`fntl`: Numerical Tools for Rcpp and Lambda Functions

Andrew M. Raim

R programmers can combine R and C++ to effectively navigate a variety of computing tasks: R excels as a language for interactive tasks such as data wrangling, analysis, and plotting; on the other hand, C++ can be used to efficiently carry out intensive computations. Rcpp and related tools have greatly simplified interoperability between the two languages. However, numerical computing tasks that involve functions as arguments, such as integration, root-finding, and optimization, which are routinely carried out in R, are not as straightforward in C++ within the Rcpp framework. The `fntl` package seeks to improve this by providing a straightforward API for numerical tools where functional arguments are specified as C++ lambda functions. Like functions in R, lambda functions can be defined in the course of a C++ program, "capturing" variables in the surrounding environment if desired, and be passed as arguments to other functions. This enables the development of R-like programs in C++, which may be appealing to Rcpp users compared to existing alternatives in the extended Rcpp family of packages. Because the overhead to evaluate a lambda function is low compared to that of evaluating an R function from C++, good performance is also possible in this paradigm.

Table of contents

Document was compiled 2024-08-20 14:14:08 EDT and corresponds to `fntl` version 0.1.0. **Contact:** andrew.raim@gmail.com, Center for Statistical Research & Methodology, U.S. Census Bureau, Washington, DC, 20233, U.S.A.

Disclaimer and Acknowledgments

This document is released to inform interested parties of ongoing research and to encourage discussion of work in progress. Any views expressed are those of the author and not those of the U.S.

Census Bureau.

Thanks to Drs. Joseph Kang and Tommy Wright (U.S. Census Bureau) for reviewing a draft of this document.

Although there are no guarantees of correctness of the `fntl` package, reasonable efforts will be made to address shortcomings. Comments, questions, corrections, and possible improvements can be communicated through the project's Github repository (https://github.com/andrewraim/fntl).

1 Introduction

Users of R (R Core Team 2024a) can implement intensive computations in compiled C++ and engage in interactive computation through the interpreted R language. The Rcpp package (Eddelbuettel et al. 2024) simplifies the process by automating interoperation between the two languages and providing an intuitive API in C++. However, numerical tools such as integration, root-finding, and optimization, which are routinely used in R, are not as readily available or easy to use when carrying out lower-level programming in C++. Numerical tools in R may be invoked from C++, but this incurs an overhead which can offset gains in performance which motivate the use of C++. The purpose of the `fntl` package is to facilitate access to such numerical tools in C++ using lambda functions. Here, R-like code can be achieved in C++ where functions are defined on the fly and passed as arguments to other routines. The name "fntl" is a portmanteau of "**f**unctions", "**n**umerical **t**ools", and "**l**ambdas".

Ihaka and Gentleman (1996) describe programming with functions as one of the main motivations in developing R. Namely, functions may be defined dynamically in the course of a program and can make use of variables from the surrounding environment. As an example, consider the following R snippet to compute the maximum likelihood estimate (MLE) of $X_1, \ldots, X_n \sim \mathrm{N}(\mu, 1)$.

```r
mu_true = 0
x = rnorm(n = 200, mean = mu_true, sd = 1)
loglik = function(mu) { sum(dnorm(x, mu, 1, log = TRUE)) }
optimize(loglik, lower = -100, upper = 100, maximum = TRUE)
```

```
$maximum
[1] -0.05775928

$objective
[1] -287.4428
```

Here the generated sample `x` is "baked in" to the definition of `loglik` so that `loglik` can be regarded as a univariate function of `mu` to be used with `optimize`. The ability to construct functions such as `loglik` in the course of a program and pass them to other functions, as any other variable, can be tremendously convenient. Wickham (2019) discusses some design patterns which are possible using functions in R. However, such conveniences come at a cost, as performance can be inefficient in both speed and memory management.

Performance limitations in R can sometimes be worked around; for example, loops and apply statements are typically slow because they are executed by an interpreter, but matrix operations are typically fast because they make use of calls to BLAS, LAPACK, and other compiled matrix libraries. Refactoring loops into matrix operations can yield a significant performance improvement when it is feasible. Compiled languages such as FORTRAN and C/C++ are often used for high performance implementation of numerical methods; however, they are not well-suited for interactive usages such as data wrangling, data analysis, and graphics like R, MATLAB, or Python. Julia (Bezanson et al. 2017) has emerged over the past decade as a language for scientific computing which is both suitable to be used interactively and compiles into efficient executable code.

R supports integration with FORTRAN and C/C++ so that high performance code can be used interactively from the R interpreter; therefore, it remains a viable alternative to languages like Julia. The `Rcpp` package largely automates integration with C/C++ code so that users can focus their efforts on solving the problem at hand. For a function call from R to C++, this automation includes unpacking the arguments from `SEXP` objects into C++ objects and later packing the result into an `SEXP` object to be returned. Overhead from repeated packing and unpacking can hinder performance; on the other hand, performance while submerged in C++ is free of this overhead. The Rcpp API provides general C++ classes for vectors, matrices, lists, and other routinely used objects from R. Additional Rcpp extension packages have been developed to provide APIs with more application-specific classes; for example, RcppArmadillo (Eddelbuettel and Sanderson 2014) exposes the API from Armadillo for numerical matrix operations.

A performance penalty is also suffered when defining a function in R and using it from C++ with Rcpp. Calling the function from C++ requires going back through the R runtime environment in addition to packing arguments into `SEXP` objects and unpacking return values from `SEXP` objects. Doing so repeatedly accumulates the penalty, and can negate performance benefits of using C++. Instead, we will consider utilizing lambda functions in C++, which were introduced in the C++11 specification. There are alternative methods of defining and passing functions as objects - such as through function pointers and functor classes - but lambda functions seem best aligned with the R style discussed by Ihaka and Gentleman (1996). Traditional functions and classes used in the function pointer and functor approaches, respectively, are declared in their own blocks prior to use. Another major difference lies in auxiliary data - such as sample data in a loglikelihood function - which are not considered to be primary arguments. With function pointers, auxiliary data are passed as additional arguments that each caller must furnish. Functors are classes which expose the function of interest and encapsulate auxiliary data using member variables.

To demonstrate lambdas, consider the following C++ code snippet.

```
auto f = [](double x, double y) -> double { return x*y; };
```

The function `f` may be invoked as usual with an expression such as `f(x, y)`. Here, `f` is a lambda with `double` arguments `x` and `y` which returns the product `x*y` as another `double`. The `auto` keyword instructs the compiler to infer the type of `f` from the return type of the right-hand side of the equality operator. In the previous example, the return type of the lambda can also be inferred, and we may rewrite it as follows.

```
auto f = [](double x, double y) { return x*y; };
```

Like an R function, variables in scope of the lambda may be "baked in" to it. Here is an example from our MLE setting.

```
Rcpp::NumericVector x = Rcpp::rnorm(200);
auto loglik = [&](double mu) {
    double out = Rcpp::sum(Rcpp::dnorm(x, mu, 1, true));
    return out;
};
```

The function `loglik` takes a `double` argument `mu` and returns a `double`. The randomly generated data `x` is included in the function definition; C++ documentation refers to `x` as a *capture* of the lambda `loglik`. The bracketed expression `[&]` specifies that captures of `loglik` should be by reference; alternatively, `[=]` would specify that captures should be done by value so that copies of the original variable are used. It is also possible to specify by-reference or by-value for each captured variable.

To be able to pass a lambda with captures as an argument to other functions, we wrap it in a Standard Template Library (STL) function as follows.

```
std::function<double(double)> loglik = [&](double mu) {
    double out = Rcpp::sum(Rcpp::dnorm(x, mu, 1, true));
    return out;
};
```

Note that the template argument of `std::function<double(double)>` describes the domain and range of the function: the `double` in parentheses is the argument and the `double` outside indicates the return type.

Lambda functions can be used directly with the Rcpp API in some cases. For example, an Rcpp Gallery post demonstrates the use of the STL `transform` function to apply a lambda to each element of a vector. The `fntl` package avoids duplicating this existing functionality.

Numerical tools implemented in `fntl` are covered in other Rcpp-related packages to some extent. The `RcppNumerical` package (Qiu et al. 2023) implements numerical integration - both univariate and multivariate - and optimization using a limited memory Broyden, Fletcher, Goldfarb and Shanno method. This is accomplished by providing an Rcpp interface to several open source libraries. The `roptim` package (Pan 2022) provides an Rcpp interface to the R API to call individual optimization methods underlying the `optim` function. Function arguments in both `RcppNumerical` and `roptim` are specified via C++ functors. This vignette of `roptim` provides a discussion on calling the R API which was helpful in the development of `fntl`. A number of numerical utilities in the GSL (Galassi et al. 2009) and Boost libraries are provided by the `RcppGSL` (Eddelbuettel and Francois 2024) and `BH` (Eddelbuettel, Emerson, and Kane 2024) packages, respectively. The `RcppGSL` package requires installation of the underlying GSL library to build the source package. The author of the present document finds the interfaces of both GSL and Boost to be somewhat daunting, which has motivated a search for alternatives.

The `fntl` package is guided by several design principles. The interface is intended to be simple and familiar to R users. External dependencies beyond the R platform itself and the Rcpp package are

avoided; this is to support use in locked-down computing environments where adding or upgrading system libraries may be nontrivial. For numerical methods, we first prefer to make use of functions exposed as entry points in the R API (R Core Team 2024b, sec. 6). In cases where a desired R method is not exposed in the API, we roll our own implementation. Such methods in `fntl` may not be exactly the same as those in R, but are intended to be comparable.

Section 2 presents an overview of the `fntl` API. Subsequent sections present the API in detail by topic: numerical integration in Section 3, numerical differentiation in Section 4, root-finding in Section 5, univariate optimization in Section 6, multivariate optimization in Section 7, and matrix operations in Section 8. The C++ examples in each section can be obtained as standalone files in the vignettes/examples folder of the `fntl` source repository.

2 Overview

2.1 A First Example

We will first consider a brief example to illustrate use of the `fntl` package. If attempting to follow along, ensure that you have successfully installed the `fntl` package on your system. The following C++ code, given in the file `examples/first.cpp`, computes the integral $B(a, b) = \int_0^1 x^{a-1}(1-x)^{b-1}dx$.

```cpp
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List first_ex(double a, double b)
{
    fntl::integrate_args args;
    args.subdivisions = 200L;

    fntl::dfd f = [&](double x) {
        return std::pow(x, a - 1) * std::pow(1 - x, b - 1);
    };
    fntl::integrate_result out = fntl::integrate(f, 0, 1, args);

    Rprintf("value: %g\n", out.value);
    Rprintf("status: %d\n", to_underlying(out.status));

    return Rcpp::wrap(out);
}
```

There are a number of points to note in this example.

1. The first two lines - the `depends` attribute and the include of `fntl.h` - are needed to access the `fntl` library from C++.

2. Functions and other objects in `fntl` are accessed through the `fntl` namespace (e.g., `fntl::integrate`).
3. The call to `integrate` follows a similar pattern as many functions in `fntl`. Primary arguments include the function `f` and the bounds of the integral, while optional arguments such as the number of subdivisions are passed in a struct of type `integrate_args`. The result of `integrate` is an `integrate_result` struct containing the integral approximation `value`, a return code `status`, and several other outputs from the operation.
4. The `integrate_args` struct uses default values for any unspecified arguments. Users often will not want to specify values such as tolerances or verbosities. Similar argument names and default values as the corresponding R function are used when possible.
5. The integrand `f` is defined as a `dfd`; this is a typedef in `fntl` which is a shorthand for `std::function<double(double)>`.
6. The last line shows the `integrate_result` being wrapped into an Rcpp `List` so that it becomes an R list when returned to R.
7. The `status` code is a value from the `integrate_status` enum class. Here it is converted to an integer using the function `to_underlying`.
8. An R function `first_ex` is generated by specifying an `Rcpp::export` annotation. This is done for most functions in this document to facilitate demonstrations.

Let us invoke the `first_ex` function through R.

```
Rcpp::sourceCpp("examples/first.cpp")
out = first_ex(2, 3)
```

```
value: 0.0833333
status: 0
```

```
print(out$value)
```

```
[1] 0.08333333
```

2.2 Arguments

There are a number of `args` structs which represent optional arguments to functions. `integrate_args` is one example; other `args` types have a prefix matching their corresponding function. Each of these may be instantiated from an Rcpp `List` or exported to an Rcpp `List` using the `as` and `wrap` mechanisms, respectively.[1]

[1] Details on implementing these mechanisms may be found in the Rcpp Extending vignette.

```
// Create args and export to a List
fntl::integrate_args args0;
Rcpp::List x = Rcpp::wrap(args0);

// Instantiate a second args struct from the list x
x["stop_on_error"] = false;
fntl::integrate_args args1 = Rcpp::as<fntl::integrate_args>(x);
```

When instantiating an `args` struct from a `List`, we throw an error if the list contains any elements with names which are not expected by the struct; this is to protect against mistakes which could occur when a field is named incorrectly where the effects may be subtle and difficult to track down.

```
// This will cause an exception
x["abcdefg"] = 0;
fntl::integrate_args args1 = Rcpp::as<fntl::integrate_args>(x);
```

In most cases, a function that takes optional `args` has an alternative form where the `args` may be omitted; this form assumes all default values for the `args`. For example, our call to `integrate` in Section 2.1 could omit the `args` as follows.

```
fntl::integrate_result out = fntl::integrate(f, 0, 1);
```

2.3 Results

Similar to `args`, there are a number of `result` structs that represent the output of a function. `integrate_result` is an example; other `result` types have a prefix matching their corresponding function. A `result` may be exported to an Rcpp `List` using the `wrap` mechanism. This was seen in Section 2.1.

```
fntl::integrate_result out = fntl::integrate(f, 0, 1, args);
Rcpp::List x = Rcpp::wrap(out);
```

2.4 Status Codes

Error conditions may result in an exception so that no return value is produced. Some conditions - such as failure to converge within a given number of iterations - may not result in an exception. Here, a `status` code is returned with the result to indicate a possible issue that may warrant further investigation. The example in Section 2.1 illustrates `integrate_status` returned by `integrate`; other `status` types have a prefix matching their corresponding function. Each `status` type is an enum class derived from either `int` or `unsigned int`: an enumeration that can be constructed from an integer or converted to an integer using the included function `to_underlying`.[2]

[2]This function is based on a post from StackOverflow.

```
fntl::integrate_status status = fntl::integrate_status::OK;  // Define a status
int err = to_underlying(status);                             // status to int
status1 = fntl::integrate_status(err);                        // int to status
```

2.5 Function Typedefs

`fntl` defines shorthands for several commonly used function types, given as follows, which are used throughout the present document.

```
typedef function<double(double)> dfd;
typedef function<double(const NumericVector&)> dfv;
typedef function<double(const NumericVector&, const NumericVector&)> dfvv;
typedef function<NumericVector(const NumericVector&)> vfv;
typedef function<NumericMatrix(const NumericVector&)> mfv;
```

Type names are intended to convey argument and return types as briefly as possible. Symbols to the right of `f` represent arguments while those to the left represent the return type; in particular, `d` is **d**ouble, `v` is numeric **v**ector, and `m` is numeric **m**atrix. For example, a `dfvv` takes two vectors and returns a double.

