Writing R Extensions in Rust

by David B. Dahl *[Version 0.1.38]*

Abstract This paper complements "Writing R Extensions," the official guide for writing R extensions, for those interested in developing R packages using Rust. It highlights idiosyncrasies of R and Rust that must be addressed by any integration and describes how to develop Rust-based packages which comply with the CRAN Repository Policy. This paper introduces the **cargo** framework, a transparent Rust-based API which wraps commonly-used parts of R's API with minimal overhead and allows a programmer to easily add additional wrappers.

Introduction

Computationally-intensive R packages are typically implemented using C, Fortran, or C++ for the sake of performance. The R Core Team maintains a document called "Writing R Extensions" which describes R's API for creating packages. This paper supplements that official guide by (i) discussing issues involved in the integration of R and Rust and (ii) providing an R package to help those interested in writing R packages based on Rust.

While both R and Rust provide foreign function interfaces (FFI) based on C (Kernighan and Ritchie, 2006), each language has its own idiosyncrasies that require some care when interfacing with the other. Packages published on CRAN (`https://cran.r-project.org/`) are subject to the CRAN Repository Policy. This paper also describes how to avoid pitfalls which may prevent acceptance of a Rust-based package on CRAN or waste time of CRAN maintainers and package contributors.

The paper introduces the newly-released **cargo** (Dahl, 2021) package which provides a framework for developing CRAN-compliant R packages using Rust and shows how to make Rust-based wrappers for R's C API. It is important to emphasize that a package developed with the **cargo** framework does *not* depend on the **cargo** package, either in source or binary form. That is, the **cargo** package produces an R package structure with all the necessary Rust code and scripts such that the package is then independent of the **cargo** package. Developers can then extend the framework for their own purposes within the generated package structure. Further, although a source package will obviously depend on Rust, there are no runtime dependencies on Rust or any other libraries, resulting in a binary package that is easy for others to use. Separate from package development, the **cargo** package also allows Rust code to be directly embedded in an R script.

One of the purposes of this paper is to encourage developers to consider Rust for writing high-performance R packages. A second purpose is to discuss technical issues which arise when interfacing R and Rust and to document the design choices of the **cargo** framework. The **cargo** framework seeks to: (i) provide a Rust interface for commonly used parts of the R API, (ii) show the developer how they can easily extend the framework to cover other parts of the R API, (iii) minimize the runtime overhead when interfacing between R and Rust, and (iv) be as transparent as possible on how the framework interfaces R and Rust. This paper assumes some familiarity with "Writing R Extensions" and package development using R's API. The paper also assumes some familiarity with Rust. The interested reader is directed to a plethora of resources online, including "The Rust Programming Language" (`https://doc.rust-lang.org/stable/book/`).

The paper is organized as follows. A brief history on Rust and its use in R is outlined. Setting up the Rust toolchain for R package development is discussed next, followed by an overview of the various parts of an R package using the **cargo** framework. Low-level and high-level interfaces between R and Rust are introduced. Threading issues and seeding a random number generator are also discussed. Defining a R function by embedding Rust code directly in an R script is shown. Finally, the paper ends with benchmarks and concluding comments.

Background on Rust and its use in R

Rust (`https://www.rust-lang.org/`) is a statically-typed, general-purpose programming language which emphasizes memory safety without compromising runtime performance. Its memory safety guarantees (against, e.g., buffer overflows, dangling pointers, and race conditions) are achieved through the language's design and the compiler's borrow checker. This avoids the memory and CPU overhead inherent in garbage-collected languages. Concurrent programming is straightforward in Rust, where most concurrency errors are compile-time errors rather than difficult-to-reproduce runtime errors. Developer productivity is aided by rustup (toolchain installer and upgrader), Cargo (package manager for downloading dependencies, publishing code, and building dependencies and

code), Rustfmt (automatic code formatter), and Clippy (linting tool to catch common mistakes and improve performance and readability).

