ardia D, Boudt K, Carl P, Mullen KM, Peterson BG (2010a). "Differential Evolution (DEoptim) for Non-Convex Portfolio Optimization." Unpublished Manuscript, URL http://ssrn.com/abstract=1584905.

Ardia D, Mullen K, Peterson B, Ulrich J (2010b). *DEoptim: Global optimization by differential evolution*. R package version 2.0-7, URL http://cran.r-project.org//package=DEoptim.

Börner J, Higgins SI, Kantelhardt J, Scheiter S (2007). "Rainfall or price variability: what

determines rangeland management decision? A simulation-optimization approach to South African savannas." *Agricultural Economics*, **37**(2-3), 189–200.

Boudt K, Peterson BG, Carl P (2008). "Hedge fund portfolio selection with modified expected shortfall." In M˜Costantino, M˜Larran, C˜Brebbia (eds.), "Computational Finance and its Applications III," volume˜41 of *WIT Transactions on Information and Communications Technologies*. WIT Press, Southampton, UK.

Eddelbuettel D, François R (2010). *Rcpp R/C++ interface package*. R package version 0.8.8, URL http://CRAN.R-project.org/package=Rcpp.

François R, Eddelbuettel D, Bates D (2010). *RcppArmadillo: Rcpp integration for Armadillo templated linear algebra library*. R package version 0.2.9, URL http://cran.r-project.org//package=RcppArmadillo.

Henningsen A, Henningsen G (2010). *micEconCES: Analysis with the Constant Elasticity of Scale (CES) function*. R package version 0.6-8, URL http://cran.r-project.org//package=micEconCES.

Mullen KM, Ardia D, Gil DL, Windover D, Cline J (2009). "DEoptim: An 'R' Package for Global Optimization by Differential Evolution." Unpublished Manuscript, URL http://ssrn.com/abstract=1526466.

Mullen KM, Krayzman V, Levin I (2010). "Atomic structure analysis at the nanoscale using the pair distribution function: simulation studies of simple elemental nanoparticles." *Journal of Applied Crystallography*, **43**(3), 483–490. URL http://dx.doi.org/10.1107/S0021889810008460.

Price KV, Storn RM, Lampinen JA (2006). *Differential Evolution – A Practical Approach to Global Optimization*. Springer, Berlin and Heidelberg. ISBN 3540209506.

Rufibach K (2010). *selectMeta: Estimation weight function in meta analysis*. R package version 1.0.1, URL http://cran.r-project.org//package=selectMeta.

Sanderson C (2010). "Armadillo: An open source C++ Algebra Library for Fast Prototyping and Computationally Intensive Experiments." *Technical report*, NICTA. URL http://arma.sf.net.

Appendix

Affiliation:

Dirk Eddelbuettel
Debian Project
River Forest, IL, USA
E-mail: edd@debian.org
URL: http://dirk.eddelbuettel.com

```c
SEXP DEoptimC(SEXP lower, SEXP upper, SEXP fn, SEXP control, SEXP rho)
{
  int i, j;

  /* External pointers to return to R */
  SEXP sexp_bestmem, sexp_bestval, sexp_nfeval, sexp_iter,
    out, out_names, sexp_pop, sexp_storepop, sexp_bestmemit, sexp_bestvalit;

  if (!isFunction(fn))
    error("fn is not a function!");
  if (!isEnvironment(rho))
    error("rho is not an environment!");

  /*-----Initialization of annealing parameters------------------------*/
  /* value to reach */
  double VTR = NUMERIC_VALUE(getListElement(control, "VTR"));
  /* chooses DE-strategy */
  int i_strategy = INTEGER_VALUE(getListElement(control, "strategy"));
  /* Maximum number of generations */
  int i_itermax = INTEGER_VALUE(getListElement(control, "itermax"));
  /* Number of objective function evaluations */
  long l_nfeval = (long)NUMERIC_VALUE(getListElement(control, "nfeval"));
  /* Dimension of parameter vector */
  int i_D = INTEGER_VALUE(getListElement(control, "npar"));
  /* Number of population members */
  int i_NP = INTEGER_VALUE(getListElement(control, "NP"));
  /* When to start storing populations */
  int i_storepopfrom = INTEGER_VALUE(getListElement(control, "storepopfrom"))-1;
  /* How often to store populations */
  int i_storepopfreq = INTEGER_VALUE(getListElement(control, "storepopfreq"));
  /* User-defined inital population */
  int i_specinitialpop = INTEGER_VALUE(getListElement(control, "specinitialpop"));
  double *initialpopv = NUMERIC_POINTER(getListElement(control, "initialpop"));
  /* User-defined bounds */
  double *f_lower = NUMERIC_POINTER(lower);
  double *f_upper = NUMERIC_POINTER(upper);
  /* stepsize */
  double f_weight = NUMERIC_VALUE(getListElement(control, "F"));
  /* crossover probability */
  double f_cross = NUMERIC_VALUE(getListElement(control, "CR"));
  /* Best of parent and child */
  int i_bs_flag = NUMERIC_VALUE(getListElement(control, "bs"));
  /* Print progress? */
  int i_trace = NUMERIC_VALUE(getListElement(control, "trace"));
  /* Re-evaluate best parameter vector? */
  int i_check_winner = NUMERIC_VALUE(getListElement(control, "checkWinner"));
  /* Average */
  int i_av_winner = NUMERIC_VALUE(getListElement(control, "avWinner"));
  /* p to define the top 100p% best solutions */
  double i_pPct = NUMERIC_VALUE(getListElement(control, "p"));
```

Panel A: C version

```c
RcppExport SEXP DEoptim(SEXP lowerS, SEXP upperS, SEXP fnS, SEXP controlS, SEXP rhoS) {

    try {
        Rcpp::NumericVector  f_lower(lowerS), f_upper(upperS);          // User-defined bounds
        Rcpp::List           control(controlS);                         // named list of params

        double VTR           = Rcpp::as<double>(control["VTR"]);        // value to reach
        int i_strategy       = Rcpp::as<int>(control["strategy"]);      // chooses DE-strategy
        int i_itermax        = Rcpp::as<int>(control["itermax"]);       // Maximum number of generations
        long l_nfeval        = 0;                                       // nb of function evals (NOT passed in)
        int i_D              = Rcpp::as<int>(control["npar"]);          // Dimension of parameter vector
        int i_NP             = Rcpp::as<int>(control["NP"]);            // Number of population members
        int i_storepopfrom   = Rcpp::as<int>(control["storepopfrom"]) - 1;  // When to start storing populations
        int i_storepopfreq   = Rcpp::as<int>(control["storepopfreq"]);  // How often to store populations
        int i_specinitialpop = Rcpp::as<int>(control["specinitialpop"]);// User-defined inital population
        Rcpp::NumericMatrix initialpopm = Rcpp::as<Rcpp::NumericMatrix>(control["initialpop"]);
        double f_weight      = Rcpp::as<double>(control["F"]);          // stepsize
        double f_cross       = Rcpp::as<double>(control["CR"]);         // crossover probability
        int i_bs_flag        = Rcpp::as<int>(control["bs"]);            // Best of parent and child
        int i_trace          = Rcpp::as<int>(control["trace"]);         // Print progress?
        int i_check_winner   = Rcpp::as<int>(control["checkWinner"]);   // Re-evaluate best parameter vector?
        int i_av_winner      = Rcpp::as<int>(control["avWinner"]);      // Average
        double i_pPct        = Rcpp::as<double>(control["p"]);          // p to define the top 100p% best solutions
```