The namespace `std` for `function` and `Rcpp` for `NumericVector` and `NumericMatrix` have been omitted in the display so that statements each fit on a single line.

2.6 Constants

The following constants are defined in `fntl` and utilized in the API.

```
double mach_eps = std::numeric_limits<double>::epsilon();
double mach_eps_2r = sqrt(mach_eps);
double mach_eps_4r = std::pow(mach_eps, 0.25);
unsigned int uint_max = std::numeric_limits<unsigned int>::max();
```

These correspond to machine epsilon ϵ, $\epsilon^{1/2}$, $\epsilon^{1/4}$, and the maximum value of an unsigned integer.

2.7 Error Actions

Several functions in `fntl` take an `error_action` as an input to determine how to react in an error state. Here is the definition of `error_action`.

```
enum class error_action : unsigned int {
    STOP = 3L,                                                    ①
    WARNING = 2L,                                                 ②
```

```
    MESSAGE = 1L,                                                    ③
    NONE = 0L                                                        ④
};
```

① Throw an exception.
② Emit a warning and proceed.
③ Print a message and proceed.
④ Do not take any of the above actions and proceed.

Functions making use of an `error_action` typically also return a status code as in Section 2.4, which can be inspected by the caller if an exception is not thrown.

2.8 Inferring Return Types

Section 1 mentioned that the return type of a lambda does not necessarily need to be specified in its interface. However, we must be vigilant when allowing the type to be inferred, especially when working with `Rcpp` objects which are converted seamlessly behind the scenes.

The following example appears harmless but is likely to crash R.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List crash_ex(Rcpp::NumericVector x0)
{
    fntl::dfv f = [](Rcpp::NumericVector x) { return Rcpp::sum(x*x); };
    auto out = fntl::gradient(f, x0);
    return Rcpp::wrap(out);
}
```

The issue appears to be that `Rcpp::sum` does not necessarily return a `double` - the expected result of a `fntl::dfv` - but something else that can be readily converted to a `double`.

One way to address this is to specify that `double` is the return type of the lambda.

```
fntl::dfv f = [](Rcpp::NumericVector x) -> double { return Rcpp::sum(x*x); };
```

A second way to address the issue is to explicitly convert the result to a `double` before returning.

```
fntl::dfv f = [](Rcpp::NumericVector x) {
    double out = Rcpp::sum(x*x);
    return out;
};
```

2.9 R Functions as Lambdas

Although not a primary intended use of the `fntl` package, it is possible to use it with R functions. This is accomplished by wrapping an `Rcpp::Function` into a lambda. This will incur the usual overhead of calling R code from C++, but may be useful for testing or in situations where the overhead is a small proportion of the run time.

Here is an example based on the first example in Section 2.1.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List callr_ex(Rcpp::Function f)
{
    fntl::dfd ff = [&](double x) {
        Rcpp::NumericVector out = f(x);
        return out(0);
    };

    fntl::integrate_result out = fntl::integrate(ff, 0, 1);
    return Rcpp::wrap(out);
}
```

We create function `f` in R and pass it as an argument to `callr_ex`. The lambda `ff` calls the function `f`, implicitly converting the input `x` into an `Rcpp::NumericVector`, and converts the output from an `Rcpp::NumericVector` to a `double`. Let us demonstrate a call to the function `callr_ex` from R.

```
Rcpp::sourceCpp("examples/callr.cpp")
a = 2
b = 3
f = function(x) { x^(a - 1) * (1 - x)^(b - 1) }
out = callr_ex(f)
print(out$value)
```

```
[1] 0.08333333
```

2.10 R Interface

The `fntl` package includes an R interface which may be used to invoke much of the underlying C++ API. This is intended for demonstration and testing purposes only; performance will generally suffer here because of the overhead in moving between C++ and R. In real applications, R code should make use of mainstream R functions rather than this R interface.

For example, the `integrate0` R function is included to call the underlying `integrate` C++ function shown in Section 2.1. (The suffix "0" is added to avoid a naming clash with R's `integrate` function). The R function `integrate_args` can be used to construct a list which is suitable to pass to `integrate0`.

```
args = integrate_args()
print(args)
```

```
$subdivisions
[1] 100

$rel_tol
[1] 0.0001220703

$abs_tol
[1] 0.0001220703

$stop_on_error
[1] TRUE
```

```
a = 2; b = 3
f = function(x) { x^(a-1) * (1-x)^(b-1) }
out = integrate0(f, 0, 1, args)
print(out$value)
```

```
[1] 0.08333333
```

2.11 Performance Illustration

To illustrate performance characteristics, let us consider a brief simulation of the relationship between mean-squared error (MSE) and sample size in a logistic regression, suppose $Y_i \sim \text{Ber}(p_i)$ are independent with $p_i = \text{logit}^{-1}(\beta_0 + \beta_1 x_i)$. Data-generating values of the coefficients are taken to be $\beta_0 = 0$ and $\beta_1 = 1$, respectively. We take R draws of $(y_1, \ldots, y_n)$ and compute the MLE $\hat{\beta}^{(r)} = (\hat{\beta}_0^{(r)}, \hat{\beta}_1^{(r)})$ for the rth draw using the L-BFGS-B optimization method. The MSE associated with sample size n is computed as $\text{MSE}_n = \frac{1}{R}\sum_{r=1}^{R} \|\hat{\beta}^{(r)} - \beta\|^2$; this is computed for sample sizes $n \in \{100, 200, 500, 1000, 10000\}$. We describe four versions of the code; to see the implementations, refer to the corresponding source files in the `examples` folder.

```
source("examples/timing1.R")
Rcpp::sourceCpp("examples/timing2.cpp")
Rcpp::sourceCpp("examples/timing3.cpp")
Rcpp::sourceCpp("examples/timing4.cpp")
```

```
n_levels = c(100, 200, 500, 1000, 10000)
```

The first version of the program `timing1_ex` is written in pure R. This version uses a naively coded version of the loglikelihood based on a loop over $y_1, \ldots, y_n$. Experienced R users will recognize that vectorization will dramatically improve the performance. However, we will proceed with the slow loglikelihood for illustration. A particular run of this code took 1.31 minutes.

```
set.seed(1234)
start = Sys.time()
timing1_ex(R = 200, n_levels)
print(Sys.time() - start)
```

The second version `timing2_ex` ports `timing1_ex` from R to C++, where loops generally need not be avoided to achieve good performance. Here we define the loglikelihood as a lambda function and use the `lbfgsb` function described in Section 7.3 to carry out optimization. A run of this code took 22.17 seconds.

```
set.seed(1234)
start = Sys.time()
timing2_ex(R = 200, n_levels)
print(Sys.time() - start)
```

A third version of the code `timing3_ex` demonstrates that overhead of calling an R function from C++ does not necessarily result in poor performance. Here we make a vectorized call to `dbinom` for each evaluation of the loglikelihood. In this case, the overhead does not contribute significantly to the run time: a run took 25.94 seconds.

```
set.seed(1234)
start = Sys.time()
timing3_ex(R = 200, n_levels)
print(Sys.time() - start)
```

Finally, a fourth version of the code `timing4_ex` demonstrates a case where the overhead of repeatedly calling an R function from C++ results in abysmal performance. Here we revert back to the loop in `timing2_ex`, but call the `plogis` function in R rather than use one provided in Rcpp. A run of this code took 9.89 minutes.

```
set.seed(1234)
start = Sys.time()
timing4_ex(R = 200, n_levels)
print(Sys.time() - start)
```

2.12 Pass by Value and Reference

References and const references can be used in C++ code to avoid making unnecessary copies of variables which waste time and memory. This additional level of control (and responsibility) is typically not considered in R programming, where pass-by-value is the standard. Consider the following function, which adds one to each element of a matrix and returns the sum of the result.

```
double sum1p(Rcpp::NumericMatrix x) { return Rcpp::sum(x + 1); }
```

Here a copy of the matrix `x` is passed to `sum1p`. This pass-by-value can be changed to pass-by-reference which avoids copying `x`.

```
double sum1p_1(Rcpp::NumericMatrix& x) { return Rcpp::sum(x + 1); }
```

The body of `sum1p_1` does not alter `x`, but there are no safeguards in place to enforce that it does not. This might raise anxiety for a user of `sum1p_1` about whether their `x` has been modified. To alleviate their anxiety, we can provide a safeguard using a const reference.

```
double sum1p_2(const Rcpp::NumericMatrix& x) { return Rcpp::sum(x + 1); }
```

The compiler will emit an error if `sum1p_2` attempts to modify `x`.

Examples given in this document tend to use pass-by-value to avoid extra visual clutter of const references. However, the `fntl` package makes frequent use of const references in both the API and internal implementation. It is recommended that users consider consider which of the three - by-value, by-reference, or by-const-reference - is most appropriate for their application in actual usage.

3 Integration

Compute the integral

$$\int_a^b f(x)dx,$$

where limit a may be finite or $-\infty$ and limit b may be finite or ∞. Directly uses the C functions Rdqags and Rdqagi underlying the R function integrate. These functions are based on two respective QUADPACK routines: dqags for the case when both limits are finite and dqagi for the case when one or both limits are infinite (Piessens et al. 1983).

Function

Source code is in the file inst/include/integrate.h.

```
integrate_result integrate(
    const dfd& f,                                                    ①
    double lower,                                                    ②
```

```
    double upper,                                                    ③
    const integrate_args& args                                       ④
)

integrate_result integrate(
    const dfd& f,                                                    ①
    double lower,                                                    ②
    double upper                                                     ③
)
```

① Function to use as the integrand.
② Lower limit a of integral; may be `R_NegInf`.
③ Upper limit b of integral; may be `R_PosInf`.
④ Additional arguments.

Optional Arguments

```
struct integrate_args {
    unsigned int subdivisions = 100L;                                ①
    double rel_tol = mach_eps_4r;                                    ②
    double abs_tol = mach_eps_4r;                                    ③
    bool stop_on_error = true;                                       ④
};
```

① The maximum number of subintervals.
② Relative accuracy requested.
③ Absolute accuracy requested.
④ If `true`, errors in `integrate` raise exceptions.

Result

```
struct integrate_result {
    double value;                                                    ①
    double abs_error;                                                ②
    int subdivisions;                                                ③
    integrate_status status;                                         ④
    int n_eval;                                                      ⑤
    std::string message;                                             ⑥

    operator SEXP() const;                                           ⑦
};
```

① The final approximation of the integral.
② Estimate of the modulus of the absolute error.
③ The number of subintervals produced in the subdivision process.
④ A code describing the status of the operation.

⑤ Number of function evaluations.
⑥ A message describing the status of the operation.
⑦ Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `integrate_result` as an `Rcpp::List`. The fields here directly correspond to those in `integrate_result`.

Name	Type	Description
`value`	`Rcpp::NumericVector`	Length 1
`abs_error`	`Rcpp::NumericVector`	Length 1
`subdivisions`	`Rcpp::IntegerVector`	Length 1
`status`	`Rcpp::IntegerVector`	Length 1
`n_eval`	`Rcpp::IntegerVector`	Length 1
`message`	`Rcpp::StringVector`	Length 1

Status Codes

```
enum class integrate_status : int {
    OK = 0L,                                    ①
    MAX_SUBDIVISIONS = 1L,                      ②
    ROUNDOFF_ERROR = 2L,                        ③
    BAD_INTEGRAND_BEHAVIOR = 3L,                ④
    ROUNDOFF_ERROR_EXTRAPOLATION_TABLE = 4L,    ⑤
    PROBABLY_DIVERGENT_INTEGRAL = 5L,           ⑥
    INVALID_INPUT = 6L                          ⑦
};
```

① OK.
② maximum number of subdivisions reached.
③ roundoff error was detected.
④ extremely bad integrand behaviour.
⑤ roundoff error is detected in the extrapolation table.
⑥ the integral is probably divergent.
⑦ the input is invalid.