Rust first appeared in 2010 as a Mozilla project, had its first stable release in 2015, and has been rated the "most loved programming language" in the Stack Overflow Annual Developer Survey (https://insights.stackoverflow.com/survey/) every year since 2016. The Rust Foundation was formed in 2021 with the founding members Amazon Web Services, Google, Huawei, Microsoft, and Mozilla. Google recently announced support for Rust within Android Open Source Project (AOSP) as an alternative to C and C++. Experimental Rust support for developing subsystems and drivers for the Linux kernel has been submitted. Linus Torvalds has been quoted on several occasions as being welcoming of the possibility of using Rust alongside C for kernel development.

Members of the R community have also been interested in Rust. The first major effort to integrate R and Rust appears to have started in early 2016 with the now-deprecated https://github.com/rustr project. The first Rust-based package appeared on CRAN in 2018 with Jeroen Ooms' **gifski** package (Ooms, 2021), with an accompanying presentation at the 2018 European R Users Meeting (eRum2018) describing how a developer can use Rust code in an R package. The approach requires the package developer to write C code which then calls Rust code. Under this approach, the Rust code itself does not have access to R's API.

In 2019, my **salso** package (Dahl et al., 2021) was the second Rust-based package on CRAN. It followed **gifski**'s approach of writing C code that calls Rust code. Around the time that the third Rust-based package **baseflow** (Pelletier et al., 2021) was accepted to CRAN, the CRAN maintainers noted that **gifski**, **salso**, and **baseflow** violated the policy that "packages should not write ... on the file system apart from the R session's temporary directory" since Cargo caches downloaded dependencies by default and uses all available CPU cores. This inspired me in early 2021 to write the **cargo** package to facilitate using Cargo in conformance with CRAN's policies and to download precompiled static libraries in case the required version of the Rust toolchain is not available on a particular CRAN build machine. It also became clear that writing C code that glues the R and Rust code is tedious, error prone, and difficult to refactor. As such, I expanded the **cargo** package to facilitate developing Rust-based packages that avoid the need to write the C glue code, allowing R to call directly into Rust code and allowing Rust code to callback into R's API directly. In 2021, CRAN accepted **caviarpd** (Dahl et al.) as another package developed using the **cargo** framework and the **salso** package was ported to the framework.

Another exciting project that interfaces R and Rust is the extendr project (https://github.com/extendr). Andy Thomason started working on the extendr project in 2020, attracting Claus Wilke and several other developers. The extendr project seeks not only to facilitate writing R packages in Rust, but also to embed the R interpreter in a Rust program. In 2021, the project released the **rextendr** package (Wilke et al., 2021) on CRAN to facilitate developing Rust-based packages. In 2021, the **baseflow** package was ported to use **rextendr**, and the **string2path** package (Yutani, 2021) became another package on CRAN developed with the aid of **rextendr**.

Those interested in interfacing Rust and R should keep an eye on the extendr project as it continues to evolve. The project is working to provide extensive automatic conversion between R types (e.g., vectors, lists, data.frames, environments, etc.) and Rust types, including attempts to handle thorny issues such as R's missing value `NA` and R's fluidity in vectors of storage mode '`double`' and '`integer`'. It aspires to eventually provide a Rust interface for all of the functionality provided by the R API to alleviate the Rust developer from having to dive into the details of R's API.

The **rextendr** package and the **cargo** package both seek to provide functionality to develop R packages which can call directly into Rust and call back to R from Rust. The extendr project is hosted on public GitHub repositories and is under rapid development; their project shows what is possible and their open discussion influenced some of my choices for the **cargo** package. The **rextendr** package and the **cargo** package address various technical issues differently and choose different design trade-offs. The advantage of the **cargo** package is its transparency and extendability, whereas the benefit of the **rextendr** package is that it provides many behind-the-scenes type conversions and aims to be more comprehensive. The lean nature of the **cargo** framework makes it simple to understand how R and Rust interface.

Installing the Rust toolchain

Developing Rust-based R packages requires the installation of several tools. The first step is to install the usual toolchain bundle to compile C/C++/Fortran packages for the chosen operating system. For example, on Windows, install Rtools (https://cran.r-project.org/bin/windows/Rtools/). On MacOS, follow the instructions here: https://mac.r-project.org/tools/.