Panel B: C++ version using **Rcpp**

Figure 4: Beginning of `DEoptim()` C/C++ function

```
/* Data structures for parameter vectors */
double **gta_popP = (double **)R_alloc(i_NP*2,sizeof(double *));
for (int i = 0; i < (i_NP*2); i++)
  gta_popP[i] = (double *)R_alloc(i_D,sizeof(double));

double **gta_oldP = (double **)R_alloc(i_NP,sizeof(double *));
for (int i = 0; i < i_NP; i++)
  gta_oldP[i] = (double *)R_alloc(i_D,sizeof(double));

double **gta_newP = (double **)R_alloc(i_NP,sizeof(double *));
for (int i = 0; i < i_NP; i++)
  gta_newP[i] = (double *)R_alloc(i_D,sizeof(double));

double *gt_bestP = (double *)R_alloc(1,sizeof(double) * i_D);

/* Data structures for objective function values associated with
 * parameter vectors */
double *gta_popC = (double *)R_alloc(i_NP*2,sizeof(double));
double *gta_oldC = (double *)R_alloc(i_NP,sizeof(double));
double *gta_newC = (double *)R_alloc(i_NP,sizeof(double));
double *gt_bestC = (double *)R_alloc(1,sizeof(double));

double *t_bestitP = (double *)R_alloc(1,sizeof(double) * i_D);
double *t_tmpP = (double *)R_alloc(1,sizeof(double) * i_D);
double *tempP = (double *)R_alloc(1,sizeof(double) * i_D);

int i_nstorepop = ceil((i_itermax - i_storepopfrom) / i_storepopfreq);
double *gd_pop = (double *)R_alloc(i_NP*i_D,sizeof(double));
double *gd_storepop = (double *)R_alloc(i_NP,sizeof(double) * i_D * i_nstorepop);
double *gd_bestmemit = (double *)R_alloc(i_itermax*i_D,sizeof(double));
double *gd_bestvalit = (double *)R_alloc(i_itermax,sizeof(double));
int gi_iter = 0;
```

Panel A: C version

```
arma::colvec minbound(f_lower.begin(), f_lower.size(), false);  // convert Rcpp vectors to arma vectors
arma::colvec maxbound(f_upper.begin(), f_upper.size(), false);
arma::mat initpopm(initialpopm.begin(), initialpopm.rows(), initialpopm.cols(), false);

arma::mat ta_popP(i_D, i_NP*2);                                 // Data structures for parameter vectors
arma::mat ta_oldP(i_D, i_NP);
arma::mat ta_newP(i_D, i_NP);
arma::colvec t_bestP(i_D);

arma::colvec ta_popC(i_NP*2);                                   // Data structures for obj. fun. values
arma::colvec ta_oldC(i_NP);
arma::colvec ta_newC(i_NP);
double t_bestC;

arma::colvec t_bestitP(i_D);
arma::colvec t_tmpP(i_D);

int i_nstorepop = ceil((i_itermax - i_storepopfrom) / i_storepopfreq);
arma::mat d_pop(i_D, i_NP);
Rcpp::List d_storepop(i_nstorepop);
arma::mat d_bestmemit(i_D, i_itermax);
arma::colvec d_bestvalit(i_itermax);
int i_iter = 0;
```

Panel B: C++ version using **Rcpp**

Figure 5: Memory allocation in `DEoptim()` C/C++ function

```
/*---optimization-------------------------------------*/
devol(VTR, f_weight, f_cross, i_bs_flag, f_lower, f_upper, fn, rho, i_trace,
      i_strategy, i_D, i_NP, i_itermax,
      initialpopv, i_storepopfrom, i_storepopfreq,
      i_specinitialpop, i_check_winner, i_av_winner,
      gta_popP, gta_oldP, gta_newP, gt_bestP,
      gta_popC, gta_oldC, gta_newC, gt_bestC,
      t_bestitP, t_tmpP, tempP,
      gd_pop, gd_storepop, gd_bestmemit, gd_bestvalit,
      &gi_iter, i_pPct, &l_nfeval);
/*---end optimization---------------------------------*/

PROTECT(sexp_bestmem = NEW_NUMERIC(i_D));
for (i = 0; i < i_D; i++) {
  NUMERIC_POINTER(sexp_bestmem)[i] = gt_bestP[i];
}

j = i_NP * i_D;
PROTECT(sexp_pop = NEW_NUMERIC(j));
for (i = 0; i < j; i++)
  NUMERIC_POINTER(sexp_pop)[i] = gd_pop[i];

j =  i_nstorepop * i_NP * i_D;
PROTECT(sexp_storepop = NEW_NUMERIC(j));
for (i = 0; i < j; i++)
  NUMERIC_POINTER(sexp_storepop)[i] = gd_storepop[i];

j = gi_iter * i_D;
PROTECT(sexp_bestmemit = NEW_NUMERIC(j));
for (i = 0; i < j; i++)
  NUMERIC_POINTER(sexp_bestmemit)[i] = gd_bestmemit[i];
j = gi_iter;
PROTECT(sexp_bestvalit = NEW_NUMERIC(j));
for (i = 0; i < j; i++)
  NUMERIC_POINTER(sexp_bestvalit)[i] = gd_bestvalit[i];

PROTECT(sexp_bestval = NEW_NUMERIC(1));
NUMERIC_POINTER(sexp_bestval)[0] = gt_bestC[0];

PROTECT(sexp_nfeval = NEW_INTEGER(1));
//INTEGER_POINTER(sexp_nfeval)[0] = 0;
INTEGER_POINTER(sexp_nfeval)[0] = l_nfeval;

PROTECT(sexp_iter = NEW_INTEGER(1));
INTEGER_POINTER(sexp_iter)[0] = gi_iter;

PROTECT(out = NEW_LIST(8));
SET_VECTOR_ELT(out, 0, sexp_bestmem);
SET_VECTOR_ELT(out, 1, sexp_bestval);
SET_VECTOR_ELT(out, 2, sexp_nfeval);
SET_VECTOR_ELT(out, 3, sexp_iter);
SET_VECTOR_ELT(out, 4, sexp_bestmemit);
SET_VECTOR_ELT(out, 5, sexp_bestvalit);
SET_VECTOR_ELT(out, 6, sexp_pop);
SET_VECTOR_ELT(out, 7, sexp_storepop);
```

Panel A: C version

```
  PROTECT(out_names = NEW_STRING(8));
  SET_STRING_ELT(out_names, 0, mkChar("bestmem"));
  SET_STRING_ELT(out_names, 1, mkChar("bestval"));
  SET_STRING_ELT(out_names, 2, mkChar("nfeval"));
  SET_STRING_ELT(out_names, 3, mkChar("iter"));
  SET_STRING_ELT(out_names, 4, mkChar("bestmemit"));
  SET_STRING_ELT(out_names, 5, mkChar("bestvalit"));
  SET_STRING_ELT(out_names, 6, mkChar("pop"));
  SET_STRING_ELT(out_names, 7, mkChar("storepop"));

  SET_NAMES(out, out_names);

  UNPROTECT(10);

  return out;
}
```