Example

Compute the integral $\int_0^\infty e^{-x^2/2}dx$. A C++ function with Rcpp interface is defined in the file `examples/integrate.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List integrate_ex(double lower, double upper)
{
```

```
    fntl::dfd f = [](double x) { return exp(-pow(x, 2) / 2); };
    auto out = fntl::integrate(f, lower, upper);
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/integrate.cpp")
out = integrate_ex(0 , Inf)
print(out)
```

```
$value
[1] 1.253314

$subdivisions
[1] 3

$n_eval
[1] 75

$abs_error
[1] 0.0001173243

$status
[1] 0

$message
[1] "OK"
```

4 Differentiation

This section presents methods for numerical differentiation. First we present simple finite differences, then Richardson extrapolation to automatically select a step size, then the functions to compute the gradient, Jacobian, and Hessian. The latter three make use of Richardson extrapolation.

4.1 Finite Differences

Compute the first and second derivatives of $f : \mathbb{R}^n \to \mathbb{R}$ numerically at point x. Denote e_i as the ith column of an $n \times n$ identity matrix. First derivatives in the ith coordinate are computed as

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h},$$
$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h},$$
$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x) - f(x - he_i)}{h},$$

for given $h > 0$ using symmetric, forward, or backward differences respectively. Second derivatives in the ith and jth coordinate, with $i, j \in \{1 \dots, n\}$, are computed as

$$\frac{\partial^2 f(x)}{\partial x_i x_j} \approx \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i - h_j e_j) - f(x - h_i e_i + h_j e_j) + f(x - h_i e_i - h_j e_j)}{4 h_i h_j},$$
$$\frac{\partial^2 f(x)}{\partial x_i x_j} \approx \frac{f(x + h_i e_i + 2h_j e_j) - f(x + he_i) - f(x + h_j e_j) + f(x)}{h_i h_j},$$
$$\frac{\partial^2 f(x)}{\partial x_i x_j} \approx \frac{f(x) - f(x - h_i e_i) - f(x - h_j e_j) + f(x - h_i e_i - h_j e_j)}{h_i h_j},$$

for given $h_1 > 0$ and $h_2 > 0$ using symmetric, forward, or backward differences respectively. The accuracy of these derivatives depends on the nature of the function f at x and the choice of h. See Section 5.7 of Press et al. (2007) for discussion.

Function

Primary location of source code is the file inst/include/fd-deriv.h.

```
double fd_deriv(
    const dfv& f,                                                    ①
    const Rcpp::NumericVector& x,                                    ②
    unsigned int i,                                                  ③
    double h,                                                        ⑤
    const fd_types& fd_type = fd_types::SYMMETRIC                    ⑦
)

double fd_deriv2(
    const dfv& f,                                                    ①
    const Rcpp::NumericVector& x,                                    ②
    unsigned int i,                                                  ③
    unsigned int j,                                                  ④
    double h_i,                                                      ⑤
    double h_j,                                                      ⑥
    const fd_types& fd_type = fd_types::SYMMETRIC                    ⑦
)
```

① Function d to differentiate.
② Point x at which derivative is taken.
③ First coordinate to differentiate.
④ Second coordinate to differentiate.
⑤ Step size in the first coordinate.
⑥ Step size in the second coordinate
⑦ Type of finite difference.

The functions `fd_deriv` and `fd_deriv2` compute first and second derivatives, respectively. The options for `fd_type` are given in the following enumeration class.

```
enum class fd_types : unsigned int {
    SYMMETRIC = 0L,
    FORWARD = 1L,
    BACKWARD = 2L
};
```

Example

Compute the first and second derivatives of $f(x) = \sin(b^\top x)$ at $x_0 = (1/2, \dots, 1/2)$ where $b = (1, \dots, n)$. The gradient and Hessian are given in closed-form by

$$\frac{\partial f(x)}{\partial x} = b\cos(b^\top x),$$
$$\frac{\partial^2 f(x)}{\partial x \partial x^\top} = -bb^\top \sin(b^\top x).$$

Let us first prepare the problem in R.

```
n = 3
b = seq_len(n)
x0 = rep(0.5, n)
eps = 0.001  ## Fix a step size for finite differences
g = function(x) { b * cos(sum(b * x)) }
h = function(x) { -tcrossprod(b) * sin(sum(b * x)) }
```

To demonstrate the numerical first derivative, a C++ function with Rcpp interface is defined in the file `examples/fd-deriv.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
double fd_deriv_ex(Rcpp::NumericVector x0, unsigned int i, double h,
    unsigned int type)
{
    fntl::dfv f = [](Rcpp::NumericVector x) {
        double ss = 0;
```

```
        for (unsigned int i = 0; i < x.length(); i++) { ss += (i+1) * x(i); }
        return std::sin(ss);
    };

    return fntl::fd_deriv(f, x0, i, h, fntl::fd_types(type));
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/fd-deriv.cpp")
out3 = out2 = out1 = numeric(n)
for (i in 1:n) {
    out1[i] = fd_deriv_ex(x0, i-1, eps, type = 0)  ## Symmetric
    out2[i] = fd_deriv_ex(x0, i-1, eps, type = 1)  ## Forward
    out3[i] = fd_deriv_ex(x0, i-1, eps, type = 2)  ## Backward
}
print(out1)
```

```
[1] -0.9899923 -1.9799837 -2.9699730
```

```
print(out2)
```

```
[1] -0.9900629 -1.9802659 -2.9706081
```

```
print(out3)
```

```
[1] -0.9899218 -1.9797014 -2.9693380
```

```
print(g(x0))
```

```
[1] -0.9899925 -1.9799850 -2.9699775
```

To demonstrate the numerical second derivative, a C++ function with Rcpp interface is defined in the file `examples/fd-deriv2.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
double fd_deriv2_ex(Rcpp::NumericVector x0, unsigned int i, unsigned int j,
```

```
    double h_i, double h_j, unsigned int type)
{
    fntl::dfv f = [](Rcpp::NumericVector x) {
        double ss = 0;
        for (unsigned int i = 0; i < x.length(); i++) { ss += (i+1) * x(i); }
        return std::sin(ss);
    };

    return fntl::fd_deriv2(f, x0, i, j, h_i, h_j, fntl::fd_types(type));
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/fd-deriv2.cpp")
out3 = out2 = out1 = matrix(NA, n, n)
for (i in 1:n) {
for (j in 1:n) {
    out1[i,j] = fd_deriv2_ex(x0, i-1, j-1, eps, eps, type = 0)  ## Symmetric
    out2[i,j] = fd_deriv2_ex(x0, i-1, j-1, eps, eps, type = 1)  ## Forward
    out3[i,j] = fd_deriv2_ex(x0, i-1, j-1, eps, eps, type = 2)  ## Backward
}
}
print(out1)
```

```
           [,1]       [,2]       [,3]
[1,] -0.1411200 -0.2822398 -0.4233593
[2,] -0.2822398 -0.5644793 -0.8467182
[3,] -0.4233593 -0.8467182 -1.2700763
```

```
print(out2)
```

```
           [,1]       [,2]       [,3]
[1,] -0.1401299 -0.2792697 -0.4174191
[2,] -0.2792697 -0.5565588 -0.8318671
[3,] -0.4174191 -0.8318671 -1.2433437
```

```
print(out3)
```

```
           [,1]       [,2]       [,3]
[1,] -0.1421099 -0.2852096 -0.4292989
[2,] -0.2852096 -0.5723986 -0.8615668
[3,] -0.4292989 -0.8615668 -1.2968031
```

```
h(x0)
```

```
          [,1]     [,2]     [,3]
[1,] -0.14112 -0.28224 -0.42336
[2,] -0.28224 -0.56448 -0.84672
[3,] -0.42336 -0.84672 -1.27008
```

4.2 Richardson Extrapolated Finite Differences

Compute the first and second derivatives of $f : \mathbb{R}^n \to \mathbb{R}$ numerically at point x using Richardson extrapolation. This is typically more accurate than the simple finite differences in 1 and does not require an informed choice of the step size h; however, it requires more evaluations of f. A general discussion of Richardson extrapolation is given in Section 9.6 of Quarteroni, Sacco, and Saleri (2007).

Taking the function $g(h)$ to be one of finite difference approximations in (**??**)–(**??**), $h > 0$ to be an initial step size, and $\delta \in (0, 1)$ to be a reduction factor, the method computes a table of values

$$A_{mq} = \begin{cases} g(\delta^m h), & \text{for } m = 0, \dots, n \text{ and } q = 0, \\ \dfrac{A_{m,q-1} - \delta^q A_{m-1,q-1}}{1 - \delta^q}, & \text{for } m = 0, \dots, n \text{ and } q = 1, \dots, m, \end{cases}$$

given a predetermined n. Note that $n + 1$ evaluations of f are required. The result is taken to be the A_{mq} such that

$$e_{mq} = \max\{|A_{mq} - A_{m,q-1}|, |A_{mq} - A_{m-1,q-1}|\}$$

is the smallest, with the corresponding $e = e_{mq}$ reported as the achieved tolerance. Furthermore, because numerical error may begin to worsen as the table is iteratively computed, the method halts if it encounters the condition

$$|A_{mq} - A_{m-1,q-1}| > \tau e,$$

with e as the smallest e_{mq} computed so far and $\tau > 1$ is a given multiplier. These criteria for convergence and stopping early due to reduced numerical precision are suggested in Section 5.7 of Press et al. (2007); similar considerations for halting criteria are considered in the Julia package Richardson.jl.

Taking $n = 0$ produces finite differences described in 1 using the initial step size h as the perturbation. Here the achieved tolerance is reported as infinite.

Function

Primary location of source code are the files inst/include/deriv.h, inst/include/deriv2.h, and inst/include/richardson.h.

```
richardson_result deriv(
    const dfv& f,                                    ①
    const Rcpp::NumericVector& x,                    ②
    unsigned int i,                                  ③
```

```
    const richardson_args& args,                                    ⑤
    const fd_types& fd_type = fd_types::SYMMETRIC                   ⑥
)

richardson_result deriv(
    const dfv& f,                                                   ①
    const Rcpp::NumericVector& x,                                   ②
    unsigned int i,                                                 ③
    const fd_types& fd_type = fd_types::SYMMETRIC                   ⑥
)

richardson_result deriv2(
    const dfv& f,                                                   ①
    const Rcpp::NumericVector& x,                                   ②
    unsigned int i,                                                 ③
    unsigned int j,                                                 ④
    const richardson_args& args,                                    ⑤
    const fd_types& fd_type = fd_types::SYMMETRIC                   ⑥
)

richardson_result deriv2(
    const dfv& f,                                                   ①
    const Rcpp::NumericVector& x,                                   ②
    unsigned int i,                                                 ③
    unsigned int j,                                                 ④
    const fd_types& fd_type = fd_types::SYMMETRIC                   ⑥
)
```

① Function d to differentiate.
② Point x at which derivative is taken.
③ Optional arguments.
④ First coordinate to differentiate.
⑤ Second coordinate to differentiate.
⑥ Type of finite difference.

The functions `deriv` and `deriv2` compute first and second derivatives, respectively. The enum class `fd_type` is described in 1.

Optional Arguments

```
struct richardson_args
{
    double delta = 0.5;                                             ①
    unsigned int maxiter = 10;                                      ②
    double h = 1;                                                   ③
    double tol = mach_eps_4r;                                       ④
```

```
    double accuracy_factor = R_PosInf;                    ⑤

    richardson_args() { };                                ⑥
    richardson_args(SEXP obj);                            ⑦
    operator SEXP() const;                                ⑧
};
```

① The factor δ used to reduce h.
② Maximum number of iterations.
③ The initial value of h.
④ Tolerance for convergence.
⑤ The factor τ used to check for loss of precision. The infinite default value disables the check.
⑥ Default constructor.
⑦ Constructor from an `Rcpp::List`.
⑧ Conversion operator to `Rcpp::List`.

Result

```
struct richardson_result
{
    double value;                                         ①
    double err;                                           ②
    unsigned int iter;                                    ③
    richardson_status status;                             ④

    operator SEXP() const;                                ⑤
};
```

① The final approximation of the derivative.
② An estimate of the error in approximation.
③ Number of iterations m used to produce the approximation.
④ A code describing the status of the operation.
⑤ Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `fd_deriv_result` as an `Rcpp::List`.

Name	Type	Description
`value`	`Rcpp::NumericVector`	A scalar based on field `value`.
`err`	`Rcpp::NumericVector`	A scalar based on field `err`.
`iter`	`Rcpp::IntegerVector`	A scalar based on field `iter`.
`status`	`Rcpp::IntegerVector`	A scalar based on field `status`.

Status Codes

```
enum class richardson_status : unsigned int {
    OK = 0L,                                                    ①
    NOT_CONVERGED = 1L,                                         ②
    NUMERICAL_PRECISION = 2L                                    ③
};
```

① OK.
② Not converged within `maxiter` iterations.
③ Early termination due to numerical precision.

Example

Compute first and second derivatives of $f(x) = \sin(b^\top x)$ at $x_0 = (1/2, \dots, 1/2)$ with $b = (1, \dots, n)$. This is a continuation of the example in 1. Let us again define the point x_0.