Install **cargo** from CRAN using `install.packages("cargo")` and run `cargo::setup_rust()`. This

function downloads the rustup (https://rustup.rs) tool chain installer, runs it, and adds all the targets listed by `cargo::target(TRUE)` to prepare for cross compiling Rust static libraries.

Rust has a six-week release cycle and the Rust toolchain is easily upgraded by running 'rustup update' at the terminal. Incompatible changes are opt-in only, so new releases are always guaranteed to run old code. Because there are immediate benefits and no costs to upgrading, Rust developers frequently develop against the latest Rust version to take advantage of new features, optimizations, and bug fixes.

Rust's rapid release cycle, however, presents challenges when submitting Rust-based packages to CRAN, as a CRAN build server may not have a recent version of the Rust toolchain. Moreover, the toolchain may not even be available on a particular CRAN build machine. A solution to this problem is to host precompiled static libraries, which is a common practice for packages unrelated to Rust. (See, for example, https://github.com/rwinlib.) The **cargo** framework provides tools to cross compile static library for the Rust component of an R package and to download them when a sufficient version of the Rust toolchain cannot be found during installation. Compiling from source is completely supported by the **cargo** framework if the Rust toolchain meets the minimum version specified in the package's 'DESCRIPTION' file.

Overview of package development using the cargo framework

The **cargo** package facilitates the development of R packages based on Rust. Development starts by creating a new package using, for example, `cargo::new_package("/path/to/package/foo")` to generate the package **foo** at the filesystem path '/path/to/package/'. If using RStudio, this can be accomplished using "File" -> "New Project..." -> "New Directory" -> "R Package Using Rust and the 'cargo' Framework". This generates and installs a complete working package that developers can modify for their own needs.

The directory structure of the new **foo** package is:

```
foo
├── DESCRIPTION
├── INSTALL
├── LICENSE
├── NAMESPACE
├── R
│   ├── convolve2.R
│   ├── myrnorm.R
│   ├── rustlib.R
│   └── zero.R
├── man ...
├── src
│   ├── Makevars
│   ├── Makevars.win
│   ├── rustlib
│   │   ├── Cargo.toml
│   │   ├── roxido ...
│   │   ├── roxido_macro ...
│   │   └── src
│   │       ├── lib.rs
│   │       └── registration.rs
│   └── shim.c
└── tools
    ├── cargo.R
    └── staticlib.R
```

Several of the resulting files and directories are specific to packages developed with the **cargo** framework. The 'src/Makevars' and 'src/Makevars.win' direct R to use the 'tools/staticlib.R' script to compile the static Rust library defined in 'src/rustlib' or, as a fallback, to download a precompiled static library. The download URL needs to be provided in the 'tools/staticlib.R' script. These static libraries can built by the `cargo::cross_compile` function. Notice that the 'DESCRIPTION' file has an entry '`SystemRequirements: Cargo (>= 1.54) for installation from sources: see INSTALL file`'. The minimum required Cargo version should be updated and the developer can determine this using cargo-msrv (https://crates.io/crates/cargo-msrv). The 'tools/cargo.R' script finds and runs the Cargo package manager according to CRAN policies by, for example, using no more than two CPU threads and downloading dependencies to a temporary directory. Unfortunately, the dependencies

must then be redownloaded and recompiled every time the package is reinstalled, which is a hassle during package development. To avoid these limitations on a local development machine, the package developer can add the followings to their personal '.Rprofile' file:

```
Sys.setenv(R_CARGO_SAVE_CACHE="TRUE")
Sys.setenv(R_CARGO_BUILD_JOBS="0")
```

As an aid to other Rust-based packages *not* using the **cargo** framework, the functionality provided by the 'tools/cargo.R' script is also available as the `run` function in the **cargo** package.