Panel A: C version (in both columns)

```
    // call actual Differential Evolution optimization given the parameters
    devol(VTR, f_weight, f_cross, i_bs_flag, minbound, maxbound, fnS, rhoS, i_trace, i_strategy, i_D, i_NP,
          i_itermax, initpopm, i_storepopfrom, i_storepopfreq, i_specinitialpop, i_check_winner, i_av_winner,
          ta_popP, ta_oldP, ta_newP, t_bestP, ta_popC, ta_oldC, ta_newC, t_bestC, t_bestitP, t_tmpP,
          d_pop, d_storepop, d_bestmemit, d_bestvalit, i_iter, i_pPct, l_nfeval);

    return Rcpp::List::create(Rcpp::Named("bestmem")   = t_bestP,       // and return a named list with results to R
                              Rcpp::Named("bestval")   = t_bestC,
                              Rcpp::Named("nfeval")    = l_nfeval,
                              Rcpp::Named("iter")      = i_iter,
                              Rcpp::Named("bestmemit") = trans(d_bestmemit),
                              Rcpp::Named("bestvalit") = d_bestvalit,
                              Rcpp::Named("pop")       = trans(d_pop),
                              Rcpp::Named("storepop")  = d_storepop);

  } catch( std::exception& ex) {
    forward_exception_to_r(ex);
  } catch(...) {
    ::Rf_error( "c++ exception (unknown reason)");
  }
  return R_NilValue;
}
```

Panel B: C++ version using **Rcpp**

Figure 6: `DEoptim()` call of `devol()` and return of results to R

```
void devol(double VTR, double f_weight, double f_cross, int i_bs_flag,
    double *lower, double *upper, SEXP fcall, SEXP rho, int trace,
    int i_strategy, int i_D, int i_NP, int i_itermax,
    double *initialpopv, int i_storepopfrom, int i_storepopfreq,
    int i_specinitialpop, int i_check_winner, int i_av_winner,
    double **gta_popP, double **gta_oldP, double **gta_newP, double *gt_bestP,
    double *gta_popC, double *gta_oldC, double *gta_newC, double *gt_bestC,
    double *t_bestitP, double *t_tmpP, double *tempP,
    double *gd_pop, double *gd_storepop, double *gd_bestmemit, double *gd_bestvalit,
    int *gi_iter, double i_pPct, long *l_nfeval)
{

#define URN_DEPTH  5   /* 4 + one index to avoid */

  /* initialize parameter vector to pass to evaluate function */
  SEXP par; PROTECT(par = NEW_NUMERIC(i_D));

  int i, j, k, x;  /* counting variables */
  int i_r1, i_r2, i_r3, i_r4;  /* placeholders for random indexes */

  int ia_urn2[URN_DEPTH];
  int i_nstorepop, i_xav;
  i_nstorepop = ceil((i_itermax - i_storepopfrom) / i_storepopfreq);

  int popcnt, bestacnt, same; /* lazy cnters */

  double *fa_minbound = lower;
  double *fa_maxbound = upper;
  double f_jitter, f_dither;

  double t_bestitC;
  double t_tmpC, tmp_best;

  double initialpop[i_NP][i_D];

  /* vars for DE/current-to-p-best/1 */
  int i_pbest;
  int p_NP = round(i_pPct * i_NP);  /* choose at least two best solutions */
      p_NP = p_NP < 2 ? 2 : p_NP;
  int sortIndex[i_NP];              /* sorted values of gta_oldC */
  for(i = 0; i < i_NP; i++) sortIndex[i] = i;

  /* vars for when i_bs_flag == 1 */
  int i_len, done, step, bound;
  double tempC;

  GetRNGstate();

  gta_popP[0][0] = 0;
```

Panel A: C version

```
void devol(double VTR, double f_weight, double f_cross, int i_bs_flag,
           arma::colvec & fa_minbound, arma::colvec & fa_maxbound, SEXP fcall, SEXP rho, int i_trace,
           int i_strategy, int i_D, int i_NP, int i_itermax, arma::mat & initialpopm,
           int i_storepopfrom, int i_storepopfreq, int i_specinitialpop, int i_check_winner, int i_av_winner,
           arma::mat &ta_popP, arma::mat &ta_oldP, arma::mat &ta_newP, arma::colvec & t_bestP,
           arma::colvec & ta_popC, arma::colvec & ta_oldC, arma::colvec & ta_newC, double & t_bestC,
           arma::colvec & t_bestitP, arma::colvec & t_tmpP,
           arma::mat &d_pop, Rcpp::List &d_storepop, arma::mat & d_bestmemit, arma::colvec & d_bestvalit,
           int & i_iterations, double i_pPct, long & l_nfeval) {

    Rcpp::DE::EvalBase *ev = NULL;                  // pointer to abstract base class
    if (TYPEOF(fcall) == EXTPTRSXP) {                  // non-standard mode: we are being passed an external pointer
        ev = new Rcpp::DE::EvalCompiled(fcall); // so assign a pointer using external pointer in fcall SEXP
    } else {                                            // standard mode: env_ is an env, fcall_ is a function
        ev = new Rcpp::DE::EvalStandard(fcall, rho);        // so assign R function and environment
    }
    const int urn_depth = 5;                            // 4 + one index to avoid
    Rcpp::NumericVector par(i_D);                  // initialize parameter vector to pass to evaluate function
    arma::icolvec::fixed<urn_depth> ia_urn2;         // fixed-size vector for urn draws
    arma::icolvec ia_urntmp(i_NP);                 // so that we don't need to re-allocated each time in permute
    arma::mat initialpop(i_D, i_NP);
    int i_nstorepop = ceil((i_itermax - i_storepopfrom) / i_storepopfreq);
    int p_NP = round(i_pPct * i_NP);                   // choose at least two best solutions
    p_NP = p_NP < 2 ? 2 : p_NP;
    arma::icolvec sortIndex(i_NP);                 // sorted values of ta_oldC
    if (i_strategy == 6) {
        for (int i = 0; i < i_NP; i++)
            sortIndex[i] = i;
    }
    GetRNGstate();
```

Panel B: C++ version using **Rcpp**

Figure 7: `devol()` beginning

```c
/* initialize initial popuplation */
for (int i = 0; i < i_NP; i++) {
  for (int j = 0; j < i_D; j++) {
    initialpop[i][j] = 0.0;
  }
}

/* initialize best members */
for (int i = 0; i < i_itermax * i_D; i++)
  gd_bestmemit[i] = 0.0;

/* initialize best values */
for (int i = 0; i < i_itermax; i++)
  gd_bestvalit[i] = 0.0;

/* initialize best population */
for (int i = 0; i < i_NP * i_D; i++)
  gd_pop[i] = 0.0;

/* initialize stored populations */
if (i_nstorepop < 0)
  i_nstorepop = 0;

for (int i = 0; i < (i_nstorepop * i_NP * i_D); i++)
  gd_storepop[i] = 0.0;

/* if initial population provided, initialize with values */
if (i_specinitialpop > 0) {
  k = 0;

  for (j = 0; j < i_D; j++) {
    for (i = 0; i < i_NP; i++) {
      initialpop[i][j] = initialpopv[k];
      k += 1;
    }
  }
}

/* number of function evaluations
 * (this is an input via DEoptim.control, but we over-write it?) */
*l_nfeval = 0;

/*------Initialization----------------------------*/
for (i = 0; i < i_NP; i++) {
  for (j = 0; j < i_D; j++) {
    if (i_specinitialpop <= 0) { /* random initial member */
      gta_popP[i][j] = fa_minbound[j] +
      unif_rand() * (fa_maxbound[j] - fa_minbound[j]);

    }
    else /* or user-specified initial member */
      gta_popP[i][j] = initialpop[i][j];
  }
  gta_popC[i] = evaluate(l_nfeval, gta_popP[i], par, fcall, rho);

  if (i == 0 || gta_popC[i] <= gt_bestC[0]) {
    gt_bestC[0] = gta_popC[i];
    for (j = 0; j < i_D; j++)
      gt_bestP[j]=gta_popP[i][j];
  }
}
```

```c
/*---assign pointers to current ("old") population---*/
gta_oldP = gta_popP;
gta_oldC = gta_popC;

/*------Iteration loop----------------------------------------*/
int i_iter = 0;
popcnt = 0;
bestacnt = 0;
i_xav = 1;
```