```
n = 3
x0 = rep(0.5, n)
```

To demonstrate the numerical first derivative, a C++ function with Rcpp interface is defined in the file `examples/deriv.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List deriv_ex(Rcpp::NumericVector x0, unsigned int i, unsigned int type)
{
    fntl::dfv f = [](Rcpp::NumericVector x) {
        double ss = 0;
        for (unsigned int i = 0; i < x.length(); i++) { ss += (i+1) * x(i); }
        return std::sin(ss);
    };

    auto out = fntl::deriv(f, x0, i, fntl::fd_types(type));
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/deriv.cpp")
out3 = out2 = out1 = numeric(n)
for (i in 1:n) {
    out1[i] = deriv_ex(x0, i-1, type = 0)$value  ## Symmetric
    out2[i] = deriv_ex(x0, i-1, type = 1)$value  ## Forward
    out3[i] = deriv_ex(x0, i-1, type = 2)$value  ## Backward
}
```

```
print(out1)
```

```
[1] -0.9899772 -1.9799546 -2.9699519
```

```
print(out2)
```

```
[1] -0.9900957 -1.9987848 -2.9704411
```

```
print(out3)
```

```
[1] -0.9898889 -1.9797778 -2.9695107
```

To demonstrate the numerical second derivative, a C++ function with Rcpp interface is defined in the file `examples/deriv2.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List deriv2_ex(Rcpp::NumericVector x0, unsigned int i, unsigned int j,
    unsigned int type)
{
    fntl::dfv f = [](Rcpp::NumericVector x) {
        double ss = 0;
        for (unsigned int i = 0; i < x.length(); i++) { ss += (i+1) * x(i); }
        return std::sin(ss);
    };

    auto out = fntl::deriv2(f, x0, i, j, fntl::fd_types(type));
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/deriv2.cpp")
out3 = out2 = out1 = matrix(NA, n, n)
for (i in 1:n) {
for (j in 1:n) {
    out1[i,j] = deriv2_ex(x0, i-1, j-1, type = 0)$value  ## Symmetric
    out2[i,j] = deriv2_ex(x0, i-1, j-1, type = 1)$value  ## Forward
    out3[i,j] = deriv2_ex(x0, i-1, j-1, type = 2)$value  ## Backward
}
```

```
}
print(out1)
```

```
           [,1]       [,2]       [,3]
[1,] -0.1411024 -0.2822182 -0.4233438
[2,] -0.2822182 -0.5644627 -0.8467095
[3,] -0.4233438 -0.8467095 -1.2700582
```

```
print(out2)
```

```
           [,1]       [,2]       [,3]
[1,] -0.1403947 -0.2800641 -0.4190080
[2,] -0.2800641 -0.5586774 -0.8358399
[3,] -0.4190080 -0.8358399 -1.2504954
```

```
print(out3)
```

```
           [,1]       [,2]       [,3]
[1,] -0.1418452 -0.2844157 -0.4277113
[2,] -0.2844157 -0.5702817 -0.8575980
[3,] -0.4277113 -0.8575980 -1.2896600
```

4.3 Gradient

The gradient of $f : \mathbb{R}^n \to \mathbb{R}$:

$$\frac{\partial f(x)}{\partial x} = \left[\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_n}\right].$$

Each coordinate is computed via `deriv` in Section 4.2.

Function

Primary location of source code is the file inst/include/gradient.h.

```
gradient_result gradient(
    const dfv& f,                                                    ①
    const Rcpp::NumericVector& x,                                    ②
    const richardson_args& args,                                     ③
    const fd_types& fd_type = fd_types::SYMMETRIC                    ④
)

gradient_result gradient(
    const dfv& f,                                                    ①
```

```
    const Rcpp::NumericVector& x,                                    ②
    const fd_types& fd_type = fd_types::SYMMETRIC                    ④
)
```

① Function to take the gradient of.
② Point at which to compute the gradient.
③ Optional arguments.
④ Type of finite difference to use. See the definition of `fd_types` in

The arguments in `args` are applied to each coordinate of the gradient.

Result

```
struct gradient_result {
    std::vector<double> value;                                       ①
    std::vector<double> err;                                         ②
    std::vector<unsigned int> iter;                                  ③

    operator SEXP() const;                                           ④
};
```

① The final approximation of the gradient.
② The respective approximation errors from `deriv` for each coordinate.
③ The respective iterations taken in `deriv` for each coordinate.
④ Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `gradient_result` as an `Rcpp::List`. The fields here directly correspond to those in `gradient_result`.

Name	Type	Description
`value`	`Rcpp::NumericVector`	Length n
`err`	`Rcpp::NumericVector`	Length n
`iter`	`Rcpp::IntegerVector`	Length n

Example

Compute the gradient of $f(x) = x^\top x$. A C++ function with Rcpp interface is defined in the file `examples/gradient.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List gradient_ex(Rcpp::NumericVector x0)
{
    const fntl::dfv& f = [](Rcpp::NumericVector x) {
```

```
        double out = Rcpp::sum(Rcpp::pow(x, 2));
        return out;
    };

    auto out = fntl::gradient(f, x0);
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/gradient.cpp")
gradient_ex(x0 = 1:4)
```

```
$value
[1] 2 4 6 8

$err
[1] 0 0 0 0

$iter
[1] 1 1 1 1
```

Compare to `grad` from the `numDeriv` package (Gilbert and Varadhan 2019).

```
f = function(x) { sum(x^2) }
numDeriv::grad(f, x = 1:4)
```

```
[1] 2 4 6 8
```

4.4 Jacobian

The Jacobian of $f : \mathbb{R}^n \to \mathbb{R}^m$:

$$\frac{\partial f(x)}{\partial x} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \cdots & \frac{\partial f_1(x)}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m(x)}{\partial x_1} & \cdots & \frac{\partial f_m(x)}{\partial x_n} \end{bmatrix}.$$

Each coordinate is computed via `deriv` in Section 4.2.

Function

Source code is in the file inst/include/jacobian.h.

```
jacobian_result jacobian(
    const vfv& f,                                                    ①
    const Rcpp::NumericVector& x,                                    ②
    const richardson_args& args                                      ③
    const fd_types& fd_type = fd_types::SYMMETRIC                    ④
)

jacobian_result jacobian(
    const vfv& f,                                                    ①
    const Rcpp::NumericVector& x,                                    ②
    const fd_types& fd_type = fd_types::SYMMETRIC                    ④
)
```

① Function to obtain the Jacobian of
② Point at which to take the Jacobian.
③ Optional arguments.
④ Type of finite difference to use. See the definition of `fd_types` in

Result

```
struct jacobian_result
{
    std::vector<double> value;                                       ①
    std::vector<double> err;                                         ②
    std::vector<unsigned int> iter;                                  ③
    double rows;                                                     ④
    double cols;                                                     ⑤

    operator SEXP() const;                                           ⑥
};
```

① The final approximation of the Jacobian stored as a vector in row-major format.
② The respective approximation errors from `deriv` for each coordinate of `value`.
③ The respective iterations taken in `deriv` for each coordinate of `value`.
④ The row dimension m of the Jacobian.
⑤ The column dimension n of the Jacobian.
⑥ Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `jacobian_result` as an `Rcpp::List`.

Name	Type	Description
`value`	`Rcpp::NumericMatrix`	An $m \times n$ matrix based on field `value`.
`err`	`Rcpp::NumericMatrix`	An $m \times n$ matrix based on field `err`.
`iter`	`Rcpp::IntegerMatrix`	An $m \times n$ matrix based on field `iter`.

Example

Compute the Jacobian of $f(x) = [f_1(x), \dots, f_5(x)]$, $f_i(x) = \sum_{j=1}^{i} \sin(x_j)$, at the point $x = (1, 2, 3, 4, 5)$. To obtain the resulting value as an $m \times n$ matrix, we apply the List conversion operator to the result. A C++ function with Rcpp interface is defined in the file `examples/jacobian.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List jacobian_ex(Rcpp::NumericVector x0)
{
    fntl::vfv f = [](Rcpp::NumericVector x) {
        Rcpp::NumericVector out = Rcpp::cumsum(Rcpp::sin(x));
        return out;
    };

    auto out = fntl::jacobian(f, x0);
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/jacobian.cpp")
out = jacobian_ex(x0 = 1:4)
names(out)
```

```
[1] "value" "err"   "iter"
```

```
print(out$value)
```

```
          [,1]       [,2]       [,3]       [,4]
[1,] 0.5402685  0.0000000  0.0000000  0.0000000
[2,] 0.5402685 -0.4161208  0.0000000  0.0000000
[3,] 0.5402685 -0.4161208 -0.9899772  0.0000000
[4,] 0.5402685 -0.4161208 -0.9899772 -0.6536335
```

Compare to `jacobian` from the `numDeriv` package (Gilbert and Varadhan 2019).

```
f = function(x) { cumsum(sin(x)) }
numDeriv::jacobian(f, x = 1:4)
```

```
          [,1]       [,2]       [,3]       [,4]
[1,] 0.5403023  0.0000000  0.0000000  0.0000000
[2,] 0.5403023 -0.4161468  0.0000000  0.0000000
[3,] 0.5403023 -0.4161468 -0.9899925  0.0000000
[4,] 0.5403023 -0.4161468 -0.9899925 -0.6536436
```

4.5 Hessian

Compute the Hessian of $f : \mathbb{R}^n \to \mathbb{R}$ numerically at point x:

$$\frac{\partial^2 f(x)}{\partial x \partial x^\top} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f(x)}{\partial x_n \partial x_n} \end{bmatrix}.$$

Each coordinate is computed via `deriv2` in Section 4.2.

Note that there is a C function fdhess within R which is based on Algorithm A5.6.2 of Dennis and Schnabel (1983). However, it is not included in the API for external use (R Core Team 2024b, sec. 6).

Function

Primary location of source code is the file inst/include/hessian.h.

```
hessian_result hessian(
    const dfv& f,                                                    ①
    const Rcpp::NumericVector& x,                                    ②
    const richardson_args& args,                                     ③
    const fd_types& fd_type = fd_types::SYMMETRIC                    ④
)

hessian_result hessian(
    const dfv& f,                                                    ①
    const Rcpp::NumericVector& x,                                    ②
    const fd_types& fd_type = fd_types::SYMMETRIC                    ④
)
```

① Function f for which Hessian is to be computed.
② Point x at which Hessian is taken.
③ Optional arguments.
④ Type of finite difference to use. See the definition of `fd_types` in

Result

The result is an $n \times n$ Hessian matrix.

```
struct hessian_result
{
    std::vector<double> value;                                    ①
    std::vector<double> err;                                      ②
    std::vector<unsigned int> iter;                               ③
    double dim;                                                   ④

    operator SEXP() const;                                        ⑤
};
```

① The value of the Hessian: a vector containing the lower-triangular in column-major order.
② The respective approximation errors from `deriv2` for each coordinate.
③ The respective iterations taken in `deriv2` for each coordinate.
④ The row and column dimension.
⑤ Conversion operator to `Rcpp::List`.

Example

Compute the Hessian of $f(x) = \sum_{i=1}^{n} \sin(x_i)$ at $x = (1, 2)$. To obtain the resulting value as an $n \times n$ matrix, we apply the List conversion operator to the result. A C++ function with Rcpp interface is defined in the file `examples/hessian.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List hessian_ex(Rcpp::NumericVector x0)
{
    fntl::dfv f = [](Rcpp::NumericVector x) {
        double out = Rcpp::sum(Rcpp::sin(x));
        return out;
    };

    auto out = fntl::hessian(f, x0);
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/hessian.cpp")
out = hessian_ex(x0 = c(1,2))
print(out)
```

```
$value
              [,1]          [,2]
[1,] -8.414450e-01  2.577303e-16
```

```
[2,]  2.577303e-16 -9.092693e-01

$err
             [,1]         [,2]
[1,] 7.918737e-05 2.854859e-16
[2,] 2.854859e-16 8.557024e-05

$iter
     [,1] [,2]
[1,]    6    1
[2,]    1    6
```

Compare to `hessian` from the `numDeriv` package (Gilbert and Varadhan 2019).

```
f = function(x) { sum(sin(x)) }
numDeriv::hessian(f, x = c(1,2))
```

```
             [,1]          [,2]
[1,] -8.414710e-01  5.137422e-13
[2,]  5.137422e-13 -9.092974e-01
```

5 Root-Finding

Find a root of $f : \mathbb{R} \to \mathbb{R}$, if present, on the interval $[a, b]$; i.e., find $x \in [a, b]$ such that $f(x) \approx 0$. The R function uniroot has its implementation in an underlying C function zeroin2 which appears not to be readily exported for external use. We therefore provide several alternatives in C++.

There are currently two implementations. The bisection method (e.g., Press et al. 2007, sec. 9.1) is simpler while Brent's method is faster and is used in `uniroot`. This implementation of Brent's method is based on the ALGOL code in Section 4.6 of Brent (1973). The two functions use a common arguments struct, results struct, and status codes, which are given below.

Optional Arguments

```
struct findroot_args {
    double tol = mach_eps_4r;                          ①
    unsigned int maxiter = 1000;                       ②
    error_action action = error_action::STOP;          ③

    findroot_args() { };                               ④
    findroot_args(SEXP obj);                           ⑤
    operator SEXP() const;                             ⑥
};
```

① Tolerance for convergence.
② Maximum number of iterations.
③ Action to take if operation returns a status code other than `OK`.
④ Default constructor.
⑤ Constructor from an `Rcpp::List`.
⑥ Conversion operator to `Rcpp::List`.

Result

```
struct findroot_result {
    double root;                                                    ①
    double f_root;                                                  ②
    unsigned int iter;                                              ③
    double tol;                                                     ④
    findroot_status status;                                         ⑤
    std::string message;                                            ⑥

    operator SEXP() const;                                          ⑦
};
```

① The final approximation for the root.
② The value of f at the root.
③ The number of iterations.
④ An estimate of the error in approximation.
⑤ A code describing the status of the operation.
⑥ A message describing the status of the operation.
⑦ Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `findroot_result` as an `Rcpp::List`. The fields here directly correspond to those in `findroot_result`.

Name	Type	Description
`root`	`Rcpp::NumericVector`	Length 1
`f_root`	`Rcpp::NumericVector`	Length 1
`iter`	`Rcpp::IntegerVector`	Length 1
`tol`	`Rcpp::NumericVector`	Length 1
`status`	`Rcpp::IntegerVector`	Length 1
`message`	`Rcpp::StringVector`	Length 1

Status Codes

```
enum class findroot_status : unsigned int {
    OK = 0L,                                                        ①
    NUMERICAL_OVERFLOW = 1L,                                        ②
    NOT_CONVERGED = 2L                                              ③
```

```
};
```

① OK.
② Numerical overflow: tol may be too small.
③ Not converged within `maxiter` iterations.

5.1 Bisection

This algorithm successively shrinks the starting interval and terminates when the absolute difference is smaller than a tolerance ϵ.