There are several calls to the `.Call` function among the scripts in the 'R' directory. The function in 'R/myrnorm.R', for example, has `.Call(.myrnorm,n,mean,sd)` which calls the Rust function `myrnorm` defined in 'src/rustlib/src/lib.rs':

```
1   mod registration;
2   use roxido::*;
3
4   #[roxido]
5   fn myrnorm(n: Rval, mean: Rval, sd: Rval) -> Rval {
6       unsafe {
7           use rbindings::*;
8           use std::convert::TryFrom;
9           let (mean, sd) = (Rf_asReal(mean.0), Rf_asReal(sd.0));
10          let length = isize::try_from(Rf_asInteger(n.0)).unwrap();
11          let vec = Rf_protect(Rf_allocVector(REALSXP, length));
12          let slice = Rval(vec).slice_double().unwrap();
13          GetRNGstate();
14          for x in slice { *x = Rf_rnorm(mean, sd); }
15          PutRNGstate();
16          Rf_unprotect(1);
17          Rval(vec)
18      }
19  }
```

Notice that the `myrnorm` function has the `#[roxido]` attribute and takes three arguments `n`, `mean`, and `sd`, all of type `Rval`, and returns a value of type `Rval`. The `#[roxido]` attribute is a procedural macro defined in 'src/rustlib/roxido_macro/src/lib.rs' which adds the qualifiers `#[no_mangle] extern "C"` when compiling to tell the Rust compiler to make the `myrnorm` function callable directly from R. The attribute also ensures that all arguments are of type `Rval` and that the return type is `Rval`. The `#[roxido]` attribute also wraps the body of the function in a call to Rust's `std::panic::catch_unwind` since unwinding from Rust code into foreign code is undefined behavior and likely crashes R. When a panic is caught, it is turned into an R error which gives the corresponding message from Rust and the line number of the panic. The package developer is encouraged to study the definition of the `#[roxido]` attribute in 'src/rustlib/roxido_macro/src/lib.rs' to better understand the interface between R and Rust.

When a developer wants to make another Rust function callable by R, say a function named `bar` taking two arguments `x` and `y`, the developer adds the `.Call(.bar,x,y)` in a script under the 'R' directory of the package and then runs `cargo::register_calls("/path/to/package/foo")`. This automatically regenerates the 'src/rustlib/src/registration.rs' file and does two things. First, the updated file provides a stub for a Rust function `bar` with arguments `x` and `y` in a commented-out block. This stub can then be copied to the 'src/rustlib/src/lib.rs' file and the function can be implemented. Second, code is generated to register functions when R loads the shared library. Again, the package developer is encouraged to study the 'src/rustlib/src/registration.rs' for examples on calling R's API from Rust.

Low-level interface to R's API

The `myrnorm` function in Rust illustrates directly using R's API in Rust. Line 7 of the listing is '`use rbindings::*`', which provides direct access to R's API through Rust bindings. These are automatically generated by the bindgen utility (https://rust-lang.github.io/rust-bindgen/) from the following R header files: 'Rversion.h', 'R.h', 'Rinternals.h', 'Rinterface.h', 'R_ext/Rdynload.h', and 'Rmath.h', although only those definitions and functions that are documented to be part of R's API (as specific by "Writing R Extensions") should be used. The documentation for the Rust bindings can be browsed using the `cargo::api_documentation` function or by executing '`cargo doc --open`' when in the 'src/rustlib/roxido' directory. Note that most of the functions in the `rbindings` module require an `SEXP` value, i.e., a pointer to R's internal `SEXPREC` structure. The `Rval` is defined as '`pub struct Rval(pub SEXP)`', a *newtype* pattern that wraps the `SEXP` value. The newtype pattern provide type safety and encapsulation, which

we utilize in the high-level interface described in the next section. Because of zero-cost abstraction, the Rust compiler generates code as if SEXP were used directly. The upshot is that, when calling R API functions, the SEXP must be extracted from an Rval value, e.g., if mean is an Rval, use mean.0 to extract its SEXP, as in line 9. Conversely, when returning from a function marked with #[roxido] attribute, wrap the SEXP value x in Rval(x), as in line 17.