Panel A: C version (in both columns)

```c
initialpop.zeros();                     // initialize initial popuplation
d_bestmemit.zeros();                    // initialize best members
d_bestvalit.zeros();                    // initialize best values
d_pop.zeros();                          // initialize best population
i_nstorepop = (i_nstorepop < 0) ? 0 : i_nstorepop;

if (i_specinitialpop > 0) {             // if initial population provided, initialize with values
    initialpop = trans(initialpopm);    // transpose as we prefer columns for population members here
}

for (int i = 0; i < i_NP; i++) {        // ------Initialization----------------------------
    if (i_specinitialpop <= 0) {        // random initial member
        for (int j = 0; j < i_D; j++) {
            ta_popP.at(j,i) = fa_minbound[j] + ::unif_rand() * (fa_maxbound[j] - fa_minbound[j]);
        }
    } else {                            // or user-specified initial member
        ta_popP.col(i) = initialpop.col(i);
    }
    memcpy(REAL(par), ta_popP.colptr(i), Rf_nrows(par) * sizeof(double));
    ta_popC[i] = ev->eval(par);
    if (i == 0 || ta_popC[i] <= t_bestC) {
        t_bestC = ta_popC[i];
        t_bestP = ta_popP.unsafe_col(i);
    }
}

ta_oldP = ta_popP.cols(0, i_NP-1);      // ---assign pointers to current ("old") population---
ta_oldC = ta_popC.rows(0, i_NP-1);

int i_iter = 0;                         // ------Iteration loop----------------------------------------
int popcnt = 0;
int i_xav = 1;
```

Panel B: C++ version using **Rcpp**

Figure 8: `devol()` initializations

```c
/* loop */
while ((i_iter < i_itermax) && (gt_bestC[0] > VTR))
  {
    /* store intermediate populations */
    if (i_iter % i_storepopfreq == 0 && i_iter >= i_storepopfrom) {
      for (i = 0; i < i_NP; i++) {
        for (j = 0; j < i_D; j++) {
          gd_storepop[popcnt] = gta_oldP[i][j];
          popcnt++;
        }
      }
    } /* end store pop */

    /* store the best member */
    for(j = 0; j < i_D; j++) {
      gd_bestmemit[bestacnt] = gt_bestP[j];
      bestacnt++;
    }
    /* store the best value */
    gd_bestvalit[i_iter] = gt_bestC[0];

    for (j = 0; j < i_D; j++)
      t_bestitP[j] = gt_bestP[j];
    t_bestitC = gt_bestC[0];

    i_iter++;

    /*----computer dithering factor -----------------*/
    f_dither = f_weight + unif_rand() * (1.0 - f_weight);

    /*---DE/current-to-p-best/1 ---------------------------------*/
    if (i_strategy == 6) {
      /* create a copy of gta_oldC to avoid changing it */
      double temp_oldC[i_NP];
      for(j = 0; j < i_NP; j++) temp_oldC[j] = gta_oldC[j];

      /* sort temp_oldC to use sortIndex later */
      rsort_with_index( (double*)temp_oldC, (int*)sortIndex, i_NP );
    }

    /*----start of loop through ensemble-------------------------*/
    for (i = 0; i < i_NP; i++) {

      /*t_tmpP is the vector to mutate and eventually select*/
      for (j = 0; j < i_D; j++)
        t_tmpP[j] = gta_oldP[i][j];
      t_tmpC = gta_oldC[i];

      permute(ia_urn2, URN_DEPTH, i_NP, i); /* Pick 4 random and distinct */

      i_r1 = ia_urn2[1];  /* population members */
      i_r2 = ia_urn2[2];
      i_r3 = ia_urn2[3];
      i_r4 = ia_urn2[4];
```

Panel A: C version

```c
while ((i_iter < i_itermax) && (t_bestC > VTR)) {    // main loop ==================================
    if (i_iter % i_storepopfreq == 0 && i_iter >= i_storepopfrom) {         // store intermediate populations
        d_storepop[popcnt++] = Rcpp::wrap( trans(ta_oldP) );
    } // end store pop

    d_bestmemit.col(i_iter) = t_bestP;      // store the best member
    d_bestvalit[i_iter] = t_bestC;          // store the best value
    t_bestitP = t_bestP;
    i_iter++;                               // increase iteration counter

    double f_dither = f_weight + ::unif_rand() * (1.0 - f_weight);  // ----computer dithering factor --------------

    if (i_strategy == 6) {                  // ---DE/current-to-p-best/1 ----------------------------------------
        arma::colvec temp_oldC = ta_oldC;                           // create copy of ta_oldC to avoid changing it
        rsort_with_index( temp_oldC.memptr(), sortIndex.begin(), i_NP );    // sort temp_oldC to use sortIndex
    }

    for (int i = 0; i < i_NP; i++) {        // ----start of loop through ensemble-----------------------

        t_tmpP = ta_oldP.col(i);            // t_tmpP is the vector to mutate and eventually select

        permute(ia_urn2.memptr(), urn_depth, i_NP, i, ia_urntmp.memptr()); // Pick 4 random and distinct
        int k = 0;                          // loop counter used in all strategies below
```

Panel B: C++ version using **Rcpp**

Figure 9: `devol()` iteration loop setup and beginning of population loop

```
/*===Choice of strategy=======================================================*/
/*---classical strategy DE/rand/1/bin---------------------------------------*/
if (i_strategy == 1) {

  j = (int)(unif_rand() * i_D); /* random parameter */
  k = 0;
  do {
    /* add fluctuation to random target */
    t_tmpP[j] = gta_oldP[i_r1][j] +
      f_weight * (gta_oldP[i_r2][j] - gta_oldP[i_r3][j]);

    j = (j + 1) % i_D;
    k++;
  }while((unif_rand() < f_cross) && (k < i_D));

}
/*---DE/local-to-best/1/bin-------------------------------------------------*/
else if (i_strategy == 2) {

  j = (int)(unif_rand() * i_D); /* random parameter */
  k = 0;
  do {
    /* add fluctuation to random target */

    t_tmpP[j] = t_tmpP[j] +
      f_weight * (t_bestitP[j] - t_tmpP[j]) +
      f_weight * (gta_oldP[i_r2][j] - gta_oldP[i_r3][j]);
    j = (j + 1) % i_D;
    k++;
  }while((unif_rand() < f_cross) && (k < i_D));

}
/*---DE/best/1/bin with jitter----------------------------------------------*/
else if (i_strategy == 3) {

  j = (int)(unif_rand() * i_D); /* random parameter */
  k = 0;
  do {
    /* add fluctuation to random target */
    f_jitter = 0.0001 * unif_rand() + f_weight;
    t_tmpP[j] = t_bestitP[j] +
      f_jitter * (gta_oldP[i_r1][j] - gta_oldP[i_r2][j]);

    j = (j + 1) % i_D;
    k++;
  }while((unif_rand() < f_cross) && (k < i_D));

}
/*---DE/rand/1/bin with per-vector-dither-----------------------------------*/
else if (i_strategy == 4) {

  j = (int)(unif_rand() * i_D); /* random parameter */
  k = 0;
  do {
    /* add fluctuation to random target */
    t_tmpP[j] = gta_oldP[i_r1][j] +
      (f_weight + unif_rand()*(1.0 - f_weight))*
      (gta_oldP[i_r2][j]-gta_oldP[i_r3][j]);

    j = (j + 1) % i_D;
    k++;
  }while((unif_rand() < f_cross) && (k < i_D));

}
```