Function

Primary location of source code is the file inst/include/findroot-bisect.h.

```
findroot_result findroot_bisect(
    const dfd& f,                                    ①
    double lower,                                    ②
    double upper,                                    ③
    const findroot_args& args                        ④
)

findroot_result findroot_bisect(
    const dfd& f,                                    ①
    double lower,                                    ②
    double upper                                     ③
)
```

① Function f for which a root is desired.
② Lower limit a of search interval. Must be finite.
③ Upper limit b of search interval. Must be finite.
④ Additional arguments.

Example

Find the root of the function $f(x) = x^2 - 1$ on $[0, 10]$. A C++ function with Rcpp interface is defined in the file `examples/findroot-bisect.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List findroot_bisect_ex(double lower, double upper)
{
    fntl::dfd f = [](double x) {
        return pow(x, 2) - 1;
```

```
    };
    auto out = fntl::findroot_bisect(f, lower, upper);
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/findroot-bisect.cpp")
out = findroot_bisect_ex(0, 10)
print(out)
```

```
$root
[1] 1.000023

$f_root
[1] 4.577689e-05

$iter
[1] 17

$tol
[1] 0.0001220703

$status
[1] 0

$message
[1] "OK"
```

5.2 Brent's Algorithm

The algorithm successively shrinks the starting interval and terminates when the absolute difference is smaller than a tolerance ϵ.

Function

Primary location of source code is the file inst/include/findroot-brent.h.

```
findroot_result findroot_brent(
    const dfd& f,                                    ①
    double lower,                                    ②
    double upper,                                    ③
    const findroot_args& args                        ④
)
```

```
findroot_result findroot_brent(
    const dfd& f,                                                    ①
    double lower,                                                    ②
    double upper                                                     ③
)
```

① Function f for which a root is desired.
② Lower limit a of search space.
③ Upper limit b of search space.
④ Additional arguments.

Example

Find the root of the function $f(x) = x^2 - 1$ on $[0, 10]$. A C++ function with Rcpp interface is defined in the file `examples/findroot-brent.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List findroot_brent_ex(double lower, double upper)
{
    fntl::dfd f = [](double x) { return pow(x, 2) - 1; };
    auto out = fntl::findroot_brent(f, lower, upper);
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/findroot-brent.cpp")
out = findroot_brent_ex(0, 10)
print(out)
```

```
$root
[1] 1.000025

$f_root
[1] 4.93727e-05

$iter
[1] 10

$tol
[1] -6.103516e-05

$status
```

```
[1] 0

$message
[1] "OK"
```

6 Univariate Optimization

Minimize the function $f : \mathbb{R} \to \mathbb{R}$ on a given interval $[a, b]$. The R optimize function is based on an underlying Brent_fmin C implementation of Brent's algorithm (Brent 1973, sec. 5). However, this function appears not to be exported from R for external use.

We provide several alternatives in C++: The golden section search method (e.g., Press et al. 2007, sec. 10.2) is simple, guaranteed to converge, and does not require information about derivatives. Brent's algorithm is more involved but converges faster. The implementation of Brent's algorithm in C++ based is on the ALGOL code in Section 5.8 of Brent (1973). The golden section and Brent functions use a common arguments struct, results struct, and status codes, which are given below.

Optional Arguments

```
struct optimize_args
{
    bool fnscale = 1;                                          ①
    double tol = mach_eps_2r;                                  ②
    unsigned int maxiter = 1000;                               ③
    unsigned int report_period = uint_max;                     ④
    error_action action = error_action::STOP;                  ⑤

    optimize_args() { };                                       ⑥
    optimize_args(SEXP obj);                                   ⑦
    operator SEXP() const;                                     ⑧
};
```

① Scaling factor to be applied to the value of $f(x)$ during optimization. Use `-1` to implement maximization rather than minimization.
② Tolerance ϵ for convergence.
③ The maximum number of iterations.
④ The frequency of reports.
⑤ Action to take if operation returns a status code other than `OK`.
⑥ Default constructor.
⑦ Constructor from an `Rcpp::List`.
⑧ Conversion operator to `Rcpp::List`.

Result

```
struct optimize_result
{
    double par;                                                    ①
    double value;                                                  ②
    unsigned int iter;                                             ③
    double tol;                                                    ④
    optimize_status status;                                        ⑤
    std::string message;                                           ⑥

    operator SEXP() const;                                         ⑦
};
```

① The final value of the optimization variable.
② The value of the function corresponding to `par`.
③ The number of iterations taken.
④ The achieved tolerance δ.
⑤ Status code from the optimizer.
⑥ A message describing the status of the operation.
⑦ Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `optimize_result` as an `Rcpp::List`. The fields here directly correspond to those in `optimize_result`.

Name	Type	Description
`par`	`Rcpp::NumericVector`	Length 1
`value`	`Rcpp::NumericVector`	Length 1
`iter`	`Rcpp::IntegerVector`	Length 1
`tol`	`Rcpp::NumericVector`	Length 1
`status`	`Rcpp::IntegerVector`	Length 1
`message`	`Rcpp::StringVector`	Length 1

Status Codes

```
enum class optimize_status : unsigned int {
    OK = 0L,                                                       ①
    NUMERICAL_OVERFLOW = 1L,                                       ②
    NOT_CONVERGED = 2L                                             ③
};
```

① OK.
② Numerical overflow: tol may be too small.
③ iteration limit had been reached.

6.1 Golden Section Search

The algorithm successively shrinks the starting interval and terminates when $\delta = b - a$ is smaller than a given tolerance ϵ.

Function

Primary location of source code is the file inst/include/goldensection.h.

```
optimize_result goldensection(
    const dfd& f,                                    ①
    double lower,                                    ②
    double upper,                                    ③
    const optimize_args& args                        ④
)

optimize_result goldensection(
    const dfd& f,                                    ①
    double lower,                                    ②
    double upper                                     ③
)
```

① Function f to minimize.
② Lower limit a of search space.
③ Upper limit b of search space.
④ Additional arguments.

Example

Maximize the function $f(x) = \exp(-x)$ on $[0, 1]$. A C++ function with Rcpp interface is defined in the file `examples/goldensection.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List goldensection_ex(double lower, double upper)
{
    fntl::optimize_args args;
    args.fnscale = -1;

    fntl::dfd f = [](double x) { return exp(-x); };
    auto out = fntl::goldensection(f, lower, upper, args);
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/goldensection.cpp")
out = goldensection_ex(0, 1)
print(out)
```

```
$par
[1] 5.720575e-09

$value
[1] -1

$iter
[1] 38

$tol
[1] 1.144115e-08

$status
[1] 0

$message
[1] "OK"
```

6.2 Brent's Algorithm

This algorithm successively shrinks the starting interval and terminates when $\delta = |x - m|$ is smaller than a tolerance ϵ, where $m = (a + b)/2$ is the midpoint of the current $[a, b]$.

Function

Primary location of source code is the file inst/include/optimize-brent.h.

```
optimize_result optimize_brent(
    const dfd& f,                                    ①
    double lower,                                    ②
    double upper,                                    ③
    const optimize_args& args                        ④
)

optimize_result optimize_brent(
    const dfd& f,                                    ①
    double lower,                                    ②
    double upper                                     ③
)
```

① Function f to minimize.

② Lower limit a of search space.
③ Upper limit b of search space.
④ Additional arguments.

Example

Maximize the function $f(x) = \exp(-x)$ on $[0, 1]$. A C++ function with Rcpp interface is defined in the file `examples/optimize-brent.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List optimize_brent_ex(double lower, double upper)
{
    fntl::optimize_args args;
    args.fnscale = -1;

    fntl::dfd f = [](double x) { return exp(-x); };
    auto out = fntl::optimize_brent(f, lower, upper, args);
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/optimize-brent.cpp")
out = optimize_brent_ex(0, 1)
print(out)
```

```
$par
[1] 1.505216e-08

$value
[1] -1

$iter
[1] 37

$tol
[1] 7.549828e-11

$status
[1] 0

$message
[1] "OK"
```

7 Multivariate Optimization

This section presents methods to minimize a function $f : \mathbb{R}^n \to \mathbb{R}$ from a given starting value x_0.

7.1 Nelder-Mead

Relies only on evaluation of f and does not use the gradient or Hessian (Nelder and Mead 1965). This function directly calls nmmin, the C function that gets invoked when using the optim R function with `method = "Nelder-Mead"`.

Function

Primary location of source code is the file inst/include/neldermead.h.

```
neldermead_result neldermead(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const neldermead_args& args                         ③
)

neldermead_result neldermead(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f                                        ②
)
```

① Initial value for optimization variable.
② Function f to minimize.
③ Additional arguments.

Optional Arguments

```
struct neldermead_args
{
    double alpha = 1.0;                                 ①
    double beta = 0.5;                                  ②
    double gamma = 2.0;                                 ③
    unsigned int trace = 0;                             ④
    double abstol = R_NegInf;                           ⑤
    double reltol = mach_eps_2r;                        ⑥
    unsigned int maxit = 500;                           ⑦
    double fnscale = 1.0;                               ⑧

    neldermead_args() { };                              ⑨
    neldermead_args(SEXP obj);                          ⑩
    operator SEXP() const;                              ⑪
};
```

① Reflection factor.
② Contraction and reduction factor.
③ Extension factor.
④ If positive, print progress info.
⑤ Absolute tolerance.
⑥ User-initialized conversion tolerance.
⑦ Maximum number of iterations.
⑧ Scaling factor to be applied to the value of $f(x)$ during optimization. Use `-1` to implement maximization rather than minimization.
⑨ Default constructor.
⑩ Constructor from an `Rcpp::List`.
⑪ Conversion operator to `Rcpp::List`.

Result

```
struct neldermead_result {
    std::vector<double> par;                                    ①
    double value;                                               ②
    neldermead_status status;                                   ③
    int fncount;                                                ④

    operator SEXP() const;                                      ⑤
};
```

① The final approximation of the optimizer.
② The value of f at the optimizer.
③ A code describing the status of the operation.
④ The number of function evaluations.
⑤ Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `neldermead_result` as an `Rcpp::List`. The fields here directly correspond to those in `neldermead_result`.

Name	Type	Description
`par`	`Rcpp::NumericVector`	Length n
`value`	`Rcpp::NumericVector`	Length 1
`status`	`Rcpp::IntegerVector`	Length 1
`fncount`	`Rcpp::IntegerVector`	Length 1

Status Codes

```
enum class neldermead_status : unsigned int {
    OK = 0L,                                                    ①
    NOT_CONVERGED = 1L,                                         ②
    SIMLEX_DEGENERACY = 10L                                     ③
```

```
};
```

① OK.
② Not converged within `maxiter` iterations.
③ Degeneracy of the Nelder–Mead simplex.

Example

Minimize the function $f(x) = \sum_{i=1}^{n} x_i^2 - 1$ for $n = 2$. Take $x_0 = (1, -1)$ as the initial value. A C++ function with Rcpp interface is defined in the file `examples/neldermead.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List neldermead_ex(Rcpp::NumericVector x0)
{
    fntl::dfv f = [](Rcpp::NumericVector x) {
                double out = Rcpp::sum(Rcpp::pow(x, 2)) - 1;
        return out;
    };

    auto out = fntl::neldermead(x0, f);
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/neldermead.cpp")
out = neldermead_ex(x0 = c(1, -1))
print(out)
```

```
$par
[1] -0.0001503306  0.0001134426

$value
[1] -1

$fncount
[1] 55

$status
[1] 0
```

7.2 BFGS

Minimization using the BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm (Nash 1990). Relies on evaluation of f and its gradient $g(x) = \frac{\partial f(x)}{\partial x}$ but not the Hessian. This function directly calls vmmin, the C function that gets invoked when using the optim R function with `method = "BFGS"`.

Function

Primary location of source code is the file inst/include/bfgs.h.

