Enhancing the speed of DAISIE model simulations

Laying a foundation for speed and efficiency improvements by translating core simulation code from R to C++

DAISIE (Dynamic Assembly of Island biotas through Speciation, Immigration and Extinction) is a framework for a dynamic stochastic mathematical model of island biogeography. It can be used to estimate the rates of colonisation, speciation and extinction from phylogenetic trees of isolated communities by maximum likelihood. It can also work the other way around; it can simulate such data sets given a set of rates.

This DAISIE framework is open-source and distributed as an R package. Therefore it is, unsurprisingly, almost entirely written in R. The R 'programming' language is an easy-to-use data manipulation and data representation language, and even though the built-in functions allow for rather intuitive ways to manipulate (large) data objects, they are quite slow. Therefore the aim of this mini-project was to replace time-consuming R functions with C++ ones while maintaining identical functionality.

To be able to maximise the result of the performance improvements, we first have to investigate what parts of the code precisely are the bottlenecks. During this project, we focussed on the core simulation function *DAISIE_sim_core_cr.R*.

Identifying bottlenecks

Before we are able to improve the performance, we first need to understand which parts (nested functions) of the simulation code in *DAISIE_sim_core_cr.R* are the slowest. We proceeded to profile the code using the R-package *ProfVis*. *ProfVis* creates an interactive graphical interface that aids users in unraveling what parts of their code are the slowest. It uses the *Rprof* function (part of the base R distribution) to collect call stack data at a specified time interval (10ms by default). It saves the data retrieved at every pause to a file, which tells the user exactly where most time is spent.

We profiled a simple piece of example code (first example from this file), and were confronted with a neat overview (Figure 1), immediately directing our attention to the following functions; *update_rates* and *DAISIE_nonoceanic_spec*.

Code	File	Memory (MB)		Time (ms)
▼ DAISIE_sim_cr	<expr>	-258.4	263.9	9590
▼ DAISIE_sim_cr_cs	DAISIE_sim_cr.R	-258.4	263.9	9590
▼ DAISIE_sim_core_cr	DAISIE_sim_cr_cs.R	-163.5	159.3	8020
▶ update_rates	DAISIE_sim_core_c...	-42.6	36.6	1720
▶ DAISIE_nonoceanic_spec	DAISIE_sim_core_c...	-26.1	23.1	1270
▶ testit::assert	DAISIE_sim_core_c...	-22.1	21.5	1080
▶ are_rates		-11.9	14.1	620
▶ DAISIE_sample_event_cr	DAISIE_sim_core_c...	-9.7	14.3	600
▶ sort.default		-14.3	10.4	580
▶ DAISIE_sim_update_state_cr	DAISIE_sim_core_c...	-8.1	13.3	500
calc_next_timeval	DAISIE_sim_core_c...	0	6.4	310
▶ are_area_pars		-6.0	2.7	170
rates <- update_rates(	DAISIE_sim_core_c...	0	4.3	130
▶ rbind	DAISIE_sim_core_c...	0	2.8	130
DAISIE_spec_tables	DAISIE_sim_core_c...	-2.6	1.0	70

Figure 1. ProfVis profiling result of an example simulation, represented as a table. Clearly *update_rates* and *DAISIE_nonoceanic_spec* are the two most time-consuming nested functions.

After these two functions were translated into C++, we proceeded to do the same for *DAISIE_sample_event_cr* and *DAISIE_sim_update_state_cr* and *DAISIE_spec_tables*. Not necessarily to improve the overall speed significantly, but in order to lay a more profound foundation for the eventual translation of the whole *DAISIE_sim_core_cr* file. All before-mentioned functions are rather deeply nested, calling for multiple conversions from R objects to C++ equivalents (e.g. a vector of vectors). Such operations are costly. Translating the whole file from R to C++ would likely speed up the whole process significantly. Sadly we were unable to achieve this within the limited time span of three weeks.

Usage of multiple programming languages

As mentioned, R is rather slow when compared to other programming languages, simply because it is an interpreted (non-compiled) language. The Rcpp R-package allows for seamless integration of C++ code within an R package, allowing users to benefit from both the speed and efficiency of C++, while also still being able to work with R data structures.