Panel A: C version

```
// ===Choice of strategy=======================================================
switch (i_strategy) {

case 1: {                               // ---classical strategy DE/rand/1/bin---------------------------------
    int j = static_cast<int>(::unif_rand() * i_D);        // random parameter
    do {                                // add fluctuation to random target
        t_tmpP[j] = ta_oldP.at(j,ia_urn2[1]) + f_weight *
            (ta_oldP.at(j,ia_urn2[2]) - ta_oldP.at(j,ia_urn2[3]));
        j = (j + 1) % i_D;
    } while ((::unif_rand() < f_cross) && (++k < i_D));
    break;
}
case 2: {                               // ---DE/local-to-best/1/bin-------------------------------------------
    int j = static_cast<int>(::unif_rand() * i_D);        // random parameter
    do {                                // add fluctuation to random target
        t_tmpP[j] = t_tmpP[j] + f_weight * (t_bestitP[j] - t_tmpP[j]) + f_weight *
            (ta_oldP.at(j,ia_urn2[2]) - ta_oldP.at(j,ia_urn2[3]));
        j = (j + 1) % i_D;
    } while ((::unif_rand() < f_cross) && (++k < i_D));
    break;
}
case 3: {                               // ---DE/best/1/bin with jitter----------------------------------------
    int j = static_cast<int>(::unif_rand() * i_D);        // random parameter
    do {                                // add fluctuation to random target
        double f_jitter = 0.0001 * ::unif_rand() + f_weight;
        t_tmpP[j] = t_bestitP[j] + f_jitter * (ta_oldP.at(j,ia_urn2[1]) - ta_oldP.at(j,ia_urn2[2]));
        j = (j + 1) % i_D;
    } while ((::unif_rand() < f_cross) && (++k < i_D));
    break;
}
case 4: {                               // ---DE/rand/1/bin with per-vector-dither-----------------------------
    int j = static_cast<int>(::unif_rand() * i_D);        // random parameter
    do {                                // add fluctuation to random target *
        t_tmpP[j] = ta_oldP.at(j,ia_urn2[1]) + (f_weight + ::unif_rand()*(1.0 - f_weight))
            * (ta_oldP.at(j,ia_urn2[2]) - ta_oldP.at(j,ia_urn2[3]));
        j = (j + 1) % i_D;
    } while ((::unif_rand() < f_cross) && (++k < i_D));
    break;
}
```

Panel B: C++ version using **Rcpp**

Figure 10: `devol()` first four strategy options

```
/*---DE/rand/1/bin with per-generation-dither--------------------------------*/
else if (i_strategy == 5) {

  j = (int)(unif_rand() * i_D); /* random parameter */
  k = 0;
  do {
    /* add fluctuation to random target */
    t_tmpP[j] = gta_oldP[i_r1][j] +
      f_dither * (gta_oldP[i_r2][j] - gta_oldP[i_r3][j]);

    j = (j + 1) % i_D;
    k++;
  }while((unif_rand() < f_cross) && (k < i_D));

}
/*---DE/current-to-p-best/1 (JADE)-------------------------------------------*/
else if (i_strategy == 6) {

  /* select from [0, 1, 2, ..., (pNP-1)] */
  i_pbest = sortIndex[(int)(unif_rand() * p_NP)];

  j = (int)(unif_rand() * i_D); /* random parameter */
  k = 0;
  do {
    /* add fluctuation to random target */
    t_tmpP[j] = gta_oldP[i][j] +
      f_weight * (gta_oldP[i_pbest][j] - gta_oldP[i][j]) +
      f_weight * (gta_oldP[i_r1][j]    - gta_oldP[i_r2][j]);

    j = (j + 1) % i_D;
    k++;
  }while((unif_rand() < f_cross) && (k < i_D));

}
/*---variation to DE/rand/1/bin: either-or-algorithm-------------------------*/
else {

  j = (int)(unif_rand() * i_D); /* random parameter */
  k = 0;
  if (unif_rand() < 0.5) { /* differential mutation, Pmu = 0.5 */
    do {
      /* add fluctuation to random target */
      t_tmpP[j] = gta_oldP[i_r1][j] +
        f_weight * (gta_oldP[i_r2][j] - gta_oldP[i_r3][j]);

      j = (j + 1) % i_D;
      k++;
    }while((unif_rand() < f_cross) && (k < i_D));
  }
  else {
    /* recombination with K = 0.5*(F+1) -. F-K-Rule */
    do {
      /* add fluctuation to random target */
      t_tmpP[j] = gta_oldP[i_r1][j] +
        0.5 * (f_weight + 1.0) * (gta_oldP[i_r2][j]
                                  + gta_oldP[i_r3][j] - 2 * gta_oldP[i_r1][j]);

      j = (j + 1) % i_D;
      k++;
    }while((unif_rand() < f_cross) && (k < i_D));

  }
}/* end if (i_strategy ...*/
```

Panel A: C version

```
casee 5: {                              // ---DE/rand/1/bin with per-generation-dither---------------
    int j = static_cast<int>(::unif_rand() * i_D);          // random parameter
    do {                                // add fluctuation to random target
        t_tmpP[j] = ta_oldP.at(j,ia_urn2[1]) + f_dither
            * (ta_oldP.at(j,ia_urn2[2]) - ta_oldP.at(j,ia_urn2[3]));
        j = (j + 1) % i_D;
    } while ((::unif_rand() < f_cross) && (++k < i_D));
    break;
}
case 6: {                               // ---DE/current-to-p-best/1 (JADE)--------------------------
    // select from [0, 1, 2, ..., (pNP-1)]
    int i_pbest = sortIndex[static_cast<int>(::unif_rand() * p_NP)];
    int j = static_cast<int>(::unif_rand() * i_D);          // random parameter
    do {                                // add fluctuation to random target
        t_tmpP[j] = ta_oldP.at(j,i) + f_weight * (ta_oldP.at(j,i_pbest) - ta_oldP.at(j,i)) +
            f_weight * (ta_oldP.at(j,ia_urn2[1]) - ta_oldP.at(j,ia_urn2[2]));
        j = (j + 1) % i_D;
    } while ((::unif_rand() < f_cross) && (++k < i_D));
    break;
}
default: {                              // ---variation to DE/rand/1/bin: either-or-algorithm--------
    int j = static_cast<int>(::unif_rand() * i_D);          // random parameter
    if (::unif_rand() < 0.5) {          // differential mutation, Pmu = 0.5
        do {                            // add fluctuation to random target */
            t_tmpP[j] = ta_oldP.at(j,ia_urn2[1]) + f_weight *
                            (ta_oldP.at(j,ia_urn2[2]) - ta_oldP.at(j,ia_urn2[3]));
            j = (j + 1) % i_D;
        } while ((::unif_rand() < f_cross) && (++k < i_D));

    } else {                            // recombination with K = 0.5*(F+1) -. F-K-Rule
        do {                            // add fluctuation to random target */
            t_tmpP[j] = ta_oldP.at(j,ia_urn2[1]) + 0.5 * (f_weight + 1.0) *
                (ta_oldP.at(j,ia_urn2[2]) + ta_oldP.at(j,ia_urn2[3]) - 2 * ta_oldP.at(j,ia_urn2[1]));
            j = (j + 1) % i_D;
        } while ((::unif_rand() < f_cross) && (++k < i_D));
    }
    break;
}
} // end switch (i_strategy) ...
```