```
bfgs_result bfgs(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const vfv& g,                                       ③
    const bfgs_args& args                               ④
)

bfgs_result bfgs(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const bfgs_args& args                               ④
)

bfgs_result bfgs(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const vfv& g                                        ③
)

bfgs_result bfgs(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f                                        ②
)
```

① Initial value for optimization variable.
② Function f to minimize.
③ Gradient function $g(x) = \frac{\partial f(x)}{\partial x}$.
④ Additional arguments.

Forms with the g argument omitted compute the gradient using finite differences, via the `gradient` method in Section 4.3.

Optional Arguments

```
struct bfgs_args {
    richardson_args deriv_args;                         ①
```

```
    double parscale = 1;                                    ②
    int trace = 0;                                          ③
    double fnscale = 1;                                     ④
    int maxit = 100;                                        ⑤
    int report = 10;                                        ⑥
    double abstol = R_NegInf;                               ⑦
    double reltol = mach_eps_2r;                            ⑧

    bfgs_args() { };                                        ⑨
    bfgs_args(SEXP obj);                                    ⑩
    operator SEXP() const;                                  ⑪
};
```

① Arguments for Richardson extrapolated numerical derivatives to compute the gradient if g is omitted in the call to `bfgs`. See Section 4.2.
② A vector of scaling values for the parameters. (Currently not used).
③ If positive, tracing information on the progress of the optimization is produced. There are six levels which give progressively more detail.
④ Scaling factor applied to the value of `f` and `g` during optimization.
⑤ The maximum number of iterations.
⑥ The frequency of reports.
⑦ Absolute tolerance.
⑧ Relative tolerance.
⑨ Default constructor.
⑩ Constructor from an `Rcpp::List`.
⑪ Conversion operator to `Rcpp::List`.

Result

```
struct bfgs_result {
    std::vector<double> par;                                ①
    double value;                                           ②
    bfgs_status status;                                     ③
    int fncount;                                            ④
    int grcount;                                            ⑤

    operator SEXP() const;                                  ⑥
};
```

① The final value of the optimization variable.
② The value of the function corresponding to `par`.
③ Status code from the optimizer.
④ Number of times the objective function was called.
⑤ Number of times the gradient function was called.
⑥ Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `bfgs_result` as an `Rcpp::List`. The fields here directly correspond to those in `bfgs_result`.

Name	Type	Description
`par`	`Rcpp::NumericVector`	Length n
`value`	`Rcpp::NumericVector`	Length 1
`status`	`Rcpp::IntegerVector`	Length 1
`fncount`	`Rcpp::IntegerVector`	Length 1
`grcount`	`Rcpp::IntegerVector`	Length 1

Status Codes

```
enum class bfgs_status : unsigned int {
    OK = 0L,                                                    ①
    NOT_CONVERGED = 1L                                          ②
};
```

① OK.
② iteration limit maxit had been reached.

Example

Maximize the function $f(x) = \exp(-x^\top x)$. First use the default numerical gradient, then use an explicitly coded the gradient function $g(x) = -2x\exp(-x^\top x)$. A C++ function with Rcpp interface is defined in the file `examples/bfgs.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List bfgs_ex(Rcpp::NumericVector x0)
{
    fntl::dfv f = [](const Rcpp::NumericVector& x) {
        Rcpp::NumericVector xx = Rcpp::pow(x, 2);
        double ss = Rcpp::sum(xx);
        return std::exp(-ss);
    };

    fntl::vfv g = [](const Rcpp::NumericVector& x) {
        Rcpp::NumericVector xx = Rcpp::pow(x, 2);
        double ss = Rcpp::sum(xx);
        return -2 * std::exp(-ss) * x;
    };

    fntl::bfgs_args args;
```

```
    args.fnscale = -1;

    auto out1 = fntl::bfgs(x0, f, args);     // with default numerical gradient
    auto out2 = fntl::bfgs(x0, f, g, args);  // with explicitly coded gradient

    return Rcpp::List::create(
        Rcpp::Named("numerical") = Rcpp::wrap(out1),
        Rcpp::Named("analytical") = Rcpp::wrap(out2)
    );
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/bfgs.cpp")
out = bfgs_ex(x0 = rep(1, 4))
print(out$numerical)
```

```
$par
[1] -1.168106e-07 -1.168106e-07 -1.168106e-07 -1.168106e-07

$value
[1] 1

$fncount
[1] 15

$grcount
[1] 11

$status
[1] 0
```

```
print(out$analytical)
```

```
$par
[1] -2.217083e-09 -2.217083e-09 -2.217083e-09 -2.217083e-09

$value
[1] 1

$fncount
[1] 18

$grcount
```

```
[1] 11

$status
[1] 0
```

7.3 L-BFGS-B

Minimization using the L-BFGS-B (Limited memory Broyden-Fletcher-Goldfarb-Shanno) algorithm (Byrd et al. 1995). Relies on evaluation of f and its gradient $g(x) = \frac{\partial f(x)}{\partial x}$ but not the Hessian. This function directly calls lbfgsb, the C function that gets invoked when using the optim R function with `method = "L-BFGS-B"`.

Function

Primary location of source code is the file inst/include/lbfgsb.h.

```
lbfgsb_result lbfgsb(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const vfv& g,                                       ③
    const lbfgsb_args& args                             ④
)

lbfgsb_result lbfgsb(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const lbfgsb_args& args                             ④
)

lbfgsb_result lbfgsb(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const vfv& g                                        ③
)

lbfgsb_result lbfgsb(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f                                        ②
)
```

① Initial value for optimization variable.
② Function f to minimize.
③ Gradient function $g(x) = \frac{\partial f(x)}{\partial x}$.
④ Additional arguments.

Forms with the g argument omitted compute the gradient using finite differences, via the `gradient` method in Section 4.3.

Optional Arguments

```
struct lbfgsb_args {
    std::vector<double> lower;                                    (1)
    std::vector<double> upper;                                    (2)
    richardson_args deriv_args;                                   (3)
    double parscale = 1;                                          (4)
    int trace = 0;                                                (5)
    double fnscale = 1;                                           (6)
    int lmm = 5;                                                  (7)
    int maxit = 100;                                              (8)
    int report = 10;                                              (9)
    double factr = 1e7;                                           (10)
    double pgtol = 0;                                             (11)

    lbfgsb_args() { };                                            (12)
    lbfgsb_args(SEXP obj);                                        (13)
    operator SEXP() const;                                        (14)
};
```

(1) A vector of lower bounds. If left unspecified, will be taken to be a vector of `-Inf` values.
(2) A vector of upper bounds. If left unspecified, will be taken to be a vector of `Inf` values.
(3) Arguments for Richardson extrapolated numerical derivatives to compute the gradient if g is omitted in the call to `lbfgsb`. See Section 4.2.
(4) A vector of scaling values for the parameters. (Currently not used).
(5) If positive, tracing information on the progress of the optimization is produced. There are six levels which give progressively more detail.
(6) Scaling factor applied to the value of `f` and `g` during optimization.
(7) Number of BFGS updates retained.
(8) The maximum number of iterations.
(9) The frequency of reports.
(10) Convergence occurs when the reduction in the objective is within this factor of the machine tolerance.
(11) A tolerance on the projected gradient in the current search direction. The check is suppressed when the value is zero.
(12) Default constructor.
(13) Constructor from an `Rcpp::List`.
(14) Conversion operator to `Rcpp::List`.

Result

```
struct lbfgsb_result {
    std::vector<double> par;                                      (1)
```

```
    double value;                                                    ②
    lbfgsb_status status;                                            ③
    int fncount;                                                     ④
    int grcount;                                                     ⑤
    std::string msg;                                                 ⑥

    operator SEXP() const;                                           ⑦
};
```

① The final value of the optimization variable.
② The value of the function corresponding to `par`.
③ Status code from the optimizer.
④ Number of times the objective function was called.
⑤ Number of times the gradient function was called.
⑥ String with additional information from the optimizer.
⑦ Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `lbfgsb_result` as an `Rcpp::List`. The fields here directly correspond to those in `lbfgsb_result`.

Name	Type	Description
`par`	`Rcpp::NumericVector`	Length n
`value`	`Rcpp::NumericVector`	Length 1
`status`	`Rcpp::IntegerVector`	Length 1
`fncount`	`Rcpp::IntegerVector`	Length 1
`grcount`	`Rcpp::IntegerVector`	Length 1
`message`	`Rcpp::StringVector`	Length 1

Status Codes

```
enum class lbfgsb_status : unsigned int {
    OK = 0L,                                                         ①
    NOT_CONVERGED = 1L,                                              ②
    WARN = 51L,                                                      ③
    ERROR = 52L,                                                     ④
};
```

① OK.
② iteration limit maxit had been reached.
③ algorithm reported a warning.
④ algorithm reported an error.

Example

Maximize the function $f(x) = \exp(-x^\top x)$. First use the default numerical gradient, then use explicitly coded gradient function $g(x) = -2x\exp(-x^\top x)$. A C++ function with Rcpp interface is defined in the file `examples/lbfgsb.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List lbfgsb_ex(Rcpp::NumericVector x0)
{
    fntl::dfv f = [](Rcpp::NumericVector x) {
        double ss = Rcpp::sum(Rcpp::pow(x, 2));
        return std::exp(-ss);
    };

    fntl::vfv g = [](Rcpp::NumericVector x) {
        double ss = Rcpp::sum(Rcpp::pow(x, 2));
            Rcpp::NumericVector out = -2 * std::exp(-ss) * x;
        return out;
    };

    fntl::lbfgsb_args args;
    args.fnscale = -1;

    auto out1 = fntl::lbfgsb(x0, f, args);     // with default numerical gradient
    auto out2 = fntl::lbfgsb(x0, f, g, args);  // with explicitly coded gradient

    return Rcpp::List::create(
        Rcpp::Named("numerical") = Rcpp::wrap(out1),
        Rcpp::Named("analytical") = Rcpp::wrap(out2)
    );
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/lbfgsb.cpp")
out = lbfgsb_ex(x0 = rep(1, 4))
print(out$numerical)
```

```
$par
[1] 2.182178e-06 2.182178e-06 2.182178e-06 2.182178e-06

$value
[1] 1
```

```
$fncount
[1] 10

$grcount
[1] 10

$status
[1] 0

$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

```
print(out$analytical)
```

```
$par
[1] -9.686732e-10 -9.686732e-10 -9.686732e-10 -9.686732e-10

$value
[1] 1

$fncount
[1] 10

$grcount
[1] 10

$status
[1] 0

$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

7.4 Conjugate Gradient

Minimization using the conjugate gradient algorithm (Fletcher and Reeves 1964; Nash 1990). Relies on evaluation of f and its gradient $g(x) = \frac{\partial f(x)}{\partial x}$ but not the Hessian. This function directly calls cgmin, the C function that gets invoked when using the optim R function with `method = "CG"`.

Function

Primary location of source code is the file inst/include/cg.h.

```
cg_result bfgs(
    const Rcpp::NumericVector& init,                                    ①
```

```
    const dfv& f,                                                    ②
    const vfv& g,                                                    ③
    const cg_args& args                                              ④
)

cg_result cg(
    const Rcpp::NumericVector& init,                                 ①
    const dfv& f,                                                    ②
    const cg_args& args                                              ④
)

cg_result cg(
    const Rcpp::NumericVector& init,                                 ①
    const dfv& f,                                                    ②
    const vfv& g                                                     ③
)

cg_result cg(
    const Rcpp::NumericVector& init,                                 ①
    const dfv& f                                                     ②
)
```

① Initial value for optimization variable.
② Function f to minimize.
③ Gradient function $g(x) = \frac{\partial f(x)}{\partial x}$.
④ Additional arguments.

Forms with the g argument omitted compute the gradient using finite differences, via the `gradient` method in Section 4.3.

Optional Arguments

```
struct cg_args
{
    richardson_args deriv_args;                                      ①
    double parscale = 1;                                             ②
    double fnscale = 1;                                              ③
    double abstol = R_NegInf;                                        ④
    double reltol = mach_eps_2r;                                     ⑤
    int type = 1;                                                    ⑥
    int trace = 0;                                                   ⑦
    int maxit = 100;                                                 ⑧

    cg_args() { };                                                   ⑨
    cg_args(SEXP obj);                                               ⑩
```

```
    operator SEXP() const;                                          (11)
};
```

(1) Arguments for Richardson extrapolated numerical derivatives to compute the gradient if g is omitted in the call to `cg`. See Section 4.2.
(2) A vector of scaling values for the parameters. (Currently not used).
(3) Scaling factor applied to the value of `f` and `g` during optimization.
(4) Absolute tolerance.
(5) Relative tolerance.
(6) Type of update: 1 for Fletcher-Reeves, 2 for Polak-Ribiere, and 3 for Beale-Sorenson.
(7) If positive, tracing information on the progress of the optimization is produced. There are six levels which give progressively more detail.
(8) The maximum number of iterations.
(9) Default constructor.
(10) Constructor from an `Rcpp::List`.
(11) Conversion operator to `Rcpp::List`.

Result

```
struct cg_result {
    std::vector<double> par;                                        (1)
    double value;                                                   (2)
    cg_status status;                                               (3)
    int fncount;                                                    (4)
    int grcount;                                                    (5)

    operator SEXP() const;                                          (6)
};
```

(1) The final value of the optimization variable.
(2) The value of the function corresponding to `par`.
(3) Status code from the optimizer.
(4) Number of times the objective function was called.
(5) Number of times the gradient function was called.
(6) Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `cg_result` as an `Rcpp::List`. The fields here directly correspond to those in `cg_result`.

Name	Type	Description
`par`	`Rcpp::NumericVector`	Length n
`value`	`Rcpp::NumericVector`	Length 1
`status`	`Rcpp::IntegerVector`	Length 1
`fncount`	`Rcpp::IntegerVector`	Length 1
`grcount`	`Rcpp::IntegerVector`	Length 1

Status Codes

```
enum class cg_status : unsigned int {
    OK = 0L,                                                    ①
    NOT_CONVERGED = 1L                                          ②
};
```

① OK.
② iteration limit maxit had been reached.

Example

Maximize the function $f(x) = \exp(-x^\top x)$. First use the default numerical gradient, then use an explicitly coded the gradient function $g(x) = -2x \exp(-x^\top x)$. A C++ function with Rcpp interface is defined in the file `examples/cg.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List cg_ex(Rcpp::NumericVector x0)
{
    fntl::dfv f = [](const Rcpp::NumericVector& x) {
        Rcpp::NumericVector xx = Rcpp::pow(x, 2);
        double ss = Rcpp::sum(xx);
        return std::exp(-ss);
    };

    fntl::vfv g = [](const Rcpp::NumericVector& x) {
        Rcpp::NumericVector xx = Rcpp::pow(x, 2);
        double ss = Rcpp::sum(xx);
        return -2 * std::exp(-ss) * x;
    };

    fntl::cg_args args;
    args.fnscale = -1;

    auto out1 = fntl::cg(x0, f, args);     // with default numerical gradient
    auto out2 = fntl::cg(x0, f, g, args);  // with explicitly coded gradient

    return Rcpp::List::create(
        Rcpp::Named("numerical") = Rcpp::wrap(out1),
        Rcpp::Named("analytical") = Rcpp::wrap(out2)
    );
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/cg.cpp")
out = cg_ex(x0 = rep(1, 4))
print(out$numerical)
```

```
$par
[1] 7.027703e-07 7.027703e-07 7.027703e-07 7.027703e-07

$value
[1] 1

$fncount
[1] 22

$grcount
[1] 11

$status
[1] 0
```

```
print(out$analytical)
```

```
$par
[1] -1.600782e-07 -1.600782e-07 -1.600782e-07 -1.600782e-07

$value
[1] 1

$fncount
[1] 25

$grcount
[1] 12

$status
[1] 0
```

7.5 Newton-Type Algorithm for Nonlinear Optimization

Minimization using the Newton-type algorithm underlying the nlm R function. The amounts to calling the C function optif9 within the R API. The implementation of `optif9` is based on Dennis and Schnabel (1983).