When accessing an R API function from Rust, care should be taken so that the R function does not throw an error. If Rust code calls an R function that throws an error, a long jump occurs over Rust stack frames, which prevents Rust from doing its usual freeing of heap allocations, resulting in a memory leak. For example, before calling REAL(x) to receive a pointer of type *mut f64 (i.e., *double in C), the developer should check that the storage mode of x is indeed 'double' by checking against Rf_isReal(x). If not, a long jump will occur when calling REAL(x).

Care must also be taken when calling R API functions that might catch a user interrupt (e.g, pressing Ctrl-C or hitting the stop button in RStudio) because an interrupt also produces a long jump and leaks memory. One R API function that catches interrupts, for example, is the Rprintf function for printing to R's console.

High-level interface wrapping R's API

To avoid the pitfalls of R API functions throwing errors or catching interrupts when called from Rust, the **cargo** package also provides a high-level interface defined in the r module. This high-level interface also alleviates the developer from deciding when results from R API functions should be protected from the R's garbage collection and the necessary bookkeeping involved in calling the Rf_unprotect function. Finally, the high-level interface provides a more idiomatic API for Rust developers. The high-level interface is not a comprehensive wrapper over R's API, but it covers common use cases and the developer can easily expand it by adding to the 'src/rustlib/roxido/src/r.rs' in the package. That is, the developer does not need to wait for the release of a new version of the **cargo** package. The high-level interface provides a check_user_interrupt function to test whether the user has tried to interrupt execution. The rprintln! macro behaves just like Rust's standard println! macro, but prints to the R console and returns true if the user interrupted. Much of the interface is provided by associated functions for the Rval structure. See the API documentation for details.

The package generated by the cargo::new_package function provides two examples of the high-level interface. These are translations of examples in "Writing R Extensions". Consider first the convolve2 function from Section 5.10.1 "Calling .Call". The translation is provided in 'src/rustlib/src/lib.rs' and shown below.

```
21      #[roxido]
22      fn convolve2(a: Rval, b: Rval) -> Rval {
23          let (a, xa) = a.coerce_double(&mut pc).unwrap();
24          let (b, xb) = b.coerce_double(&mut pc).unwrap();
25          let (ab, xab) = Rval::new_vector_double(a.len() + b.len() - 1, &mut pc);
26          for xabi in xab.iter_mut() { *xabi = 0.0 }
27          for (i, xai) in xa.iter().enumerate() {
28              for (j, xbj) in xb.iter().enumerate() {
29                  xab[i + j] += xai * xbj;
30              }
31          }
32          ab
33      }
```

Notice on lines 23 and 24 the calls to Rval's coerce_double method. The developer is encouraged to read the definition of this method in 'src/rustlib/roxido/src/r.rs', but the gist of the method is to check R's type of the Rval and convert it to R's storage mode 'double', if needed and if possible. The method returns either a tuple giving a (potentially-new) Rval and an f64 slice into it, or an error. If the developer is confident that the method will not fail, the developer can simply call the unwrap method, as in lines 23 and 44, but more formal error handling can be implemented in the usual Rust manner. If unwrap is called on an error message, the code will panic and a helpful message regarding the location of the panic is displayed in the R console. No memory leak occurs and the R session is still valid. Thus, panics in the **cargo** framework are controlled events.

In contrast to the coerce_double method, a slice into R's memory for vectors of doubles, integers, and logicals can be obtained without a potential memory allocation using x.slice_double(), x.slice_integer(), and x.slice_logical() when x is an Rval. In any case, these slices are views into R's internal memory. Care should be taken when dealing with R's special values. For example, R's NA value for an element of an 'integer' vector corresponds to Rust's i32::MIN (which is not a special

value in Rust). So, for example, `NA_integer_ * 0L` in R equals `NA_integer_`, but it equals `0` in Rust. Associated functions, such as `Rval::is_na_integer`, are provided to test against R's special values. See Section 5.10.3 "Missing and special values" in "Writing R Extensions" for a discussion of this issue.