Panel B: C++ version using **Rcpp**

Figure 11: `devol()` remaining three strategy options

```
  /*----boundary constraints, bounce-back method was not enforcing bounds correctly*/
  for (j = 0; j < i_D; j++) {
    if (t_tmpP[j] < fa_minbound[j]) {
      t_tmpP[j] = fa_minbound[j] +
        unif_rand() * (fa_maxbound[j] - fa_minbound[j]);
    }
    if (t_tmpP[j] > fa_maxbound[j]) {
      t_tmpP[j] =  fa_maxbound[j] -
        unif_rand() * (fa_maxbound[j] - fa_minbound[j]);
    }
  }

  /*------Trial mutation now in t_tmpP-----------------*/
  /* Evaluate mutant in t_tmpP[]*/

  t_tmpC = evaluate(l_nfeval, t_tmpP, par, fcall, rho);

  /* note that i_bs_flag means that we will choose the
   *best NP vectors from the old and new population later*/
  if (t_tmpC <= gta_oldC[i] || i_bs_flag) {
    /* replace target with mutant */
    for (j = 0; j < i_D; j++)
      gta_newP[i][j]=t_tmpP[j];
    gta_newC[i]=t_tmpC;
    if (t_tmpC <= gt_bestC[0]) {
      for (j = 0; j < i_D; j++)
        gt_bestP[j]=t_tmpP[j];
      gt_bestC[0]=t_tmpC;
    }
  }
  else {
    for (j = 0; j < i_D; j++)
      gta_newP[i][j]=gta_oldP[i][j];
    gta_newC[i]=gta_oldC[i];

  }
} /* End mutation loop through pop. */
```

Panel A: C version

```
    for (int j = 0; j < i_D; j++) {     // boundary constr., bounce-back meth. not enforcing bounds
        if (t_tmpP[j] < fa_minbound[j]) {
            t_tmpP[j] = fa_minbound[j] + ::unif_rand() * (fa_maxbound[j] - fa_minbound[j]);
        }
        if (t_tmpP[j] > fa_maxbound[j]) {
            t_tmpP[j] = fa_maxbound[j] - ::unif_rand() * (fa_maxbound[j] - fa_minbound[j]);
        }
    }

    // ------Trial mutation now in t_tmpP-----------------
    memcpy(REAL(par), t_tmpP.memptr(), Rf_nrows(par) * sizeof(double));
    double t_tmpC = ev->eval(par);              // Evaluate mutant in t_tmpP
    if (t_tmpC <= ta_oldC[i] || i_bs_flag) {    // i_bs_flag means will choose best NP later
        ta_newP.col(i) = t_tmpP;                // replace target with mutant
        ta_newC[i] = t_tmpC;
        if (t_tmpC <= t_bestC) {
            t_bestP = t_tmpP;
            t_bestC = t_tmpC;
        }
    } else {
        ta_newP.col(i) = ta_oldP.col(i);
        ta_newC[i] = ta_oldC[i];
    }
} // End mutation loop through pop., ie the "for (i = 0; i < i_NP; i++)"
```

Panel B: C++ version using **Rcpp**

Figure 12: `devol()` remainder of population mutation loop

```
if(i_bs_flag) {
  /* examine old and new pop. and take the best NP members
   * into next generation */
  for (i = 0; i < i_NP; i++) {
    for (j = 0; j < i_D; j++)
      gta_popP[i][j] = gta_oldP[i][j];
    gta_popC[i] = gta_oldC[i];
  }
  for (i = 0; i < i_NP; i++) {
    for (j = 0; j < i_D; j++)
      gta_popP[i_NP+i][j] = gta_newP[i][j];
    gta_popC[i_NP+i] = gta_newC[i];
  }
  i_len = 2 * i_NP;
  step = i_len;  /* array length */
  while (step > 1) {
    step /= 2;   /* halve the step size */
    do {
      done = 1;
      bound  = i_len - step;
      for (j = 0; j < bound; j++) {
          i = j + step + 1;
          if (gta_popC[j] > gta_popC[i-1]) {
              for (k = 0; k < i_D; k++)
                tempP[k] = gta_popP[i-1][k];
              tempC = gta_popC[i-1];
              for (k = 0; k < i_D; k++)
                gta_popP[i-1][k] = gta_popP[j][k];
              gta_popC[i-1] = gta_popC[j];
              for (k = 0; k < i_D; k++)
                gta_popP[j][k] = tempP[k];
              gta_popC[j] = tempC;
                done = 0;
                /* if a swap has been made we are not finished yet */
          }  /* if */
      }  /* for */
    } while (!done);   /* while */
  } /*while (step > 1) */
  /* now the best NP are in first NP places in gta_pop, use them */
  for (i = 0; i < i_NP; i++) {
    for (j = 0; j < i_D; j++)
      gta_newP[i][j] = gta_popP[i][j];
    gta_newC[i] = gta_popC[i];
  }
} /*i_bs_flag*/
```

Panel A: C version

```
if (i_bs_flag) {       // examine old and new pop. and take the best NP members into next generation

    ta_popP.cols(0, i_NP-1) = ta_oldP;
    ta_popC.rows(0, i_NP-1) = ta_oldC;

    ta_popP.cols(i_NP, 2*i_NP-1) = ta_newP;
    ta_popC.rows(i_NP, 2*i_NP-1) = ta_newC;

    int i_len = 2 * i_NP;
    int step = i_len, done;        // array length
    while (step > 1) {
        step /= 2;                   // halve the step size
        do {
            done = 1;
            int bound  = i_len - step;
            for (int j = 0; j < bound; j++) {
                int i = j + step + 1;
                if (ta_popC[j] > ta_popC[i-1]) {
                    ta_popP.swap_cols(j, i-1);
                    ta_popC.swap_rows(j, i-1);
                    done = 0;
                }  // if
            }  // for
        } while (!done); // while
    } // while (step > 1)
    ta_newP = ta_popP.cols(0, i_NP-1);        // now the best NP are in first NP places in gta_pop, use them
    ta_newC = ta_popC.rows(0, i_NP-1);
} // i_bs_flag
```