The gradient $g(x) = \frac{\partial f(x)}{\partial x}$ and Hessian $h(x) = \frac{\partial^2 f(x)}{\partial x \partial x^\top}$ may be provided explicitly if available. When the gradient and/or Hessian are not specified, numerical approximations are used by `optif9`.

Function

Primary location of source code is the file inst/include/nlm.h.

```
nlm_result nlm(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const vfv& g,                                       ③
    const mfv& h,                                       ④
    const nlm_args& args                                ⑤
)

nlm_result nlm(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const vfv& g,                                       ③
    const nlm_args& args                                ⑤
)

nlm_result nlm(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const nlm_args& args                                ⑤
)

nlm_result nlm(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const vfv& g,                                       ③
    const mfv& h                                        ④
)

nlm_result nlm(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f,                                       ②
    const vfv& g                                        ③
)

nlm_result nlm(
    const Rcpp::NumericVector& init,                    ①
    const dfv& f                                        ②
)
```

① Initial value for optimization variable.
② Function f to minimize.
③ Gradient of f.
④ Hessian of f.
⑤ Additional arguments.

Optional Arguments

```
struct nlm_args
{
    std::vector<double> typsize;                                    ①
    unsigned int print_level = 0;                                   ②
    double fscale = 1;                                              ③
    double fnscale = 1;                                             ④
    unsigned int ndigit = 12;                                       ⑤
    double gradtol = 1e-6;                                          ⑥
    double stepmax = R_PosInf;                                      ⑦
    double steptol = 1e-6;                                          ⑧
    int iterlim = 100;                                              ⑨
    unsigned int method = 1;                                        ⑩
    double trust_radius = 1.0;                                      ⑪

    nlm_args() { };                                                 ⑫
    nlm_args(SEXP obj);                                             ⑬
    operator SEXP() const;                                          ⑭
};
```

① An estimate of the size of each parameter at the minimum.
② Verbosity of messages during optimization:
③ An estimate of the size of f at the minimum.
④ Scaling factor applied to the value of `f`, `g`, and `h` during optimization. Taking this to be `-1` changes the optimization to maximization.
⑤ Number of significant digits in the function f.
⑥ Tolerance to terminate algorithm based on the distance of gradient to zero.
⑦ Maximum allowable scaled step length
⑧ Tolerance to terminate algorithm based on relative step size of successive iterates.
⑨ The maximum number of iterations.
⑩ Algorithm used in optimization:
⑪ Radius of trust region.
⑫ Default constructor.
⑬ Constructor from an `Rcpp::List`.
⑭ Conversion operator to `Rcpp::List`.

Arguments correspond to those in `nlm` with the following exceptions.

- The arguments `method` and `trust_radius` are provided from within `optif9` but not exposed from `nlm`.

- An argument `hessian`is provided in `nlm` to compute the value of the Hessian via finite differences using the final value of the optimization variable. That is not provided here, but may be requested using the Hessian function in Section 4.5.
- The `nlm` function provides an `check.analyticals` argument to check the correctness of provided expressions for the gradient and Hessian. This argument is not provided here, but a similar check can be done using the numerical gradient and Hessian functions in Section 4.3 and Section 4.5, respectively.

See the `nlm` manual page for additional details about other arguments.

If `typsize` is given as the default empty value, it is transformed internally to a vector of n ones to match the default in `nlm`. Similarly, if `stepmax` is given as the default infinity value, it is transformed internally to match the default value in `nlm`.

Result

```
struct nlm_result
{
    std::vector<double> par;                                    ①
    std::vector<double> grad;                                   ②
    double estimate;                                            ③
    int iterations;                                             ④
    nlm_status status;                                          ⑤

    operator SEXP() const;                                      ⑥
};
```

① The final value of the optimization variable.
② The final value of the gradient.
③ The value of the function corresponding to `par`.
④ Number of iterations carried out.
⑤ Status code from the optimizer.
⑥ Conversion operator to `Rcpp::List`.

The `SEXP` conversion operator produces the following representation of `nlm_result` as an `Rcpp::List`. The fields here directly correspond to those in `nlm_result`.

Name	Type	Description
`par`	`Rcpp::NumericVector`	Length n
`grad`	`Rcpp::NumericVector`	Length n
`estimate`	`Rcpp::NumericVector`	Length 1
`iterations`	`Rcpp::IntegerVector`	Length 1
`status`	`Rcpp::IntegerVector`	Length 1
`hessian`	`Rcpp::IntegerVector`	Length n^2

Status Codes

```
enum class nlm_status : unsigned int {
    OK = 0L,                                                    ①
    GRADIENT_WITHIN_TOL = 1L,                                   ②
    ITERATES_WITH_TOL = 2L,                                     ③
    NO_LOWER_STEP = 3L,                                         ④
    ITERATION_MAX = 4L,                                         ⑤
    STEP_SIZE_EXCEEDED = 5L                                     ⑥
};
```

① OK.
② Relative gradient is within given tolerance.
③ Relative step size of successive iterates is within given tolerance.
④ Last global step failed to locate a point lower than `estimate`.
⑤ Reached maximum number of iterations.
⑥ Maximum step size `stepmax` exceeded five consecutive times.

See the manual page for `nlm` for more detail about these statuses.

Example

Maximize the function $f(x) = \exp(-x^\top x)$. The associated gradient and Hessian functions are $g(x) = -2f(x)x$ and $h(x) = (4xx^\top - 2I)f(x)$, respectively. Consider three calls: first with a numerical gradient and Hessian, then with explicitly coded gradient, and finally with both explicitly coded gradient and Hessian. A C++ function with Rcpp interface is defined in the file `examples/nlm.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List nlm_ex(Rcpp::NumericVector x0)
{
    fntl::dfv f = [](const Rcpp::NumericVector& x) {
        Rcpp::NumericVector xx = Rcpp::pow(x, 2);
        double ss = Rcpp::sum(xx);
        return std::exp(-ss);
    };

    fntl::vfv g = [&](const Rcpp::NumericVector& x) {
        double fx = f(x);
            Rcpp::NumericVector out = -2 * fx * x;
        return out;
    };

    fntl::mfv h = [&](const Rcpp::NumericVector& x) {
        unsigned int n = x.size();
```

```
        Rcpp::NumericMatrix out(n,n);
        double fx = f(x);

        for (unsigned int j = 0; j < n; j++) {
            for (unsigned int i = 0; i < n; i++) {
                out(i,j) = fx * ( 4*x(i)*x(j) - 2*(i == j) );
            }
        }

        return out;
    };

    fntl::nlm_args args;
    args.fnscale = -1;

    // 1. Use default numerical gradient and hessian.
    // 2. Use explicitly coded gradient and numerical hessian.
    // 3. Use explicitly coded gradient and numerical hessian.
    auto out1 = fntl::nlm(x0, f, args);
    auto out2 = fntl::nlm(x0, f, g, args);
    auto out3 = fntl::nlm(x0, f, g, h, args);

    return Rcpp::List::create(
        Rcpp::Named("res1") = Rcpp::wrap(out1),
        Rcpp::Named("res2") = Rcpp::wrap(out2),
        Rcpp::Named("res3") = Rcpp::wrap(out3)
    );
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/nlm.cpp")
out = nlm_ex(x0 = rep(1, 4))
nn = c("par", "grad")
print(out$res1[nn])
```

```
$par
[1] -4.999612e-07 -4.999612e-07 -4.999612e-07 -4.999612e-07

$grad
[1] 1.110223e-10 1.110223e-10 1.110223e-10 1.110223e-10
```

```
print(out$res2[nn])
```

```
$par
[1] -7.416746e-13 -7.416746e-13 -7.416746e-13 -7.416746e-13

$grad
[1] -1.483349e-12 -1.483349e-12 -1.483349e-12 -1.483349e-12
```

```
print(out$res3[nn])
```

```
$par
[1] -8.879037e-12 -8.879003e-12 -8.878964e-12 -8.879003e-12

$grad
[1] -1.775807e-11 -1.775801e-11 -1.775793e-11 -1.775801e-11
```

8 Matrix Operations

This section presents several matrix operations based on a lambda function.

8.1 Apply

Apply a function to the elements, rows, or columns of an $m \times n$ matrix. Suppose $X \in \mathbb{R}^{m \times n}$ is a matrix with rows $x_{1\bullet}, \ldots, x_{m\bullet}$ and columns $x_{\bullet 1}, \ldots, x_{\bullet n}$.

The function `mat_apply` is an elementwise application of $f : \mathbb{R} \to \mathbb{R}$ which computes

$$\texttt{mat\_apply}(X, f) = \begin{bmatrix} f(x_{11}) & \cdots & f(x_{1n}) \\ \vdots & \ddots & \vdots \\ f(x_{m1}) & \cdots & f(x_{mn}) \end{bmatrix}.$$

The function `row_apply` is a rowwise application of $g : \mathbb{R}^n \to \mathbb{R}$ which computes

$$\texttt{row\_apply}(X, g) = (g(x_{1\bullet}), \ldots, g(x_{m\bullet})).$$

The function `col_apply` is a columnwise application of $h : \mathbb{R}^m \to \mathbb{R}$ which computes

$$\texttt{col\_apply}(X, h) = (h(x_{\bullet 1}), \ldots, h(x_{\bullet n})).$$

The above replicate the behavior of following R calls, respectively.

```
apply(X, c(1,2), f)
apply(X, 1, f)
apply(X, 2, f)
```

Functions

Primary location of source code is the file inst/include/apply.h.

```
template <typename T, int RTYPE>
Rcpp::Vector<RTYPE> row_apply(
    const Rcpp::Matrix<RTYPE>& X,                                    ①
    const std::function<T(const Rcpp::Vector<RTYPE>&)>& f            ②
)

template <typename T, int RTYPE>
Rcpp::Vector<RTYPE> col_apply(
    const Rcpp::Matrix<RTYPE>& X,                                    ①
    const std::function<T(const Rcpp::Vector<RTYPE>&)>& f            ②
)

template <typename T, int RTYPE>
Rcpp::Matrix<RTYPE> mat_apply(
    const Rcpp::Matrix<RTYPE>& X,                                    ①
    const std::function<T(T)>& f                                     ②
)
```

① An Rcpp matrix object.
② Function f to apply.

The functions `mat_apply`, `row_apply`, and `col_apply` correspond to elementwise, rowwise, and columnwise apply.

Rcpp matrix objects may be of type `NumericMatrix`, `IntegerMatrix`, or one of the others defined in the Rcpp API. The template argument `RTYPE` represents the type of data stored in in the matrix, taking on value `REALSXP` for `NumericMatrix`, `INTSXP` for `IntegerMatrix`, etc. The template argument `T` represents an underlying C++ variable type such as `double`, `int`, etc.

The domain and range of function `f` should match the class of `x` and type of apply operation. As an example, suppose `X` is an object of type `NumericMatrix`.

- The domain of `row_apply` should be of type `const NumericVector&` and the range should be of type `double`.
- The domain of `col_apply` should be of type `const NumericVector&` and the range should be of type `double`.
- The domain and range of `mat_apply` should be of type `double`.

Example

Compute the square of each element of a matrix, then its rowwise sums, then its columnwise sums. A C++ function with Rcpp interface is defined in the file `examples/apply.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List apply_ex(Rcpp::NumericMatrix X)
```

```
{
    fntl::dfd f = [](double x) { return std::pow(x, 2); };
    fntl::dfv g = [](Rcpp::NumericVector x) {
            return Rcpp::sum(x);
    };

    return Rcpp::List::create(
        Rcpp::Named("pows") = fntl::mat_apply(X, f),
        Rcpp::Named("rowsums") = fntl::row_apply(X, g),
        Rcpp::Named("colsums") = fntl::col_apply(X, g)
    );
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/apply.cpp")
X = matrix(1:12, 4, 3)
out = apply_ex(X)
print(out)
```

```
$pows
     [,1] [,2] [,3]
[1,]    1   25   81
[2,]    4   36  100
[3,]    9   49  121
[4,]   16   64  144

$rowsums
[1] 0 0 0 0

$colsums
[1] 0 0 0
```

8.2 Outer

Construct a matrix from a real-valued function of two arguments. Or carry out a matrix multiplication without explicitly constructing the matrix.

Suppose $f(x, x') : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ and $X \in \mathbb{R}^{n \times d}$ is a matrix with rows $x_1, \ldots, x_n$. Also let $a = [a_1 \cdots a_n]^\top$ be a fixed vector. The `outer` operation computes the $n \times n$ symmetric matrix

$$\text{outer}(X, f) = \begin{bmatrix} f(x_1, x_1) & \cdots & f(x_1, x_n) \\ \vdots & \ddots & \vdots \\ f(x_n, x_1) & \cdots & f(x_n, x_n) \end{bmatrix}$$

and the `outer_matvec` operation computes the n-dimensional vector

$$\texttt{outer\_matvec}(X, f, a) = \begin{bmatrix} f(x_1, x_1) & \cdots & f(x_1, x_n) \\ \vdots & \ddots & \vdots \\ f(x_n, x_1) & \cdots & f(x_n, x_n) \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}.$$

Now suppose $f(x, y) : \mathbb{R}^{d_1} \times \mathbb{R}^{d_2} \to \mathbb{R}$, $X \in \mathbb{R}^{m \times d_1}$ is a matrix with rows $x_1, \ldots, x_m$, $Y \in \mathbb{R}^{n \times d_2}$ is a matrix with rows $y_1, \ldots, y_n$, and $a = [a_1 \cdots a_n]^\top$ is a fixed vector. The `outer` operation computes the $m \times n$ matrix

$$\texttt{outer}(X, Y, f) = \begin{bmatrix} f(x_1, y_1) & \cdots & f(x_1, y_n) \\ \vdots & \ddots & \vdots \\ f(x_m, y_1) & \cdots & f(x_m, y_n) \end{bmatrix}$$

and the `outer_matvec` operation computes the m-dimensional vector

$$\texttt{outer\_matvec}(X, Y, f, a) = \begin{bmatrix} f(x_1, y_1) & \cdots & f(x_1, y_n) \\ \vdots & \ddots & \vdots \\ f(x_m, y_1) & \cdots & f(x_m, y_n) \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix}.$$

The `outer` functions above replicate the behavior of the `outer` function in R.