Notice the argument to the `coerce_double` method on lines 23 and 24 is `&mut pc`. The wrapper code provided by the `#[roxido]` attribute includes `let mut pc = Pc::new()`. Many of the functions take a shared mutable reference to a `Pc` structure. The purpose of the `Pc` structure is to handle the bookkeeping associated with `Rf_protect` and `Rf_unprotect` calls related to R's garbage collection. It has a single public method `protect` which takes an `SEXP`, calls `Rf_protect` on it, increments an interval counter, and returns the `SEXP`. When an instance of the `Pc` structure goes out of scope, the Rust compiler automatically inserts a call to its associated `drop` function which calls `Rf_unprotect(x)` using its interval counter `x`. Not only does the developer not need to manually track the number of protected items, the developer does not need to worry about when a value should be protected. If the method requires a shared mutable reference to a `Pc`, then protection is needed and automatically handled by the function and not the developer.

Now consider the `zero` function described in Section 5.11.1 "Zero-finding" of "Writing R Extensions". The translation to the **cargo** framework is provided in the package generated by the `new_package` function. The code is provided in 'src/rustlib/src/lib.rs' and shown below. As with the previous `convolve2` function, this is a "drop-in" replacement for the function defined in "Writing R Extensions".

```
35      #[roxido]
36      fn zero(f: Rval, guesses: Rval, stol: Rval, rho: Rval) -> Rval {
37          let slice = guesses.slice_double().unwrap();
38          let (mut x0, mut x1, tol) = (slice[0], slice[1], stol.as_f64());
39          if tol <= 0.0 { panic!("non-positive tol value"); }
40          let symbol = Rval::new_symbol("x", &mut pc);
41          let feval = |x: f64| {
42              let mut pc = Pc::new();
43              symbol.assign(Rval::new(x, &mut pc), rho);
44              f.eval(rho, &mut pc).unwrap().as_f64()
45          };
46          let mut f0 = feval(x0);
47          if f0 == 0.0 { return Rval::new(x0, &mut pc); }
48          let f1 = feval(x1);
49          if f1 == 0.0 { return Rval::new(x1, &mut pc); }
50          if f0 * f1 > 0.0 { panic!("x[0] and x[1] have the same sign"); }
51          loop {
52              let xc = 0.5 * (x0 + x1);
53              if (x0 - x1).abs() < tol { return Rval::new(xc, &mut pc); }
54              let fc = feval(xc);
55              if fc == 0.0 { return Rval::new(xc, &mut pc); }
56              if f0 * fc > 0.0 { x0 = xc; f0 = fc; } else { x1 = xc; }
57          }
58      }
```

This example shows the creation of new R objects from Rust values (e.g., lines 40, 43, 47, etc.) and extracting Rust values from R objects (e.g., 37, 38, and 44). Line 44 demonstrates evaluating an R expression such that errors are caught rather than causing a long jump. Again, the full high-level API can be browsed using the `cargo::api_documentation` function or by executing '`cargo doc --open`' when in the 'src/rustlib/roxido' directory.

Miscellaneous: Threading issues and seeding a RNG

Rust supports "fearless concurrency," making it safe and easy to harness the power of multiple CPU cores. One should bear in mind, however, that R's internals are fundamentally designed for single-threaded access. Any callbacks into R (using the low-level or high-level interface) should come from the same thread from which R originally called the Rust code.

R users expect to get reproducible results from simulation code when they use R's `set.seed` function. There are two options for Rust code to achieve this: (i) produce random numbers using R's API (as in the previous `myrnorm` example) or (ii) seed a Rust random number generator from R's random number generator. To aid with the second approach, the **cargo** framework provides the `random_bytes` function.

Embedding Rust code in an R script

Beyond package development, the **cargo** package also supports defining functions by embedding Rust code directly in an R script. This facilitates experimentation and avoids the need to set up a new R package. The approach, however, loses the developer aids of an integrated development environment. As such, it is only recommended to use this for small code snippets. To demonstrate, consider the balanced linear assignment problem, a combinatorial optimization problem in which N workers are assigned to N tasks such that the sum of costs of getting all tasks completed is minimized. Suppose there are four workers and tasks and the cost matrix in R is as follows, with each row being the costs of the four tasks for a particular worker.

```
cost_matrix <- matrix(c(
    5, 9, 4, 6,
    8, 7, 8, 6,
    6, 7, 9, 3,
    2, 3, 3, 1
), nrow=4, byrow=TRUE)
```