When replacing parts of R code with C++, data objects have to be received as, and converted back to R data structures. This is done using Rcpp Sugar, which allows you to create e.g. a NumericMatrix out of a vector of vectors (type double). To transform these R data objects to C++ and vice versa, we developed a few helper functions, such as; *getNumericMatrixR* and *getDoubleMatrixCpp* (see helper_functions.h).

While translating code from R to C++, we continuously tested the code to assess whether the code was still generating the correct output.

Test-driven development

Test-driven development entails that the functionality of (re-)written code is continuously (re-)evaluated by means of unit testing. Simply put, unit tests e.g. directly call a function using hard-coded input values in order to influence the outcome of that function. The function's output can then be tested for correctness using (in this case) *expect* statements (testthat package).

See the next page for the results

Results

As a result of the implementation of the above-mentioned functions in C++, we were able to improve execution speed significantly (Figure 2). The two most time-consuming functions (*update_rates* and *DAISIE_nonoceanic_spec*) were sped up tremendously while simultaneously using less memory (not included in the figure).

Code	File	Time (ms)
▼ DAISIE_sim_cr	<expr>	9210
▼ DAISIE_sim_cr_cs	DAISIE_sim_cr.R	9210
▼ DAISIE_sim_core_cr	DAISIE_sim_cr_cs.R	7600
▶ update_rates	DAISIE_sim_core_c…	1810
▶ DAISIE_nonoceanic_spec	DAISIE_sim_core_c…	1130
▶ testit::assert	DAISIE_sim_core_c…	980
▶ are_rates		660
▶ DAISIE_sample_event_cr	DAISIE_sim_core_c…	490
▶ sort.default		480
▶ DAISIE_sim_update_state_cr	DAISIE_sim_core_c…	460
rates <- update_rates(	DAISIE_sim_core_c…	160
calc_next_timeval	DAISIE_sim_core_c…	150
rbind	DAISIE_sim_core_c…	140
colnames<-	DAISIE_sim_core_c…	110
are_area_pars		110
sort	DAISIE_sim_core_c…	90
num_immigrants <- length(which(isl	DAISIE_sim_core_c…	40
DAISIE_spec_tables	DAISIE_sim_core_c…	40

Code	File	Time (ms)
▼ DAISIE_sim_cr	<expr>	5400
▼ DAISIE_sim_cr_cs	DAISIE_sim_cr.R	5400
▼ DAISIE_sim_core_cr	DAISIE_sim_cr_cs.R	3840
▶ testit::assert	DAISIE_sim_core_c…	1070
▶ are_rates		590
▶ sort.default		480
▶ DAISIE_sim_update_state_cr	DAISIE_sim_core_c…	270
▶ DAISIE_spec_tables_cpp	DAISIE_sim_core_c…	190
update_rates_cpp	DAISIE_sim_core_c…	190
DAISIE_nonoceanic_spec_cpp	DAISIE_sim_core_c…	160
▶ rbind	DAISIE_sim_core_c…	120
are_area_pars		110
calc_next_timeval	DAISIE_sim_core_c…	110
rates <- update_rates_cpp(	DAISIE_sim_core_c…	80
which	DAISIE_sim_core_c…	70
▶ sort	DAISIE_sim_core_c…	60
island <- DAISIE_create_island(	DAISIE_sim_core_c…	40
DAISIE_sample_event_cr_cpp	DAISIE_sim_core_c…	40

Figure 2. Profiling output of the original DAISIE simulation example (left), and the one where certain parts of the code were translated into C++ (right).

Future work

While this mini-project only touches on a small part of the DAISIE framework, it can be concluded that efforts on investigation and improvement of the simulation code are worthwhile. The specific code that was investigated and translated has improved significantly (~50%). This was achieved by a single person, who was unfamiliar with the code, in merely three weeks' time.

ProfVis proved to be an excellent tool to identify bottlenecks in the original code. As mentioned, the whole file (*DAISIE_sim_core_cr.R*) might improve in speed even more once fully translated, although this calls for some further investigation. On the contrary, for such a complete and production-ready package it might be more efficient to create Rcpp functions for only the most time-consuming functions of all (e.g. *update_rates* and *DAISIE_nonoceanic_spec*).

For an up-to-date TODO list, see the README file in the code repository.

All code changes were tracked and can be found in this GitHub repository.

Olle de Jong
S4970934