Panel B: C++ version using **Rcpp**

Figure 13: `devol()` case of `i_bs_flag`

```
    /* have selected NP mutants move on to next generation */
    for (i = 0; i < i_NP; i++) {
      for (j = 0; j < i_D; j++)
        gta_oldP[i][j] = gta_newP[i][j];
      gta_oldC[i] = gta_newC[i];
    }
    /* check if the best stayed the same, if necessary */
    if(i_check_winner)  {
      same = 1;
      for (j = 0; j < i_D; j++)
        if(t_bestitP[j] != gt_bestP[j]) {
          same = 0;
        }
      if(same && i_iter > 1)  {
        i_xav++;
        /* if re-evaluation of winner */
        tmp_best = evaluate(l_nfeval, gt_bestP, par, fcall, rho);

        /* possibly letting the winner be the average of all past generations */
        if(i_av_winner)
          gt_bestC[0] = ((1/(double)i_xav) * gt_bestC[0])
            + ((1/(double)i_xav) * tmp_best)
            + (gd_bestvalit[i_iter-1] * ((double)(i_xav - 2))/(double)i_xav);
        else
          gt_bestC[0] = tmp_best;

      }
      else {
        i_xav = 1;
      }

    }
    for (j = 0; j < i_D; j++)
      t_bestitP[j] = gt_bestP[j];
    t_bestitC = gt_bestC[0];

    if( trace > 0 ) {
      if( (i_iter % trace) == 0 ) {
        Rprintf("Iteration: %d bestvalit: %f bestmemit:", i_iter, gt_bestC[0]);
        for (j = 0; j < i_D; j++)
          Rprintf("%12.6f", gt_bestP[j]);
        Rprintf("\n");
      }
    }
  } /* end loop through generations */

  /* last population */
  k = 0;
  for (i = 0; i < i_NP; i++) {
    for (j = 0; j < i_D; j++) {
      gd_pop[k] = gta_oldP[i][j];
      k++;
    }
  }

  *gi_iter = i_iter;

  PutRNGstate();
  UNPROTECT(1);

}
```

Panel A: C version

```
    ta_oldP = ta_newP;                      // have selected NP mutants move on to next generation
    ta_oldC = ta_newC;

    if (i_check_winner)  {                  // check if the best stayed the same, if necessary
        int same = 1;
        for (int j = 0; j < i_D; j++) {
            if (t_bestitP[j] != t_bestP[j]) {
                same = 0;
            }
        }
        if (same && i_iter > 1)  {
            i_xav++;
            memcpy(REAL(par), t_bestP.memptr(), Rf_nrows(par) * sizeof(double));
            double tmp_best = ev->eval(par);// if re-evaluation of winner
            if (i_av_winner)         //  poss. letting winner be avg of all past generations
                t_bestC = ((1/(double)i_xav) * t_bestC) + ((1/(double)i_xav) * tmp_best) +
                    (d_bestvalit[i_iter-1] * ((double)(i_xav - 2))/(double)i_xav);
            else
                t_bestC = tmp_best;
        } else {
            i_xav = 1;
        }
    }
    t_bestitP = t_bestP;

    if ( (i_trace > 0)  &&  ((i_iter % i_trace) == 0) ) {
        Rprintf("Iteration: %d bestvalit: %f bestmemit:", i_iter, t_bestC);
        for (int j = 0; j < i_D; j++)
            Rprintf("%12.6f", t_bestP[j]);
        Rprintf("\n");
    }
  } // end loop through generations

    d_pop = ta_oldP;
    i_iterations = i_iter;
    l_nfeval = ev->getNbEvals();
    PutRNGstate();
}
```

Panel B: C++ version using **Rcpp**

Figure 14: `devol()` population processing and return preparation

```
double evaluate(long *l_nfeval, double *param, SEXP par, SEXP fcall, SEXP env)
{
   int i;
   SEXP sexp_fvec, fn;
   double f_result;

   for (i = 0; i < nrows(par); i++) {
      NUMERIC_POINTER(par)[i] = param[i];
   }
   PROTECT(fn = lang2(fcall, par));
      (*l_nfeval)++;  /* increment function evaluation count */

   PROTECT(sexp_fvec = eval(fn, env));
   f_result = NUMERIC_POINTER(sexp_fvec)[0];

   UNPROTECT(2);
   if(ISNAN(f_result))
     error("NaN value of objective function! \nPerhaps adjust the bounds.");

   return(f_result);
}
```

Panel A: C version

```
namespace Rcpp {
    namespace DE {

        class EvalBase {
        public:
            EvalBase() : neval(0) {};
            virtual double eval(SEXP par) = 0;
            unsigned long getNbEvals() { return neval; }
        protected:
            unsigned long int neval;
        };

        class EvalStandard : public EvalBase {
        public:
            EvalStandard(SEXP fcall_, SEXP env_) : fcall(fcall_), env(env_) {}
            double eval(SEXP par) {
                neval++;
                return defaultfun(par);
            }
        private:
            SEXP fcall, env;
            double defaultfun(SEXP par) {                // essentialy the same as the old evaluate
                SEXP fn = ::Rf_lang2(fcall, par);        // this could be done with Rcpp
                SEXP sexp_fvec = ::Rf_eval(fn, env);       // but is still a lot slower right now
                double f_result = REAL(sexp_fvec)[0];
                if (ISNAN(f_result))
                    ::Rf_error("NaN value of objective function! \nPerhaps adjust the bounds.");
                return(f_result);
            }
        };

        typedef double (*funcPtr)(SEXP);
        class EvalCompiled : public EvalBase {
        public:
            EvalCompiled( Rcpp::XPtr<funcPtr> xptr ) {
                funptr = *(xptr);
            };
            EvalCompiled( SEXP xps ) {
                Rcpp::XPtr<funcPtr> xptr(xps);
                funptr = *(xptr);
            };
            double eval(SEXP par) {
                neval++;
                return funptr(par);
            }
        private:
            funcPtr funptr;
        };

    }
}
```

Panel B: C++ version using **Rcpp**

Figure 15: `evaluate()` function versus Evaluation classes permitting R and C++ objective functions

```
DEoptim <- function(fn, lower, upper, control = DEoptim.control(), ...) {
  fn1  <- function(par) fn(par, ...)
  if (length(lower) != length(upper))
    stop("'lower' and 'upper' are not of same length")
  if (!is.vector(lower))
    lower <- as.vector(lower)
  if (!is.vector(upper))
    upper <- as.vector(upper)
  if (any(lower > upper))
    stop("'lower' > 'upper'")
  if (any(lower == "Inf"))
    warning("you set a component of 'lower' to 'Inf'. May imply 'NaN' results", immediate. = TRUE)
  if (any(lower == "-Inf"))
    warning("you set a component of 'lower' to '-Inf'. May imply 'NaN' results", immediate. = TRUE)
  if (any(upper == "Inf"))
    warning("you set a component of 'upper' to 'Inf'. May imply 'NaN' results", immediate. = TRUE)
  if (any(upper == "-Inf"))
    warning("you set a component of 'upper' to '-Inf'. May imply 'NaN' results", immediate. = TRUE)
  if (!is.null(names(lower)))
    nam <- names(lower)
  else if (!is.null(names(upper)) & is.null(names(lower)))
    nam <- names(upper)
  else
    nam <- paste("par", 1:length(lower), sep = "")

  ctrl <- do.call(DEoptim.control, as.list(control))
  ctrl$npar <- length(lower)
  if (ctrl$NP < 4) {
    warning("'NP' < 4; set to default value 50\n", immediate. = TRUE)
    ctrl$NP <- 50
  }
  if (ctrl$NP < 10*length(lower))
    warning("For many problems it is best to set 'NP' (in 'control') to be at least "
            "ten times the length of the parameter vector. \n", immediate. = TRUE)
  if (!is.null(ctrl$initialpop)) {
    ctrl$specinitialpop <- TRUE
    if(!identical(as.numeric(dim(ctrl$initialpop)), c(ctrl$NP, ctrl$npar)))
      stop("Initial population is not a matrix with dim. NP x length(upper).")
  }
  else {
    ctrl$specinitialpop <- FALSE
    ctrl$initialpop <- 0.0
  }
  ##
  ctrl$trace <- as.numeric(ctrl$trace)
  ctrl$specinitialpop <- as.numeric(ctrl$specinitialpop)
  ctrl$initialpop <- as.numeric(ctrl$initialpop)
```