Functions

Primary location of source code is the file inst/include/outer.h.

```
Rcpp::NumericMatrix outer(
    const Rcpp::NumericMatrix& X,                                    ①
    const dfvv& f                                                    ③
)

Rcpp::NumericMatrix outer(
    const Rcpp::NumericMatrix& X,                                    ①
    const Rcpp::NumericMatrix& Y,                                    ②
    const dfvv& f                                                    ③
)

Rcpp::NumericVector outer_matvec(
    const Rcpp::NumericMatrix& X,                                    ①
    const dfvv& f,                                                   ③
    const Rcpp::NumericVector& a                                     ④
)

Rcpp::NumericVector outer_matvec(
    const Rcpp::NumericMatrix& X,                                    ①
    const Rcpp::NumericMatrix& Y,                                    ②
    const dfvv& f,                                                   ③
    const Rcpp::NumericVector& a                                     ④
)
```

① An Rcpp matrix object of dimension $m \times d$.
② An Rcpp matrix object of dimension $n \times d$.
③ Function f to apply.
④ An Rcpp vector object of dimension n.

Example

Compute the distance between pairs of rows of X, then compute the distance between each pairs of rows taken from X and Y. Then multiply the respective matrices by $a = [1, \dots, 1]$. A C++ function with Rcpp interface is defined in the file `examples/outer.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List outer_ex(Rcpp::NumericMatrix X, Rcpp::NumericMatrix Y,
    Rcpp::NumericVector a, Rcpp::NumericVector b)
{
    fntl::dfvv f =
    [](Rcpp::NumericVector x, Rcpp::NumericVector y) {
        double norm2 = Rcpp::sum(Rcpp::pow(x - y, 2));
        return std::sqrt(norm2);
    };

    return Rcpp::List::create(
        Rcpp::Named("out1") = fntl::outer(X, f),
        Rcpp::Named("out2") = fntl::outer(X, Y, f),
        Rcpp::Named("out3") = fntl::outer_matvec(X, f, a),
        Rcpp::Named("out4") = fntl::outer_matvec(X, Y, f, b)
    );
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/outer.cpp")
m = 5; n = 3; d = 2
X = matrix(rnorm(10), m, d)
Y = matrix(rnorm(6), n, d)
a = rep(1, m)
b = rep(1, n)
out = outer_ex(X, Y, a, b)
print(out)
```

```
$out1
          [,1]      [,2]      [,3]      [,4]     [,5]
[1,] 0.0000000 1.1024866 0.4622356 0.7590347 1.882926
```

```
[2,] 1.1024866 0.0000000 0.8913073 1.8098040 2.981883
[3,] 0.4622356 0.8913073 0.0000000 1.1963281 2.230383
[4,] 0.7590347 1.8098040 1.1963281 0.0000000 1.211293
[5,] 1.8829261 2.9818827 2.2303831 1.2112927 0.000000

$out2
          [,1]      [,2]      [,3]
[1,] 2.3684959 1.4722509 1.0635563
[2,] 3.4607857 2.5105686 1.6286383
[3,] 2.7293683 1.6365318 1.4789857
[4,] 1.6639318 1.1887846 0.8832083
[5,] 0.5039864 0.9974981 1.9383965

$out3
[1] 4.206683 6.785481 4.780254 4.976459 8.306485

$out4
[1] 4.904303 7.599993 5.844886 3.735925 3.439881
```

8.3 Matrix-Free Linear Solve

Solve a linear system $Ax = b$ for symmetric positive definite $A \in \mathbb{R}^{n \times n}$ (Nocedal and Wright 2006). Here the operation Ax is specified through a function $\ell(x) = Ax$. This can be used to avoid explicit storage of large matrices when A is sparse. The solution x^* is found by minimizing the quadratic function

$$f(x) = \frac{1}{2} x^\top \ell(x) - b^\top x,$$

so that $f'(x^*) = 0 \iff Ax^* = b$. Minimization is carried out using the conjugate gradient method in Section 7.4.

Functions

Primary location of source code is the file inst/include/solve_cg.h.

```
cg_result solve_cg(
    const vfv& l,                                    ①
    const Rcpp::NumericVector& b,                    ②
    const Rcpp::NumericVector& init,                 ③
    const cg_args& args                              ④
)

cg_result solve_cg(
    const vfv& l,                                    ①
    const Rcpp::NumericVector& b,                    ②
    const Rcpp::NumericVector& init                  ③
)
```

```
cg_result solve_cg(
    const vfv& l,                                                    ①
    const Rcpp::NumericVector& b                                     ②
)
```

① The function $\ell : \mathbb{R}^n \to \mathbb{R}^n$.
② The vector $b \in \mathbb{R}^n$.
③ Initial value for x.
④ Optional arguments to CG method.

The routine checks that `l(init)` is an n-dimensional vector, but there is no check that the operation ℓ represents a symmetric positive definite matrix.

See Section 7.4 for details on `cg_args` and `cg_result`.

Example

Solve the equation $Ax = b$ with $n \times n$ matrix A having value 2 on the main diagonal and value 1 on the upper and lower diagonals; let $b = [1, \dots, 1]$. The initial value for the solution is taken to be the default $x = 0$. A C++ function with Rcpp interface is defined in the file `examples/solve-cg.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::List solve_cg_ex(Rcpp::NumericVector b)
{
    fntl::vfv l = [](Rcpp::NumericVector x) {
        unsigned int n = x.size();
        Rcpp::NumericVector out(n);

        for (unsigned int i = 1; i < n-1; i++) {
            out(i) = x(i-1) + 2*x(i) + x(i+1);
        }
        out(0) = 2*x(0) + x(1);
        out(n-1) = x(n-2) + 2*x(n-1);

        return out;
    };

    auto out = fntl::solve_cg(l, b);
    return Rcpp::wrap(out);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/solve-cg.cpp")
b = rep(1, 10)
out = solve_cg_ex(b)
print(out)
```

```
$par
 [1] 0.45454542 0.09090782 0.36363901 0.18181637 0.27272775 0.27272775
 [7] 0.18181637 0.36363901 0.09090782 0.45454542

$value
[1] -1.363636

$fncount
[1] 71

$grcount
[1] 31

$status
[1] 0
```

Compare the above to solving dense the system as follows.

```
A = matrix(0, 10, 10)
diag(A) = 2
A[cbind(1:9, 2:10)] = 1
A[cbind(2:10, 1:9)] = 1
solve(A, b)
```

```
 [1] 0.45454545 0.09090909 0.36363636 0.18181818 0.27272727 0.27272727
 [7] 0.18181818 0.36363636 0.09090909 0.45454545
```

8.4 Which

Identify the indices of a matrix which satisfy a given indicator function. Specifically, let $X \in \mathbb{S}^{m \times n}$ be a matrix whose elements are in the domain $\mathbb{S}$ which may be doubles, integers, or another Rcpp `Matrix` type. Let $f : \mathbb{S} \to \{0, 1\}$ be an indicator function and suppose k of the mn elements in X satisfy the indicator. The `which` operation produces a $k \times 2$ matrix

$$\text{which}(X, f) = \begin{bmatrix} i_1 & j_1 \\ \vdots & \vdots \\ i_k & j_k \end{bmatrix},$$

where each pair (i_ℓ, j_ℓ) are coordinates of an element in X that satisfies f. Indices i_ℓ and j_ℓ represent the row and column index, respectively. Indices are *zero-based* by default as they are primarily intended to be used in C++ code.

An equivalent R operation is the following. Note that indices in R are *one-based*.

```
which(f(X), arr.ind = TRUE)
```

Functions

Primary location of source code is the file inst/include/which.h.

```
template <typename T, int RTYPE>
Rcpp::IntegerMatrix which(
    const Rcpp::Matrix<RTYPE>& X,                    ①
    const std::function<bool(T)>& f),                ②
    bool one_based = false                           ③
)
```

① An Rcpp matrix object.
② Function f to apply.
③ If `one_based = false`, zero-based indices are produced which are suitable for use with C++ code. If `true`, one-based indices are produced.

Example

Construct a matrix and identify the elements between 0 and 0.5. A C++ function with Rcpp interface is defined in the file `examples/which.cpp`.

```
// [[Rcpp::depends(fntl)]]
#include "fntl.h"

// [[Rcpp::export]]
Rcpp::IntegerMatrix which_ex(Rcpp::NumericMatrix X)
{
    std::function<bool(double)> f = [](double x) { return x > 0 && x < 0.5; };
    return fntl::which(X, f);
}
```

Call the function from R.

```
Rcpp::sourceCpp("examples/which.cpp")
x = runif(10, -1, 1)
X = matrix(x, 2, 5)
out = which_ex(X)
print(X)
```

```
           [,1]       [,2]       [,3]      [,4]      [,5]
[1,] -0.1206053 -0.4555856 -0.2394214 0.4710019 0.6293048
[2,]  0.8606477  0.3180581  0.8707816 0.1801809 0.7648177
```

```
print(out)
```

```
     row col
[1,]   1   1
[2,]   0   3
[3,]   1   3
```

Here is the result in R for comparison.

```
f = function(x) { x > 0 & x < 0.5 }
which(f(X), arr.ind = TRUE) - 1
```

```
     row col
[1,]   1   1
[2,]   0   3
[3,]   1   3
```

9 References

Bezanson, Jeff, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. "Julia: A Fresh Approach to Numerical Computing." *SIAM Review* 59 (1): 65–98. https://doi.org/10.1137/141000671.

Brent, R. P. 1973. *Algorithms for Minimization Without Derivatives*. Prentice-Hall.

Byrd, Richard H., Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. 1995. "A Limited Memory Algorithm for Bound Constrained Optimization." *SIAM Journal on Scientific Computing* 16 (5): 1190–1208. https://doi.org/10.1137/0916069.

Dennis, J. E., and Robert B. Schnabel. 1983. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall.

Eddelbuettel, Dirk, John W. Emerson, and Michael J. Kane. 2024. *Boost c++ Header Files*. https://cran.r-project.org/package=BH.

Eddelbuettel, Dirk, and Romain Francois. 2024. *'Rcpp' Integration for 'GNU GSL' Vectors and Matrices*. https://cran.r-project.org/package=RcppGSL.

Eddelbuettel, Dirk, Romain Francois, JJ Allaire, Kevin Ushey, Qiang Kou, Nathan Russell, Inaki Ucar, Douglas Bates, and John Chambers. 2024. *Rcpp: Seamless r and c++ Integration*. https://CRAN.R-project.org/package=Rcpp.

Eddelbuettel, Dirk, and Conrad Sanderson. 2014. "RcppArmadillo: Accelerating R with High-Performance C++ Linear Algebra." *Computational Statistics and Data Analysis* 71: 1054–63. https://doi.org/10.1016/j.csda.2013.02.005.

Fletcher, R., and C. M. Reeves. 1964. "Function Minimization by Conjugate Gradients." *The Computer Journal* 7 (2): 149–54. https://doi.org/10.1093/comjnl/7.2.149.

Galassi, Mark, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Patrick Alken, Michael Booth, Fabrice Rossi, and Rhys Ulerich. 2009. *GNU Scientific Library Reference Manual*. 3rd ed. http://www.gnu.org/software/gsl/.

Gilbert, Paul, and Ravi Varadhan. 2019. *numDeriv: Accurate Numerical Derivatives*. https://CRAN.R-project.org/package=numDeriv.

Ihaka, Ross, and Robert Gentleman. 1996. "R: A Language for Data Analysis and Graphics." *Journal of Computational and Graphical Statistics* 5 (3): 299–314. https://doi.org/10.1080/10618600.1996.10474713.

Nash, J. C. 1990. *Compact Numerical Methods for Computers*. 2nd ed. Adam Hilger.

Nelder, J. A., and R. Mead. 1965. "A Simplex Method for Function Minimization." *The Computer Journal* 7 (4): 308–13. https://doi.org/10.1093/comjnl/7.4.308.

Nocedal, Jorge, and Stephen J. Wright. 2006. *Numerical Optimization*. 2nd ed. Springer.

Pan, Yi. 2022. *Roptim: General Purpose Optimization in R Using C++*. https://CRAN.R-project.org/package=roptim.

Piessens, R., E. deDoncker-Kapenga, C. W. Uberhuber, and D. K. Kahaner. 1983. *QUADPACK: A Subroutine Package for Automatic Integration*. Springer.

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. Cambridge University Press.

Qiu, Yixuan, Sreekumar Balan, Matt Beall, Mark Sauder, Naoaki Okazaki, and Thomas Hahn. 2023. *RcppNumerical: 'Rcpp' Integration for Numerical Computing Libraries*. https://CRAN.R-project.org/package=RcppNumerical.

Quarteroni, Alfio, Riccardo Sacco, and Fausto Saleri. 2007. *Numerical Mathematics*. 2nd ed. Springer. https://doi.org/https://doi.org/10.1007/b98885.

R Core Team. 2024a. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

———. 2024b. *Writing R Extensions*. https://cran.r-project.org/doc/manuals/r-release/R-exts.html.

Wickham, Hadley. 2019. *Advanced R*. 2nd ed. Chapman & Hall/CRC.