Panel A: R version in **DEoptim**

```
DEoptim <- function(fn, lower, upper, control = DEoptim.control(), env, ...) {
  ##fn1  <- function(par) fn(par, ...)
  if (length(lower) != length(upper))
    stop("'lower' and 'upper' are not of same length")
  if (!is.vector(lower))
    lower <- as.vector(lower)
  if (!is.vector(upper))
    upper <- as.vector(upper)
  if (any(lower > upper))
    stop("'lower' > 'upper'")
  if (any(lower == "Inf"))
    warning("you set a component of 'lower' to 'Inf'. May imply 'NaN' results", immediate. = TRUE)
  if (any(lower == "-Inf"))
    warning("you set a component of 'lower' to '-Inf'. May imply 'NaN' results", immediate. = TRUE)
  if (any(upper == "Inf"))
    warning("you set a component of 'upper' to 'Inf'. May imply 'NaN' results", immediate. = TRUE)
  if (any(upper == "-Inf"))
    warning("you set a component of 'upper' to '-Inf'. May imply 'NaN' results", immediate. = TRUE)
  if (!is.null(names(lower)))
    nam <- names(lower)
  else if (!is.null(names(upper)) & is.null(names(lower)))
    nam <- names(upper)
  else
    nam <- paste("par", 1:length(lower), sep = "")
  if (missing(env))
    env <- new.env()

  ctrl <- do.call(DEoptim.control, as.list(control))
  ctrl$npar <- length(lower)
  if (ctrl$NP < 4) {
    warning("'NP' < 4; set to default value 50\n", immediate. = TRUE)
    ctrl$NP <- 50
  }
  if (ctrl$NP < 10*length(lower))
    warning("For many problems it is best to set 'NP' (in 'control') to be at least ten"
            " times the length of the parameter vector. \n", immediate. = TRUE)
  if (!is.null(ctrl$initialpop)) {
    ctrl$specinitialpop <- TRUE
    if(!identical(as.numeric(dim(ctrl$initialpop)), c(ctrl$NP, ctrl$npar)))
      stop("Initial population is not a matrix with dim. NP x length(upper).")
  }
  else {
    ctrl$specinitialpop <- FALSE
    ctrl$initialpop <- matrix(0,1,1)    # dummy matrix
  }
  ##
  ctrl$trace <- as.numeric(ctrl$trace)
  ctrl$specinitialpop <- as.numeric(ctrl$specinitialpop)
```

Panel B: R version in **RcppDE**

Figure 16: First half of R function `DEoptim()`

```
    outC <- .Call("DEoptimC", lower, upper, fn1, ctrl, new.env(),
                  PACKAGE = "DEoptim")
    ##
    if (length(outC$storepop) > 0) {
      nstorepop <- floor((outC$iter - ctrl$storepopfrom) / ctrl$storepopfreq)
      storepop <- list()
      cnt <- 1
      for(i in 1:nstorepop) {
        idx <- cnt:((cnt - 1) + (ctrl$NP * ctrl$npar))
        storepop[[i]] <- matrix(outC$storepop[idx], nrow = ctrl$NP, ncol = ctrl$npar,
                                byrow = TRUE)
        cnt <- cnt + (ctrl$NP * ctrl$npar)
        dimnames(storepop[[i]]) <- list(1:ctrl$NP, nam)
      }
    }
    else {
      storepop = NULL
    }

    ## optim
    bestmem <- as.numeric(outC$bestmem)
    names(bestmem) <- nam
    bestval <- as.numeric(outC$bestval)
    nfeval <- as.numeric(outC$nfeval)
    iter <- as.numeric(outC$iter)

    ## member
    names(lower) <- names(upper) <- nam
    bestmemit <- matrix(outC$bestmemit, nrow = iter,
                        ncol = ctrl$npar, byrow = TRUE)

    dimnames(bestmemit) <- list(1:iter, nam)
    bestvalit <- as.numeric(outC$bestvalit[1:iter])
    pop <- matrix(outC$pop, nrow = ctrl$NP, ncol = ctrl$npar,
                  byrow = TRUE)
    storepop <- as.list(storepop)

    outR <- list(optim = list(
                 bestmem = bestmem,
                 bestval = bestval,
                 nfeval = nfeval,
                 iter = iter),
               member = list(
                 lower = lower,
                 upper = upper,
                 bestmemit = bestmemit,
                 bestvalit = bestvalit,
                 pop = pop,
                 storepop = storepop))

    attr(outR, "class") <- "DEoptim"
    return(outR)
}
```

Panel A: R version in **DEoptim**

```
    outC <- .Call("DEoptim", lower, upper, fn, ctrl, env, PACKAGE = "RcppDE")
    ##
    if (length(outC$storepop) > 0) {
      nstorepop <- floor((outC$iter - ctrl$storepopfrom) / ctrl$storepopfreq)
      ## storepop <- list()
      ## cnt <- 1
      ## for(i in 1:nstorepop) {
      ##   idx <- cnt:((cnt - 1) + (ctrl$NP * ctrl$npar))
      ##   storepop[[i]] <- matrix(outC$storepop[idx], nrow = ctrl$NP, ncol = ctrl$npar,
      ##                           byrow = TRUE)
      ##   cnt <- cnt + (ctrl$NP * ctrl$npar)
      ##   dimnames(storepop[[i]]) <- list(1:ctrl$NP, nam)
      ## }
      storepop <- outC$storepop[1:nstorepop]
      for (i in 1:length(storepop)) dimnames(storepop[[i]]) <- list(1:ctrl$NP, nam)
    }
    else {
      storepop = NULL
    }

    ## optim
    bestmem <- as.numeric(outC$bestmem)
    names(bestmem) <- nam
    bestval <- as.numeric(outC$bestval)
    nfeval <- as.numeric(outC$nfeval)
    iter <- as.numeric(outC$iter)

    ## member
    names(lower) <- names(upper) <- nam
    #bestmemit <- matrix(outC$bestmemit, nrow = iter, ncol = ctrl$npar, byrow = TRUE)
    bestmemit <- outC$bestmemit

    dimnames(bestmemit) <- list(1:iter, nam)
    bestvalit <- as.numeric(outC$bestvalit[1:iter])
    #pop <- matrix(outC$pop, nrow = ctrl$NP, ncol = ctrl$npar, byrow = TRUE)
    pop <- outC$pop
    storepop <- as.list(storepop)

    outR <- list(optim = list(
                 bestmem = bestmem,
                 bestval = bestval,
                 nfeval = nfeval,
                 iter = iter),
               member = list(
                 lower = lower,
                 upper = upper,
                 bestmemit = bestmemit,
                 bestvalit = bestvalit,
                 pop = pop,
                 storepop = storepop))

    attr(outR, "class") <- "DEoptim"
    return(outR)
}
```

Panel B: R version in **RcppDE**

Figure 17: Second half of R function `DEoptim()`