The Hungarian algorithm (Kuhn, 1955) solves the linear assignment problem and is implemented in the **RcppHungarian** package (Silverman, 2019) on CRAN. The Jonker-Volgenant algorithm (Jonker and Volgenant, 1987), however, is faster and available in the lapjv Rust crate (Dmytrenko, 2020). The following code uses the `rust_fn` from the **cargo** package to define an R function based on embedded Rust code utilizing the lapjv crate.

```
library("cargo")
lapjv <- rust_fn(weights, dependencies='lapjv = "0.2.0"', '
    if !weights.is_square_matrix() || !weights.is_double_or_integer() {
        panic!("The weights argument must be a square numeric matrix.");
    }
    let weights_vec = weights.coerce_double(&mut pc).unwrap().1.to_vec();
    let n = weights.nrow();
    let weights = lapjv::Matrix::from_shape_vec((n, n), weights_vec).unwrap();
    let solution = lapjv::lapjv(&weights).unwrap().0;
    let cost = lapjv::cost(&weights, &solution[..]);
    let (pairs, slice) = Rval::new_matrix_integer(n, 2, &mut pc);
    for (i, x) in slice[..n].iter_mut().enumerate() { *x = i as i32 + 1; }
    let s = &mut slice[n..];
    for (i, y) in solution.into_iter().enumerate() { s[y] = i as i32 + 1; }
    let result = Rval::new_list(2, &mut pc);
    result.names_gets(Rval::new(["cost", "pairs"], &mut pc));
    result.set_list_element(0, Rval::new(cost, &mut pc));
    result.set_list_element(1, pairs);
    result
')
lapjv(cost_matrix)
```

The `lapjv` function takes one unnamed argument `weights`, which is passed to the embedded Rust code as the variable `weights` of type `Rval`. This code depends on version 0.2.0 of the lapjv crate and is automatically downloaded and compiled by Cargo because of the argument `dependencies='lapjv = "0.2.0"'` in the call to the `rust_fn` function. Downloading and compiling the dependencies can take several seconds, but subsequent compilations are very fast due to caching. For example, on our machine, the first compilation took 12.95 CPU seconds and 6.57 elapsed seconds, whereas recompilation of slightly changed code only took 1.81 CPU seconds and 0.97 elapsed seconds. This caching persists between R sessions. When a function defined by `rust_fn` is garbage collected, its associated shared library is automatically unloaded.

Running the code produces a list giving the total cost and a matrix which pairs each worker to a task. Note that when the cost matrix is 1000×1000 of standard normal values, this implementation takes only 0.068 seconds whereas the **RcppHungarian** package finds the same solution in 4.866 seconds, i.e., 70 times slower. The point is not that C++ is slower than Rust, rather that the choice of algorithms can be important and that the **cargo** package makes it easy to pull in high quality Rust code from others with little effort.

Benchmarks

Here the overhead of calling a Rust function from R using our **cargo** framework is investigated. Benchmarks are shown against the **rextendr** framework and the standard mechanism for calling a C function from R. For this benchmark, we use version 0.1.37 of **cargo** and version 0.2.0 of **rextendr**, running in R version 4.1.0. An algorithm that executes quickly is purposefully used to benchmark the overhead of calling into and returning from compiled code. Rust and C themselves are not benchmarked here, but the reader is referred to The Computer Language Benchmarks Game (https://benchmarksgame-team.pages.debian.net/benchmarksgame/), which shows Rust beating GCC's C in about half of the benchmarks.

Consider various implementations to compute the Euclidean norm `sqrt(sum(x^2))`. Several versions based on **rextendr** are provided to account for the following: (i) **rextendr** can automatically convert `Robj` (its wrapper over an `SEXP` value) and many Rust types, and (ii) when defining embedded functions, **rextendr** does not cache the lookup of the function pointer.

```
writeLines(con="f_C.c", "
    #include <Rinternals.h>
    SEXP f_C(SEXP x) {
        int n = Rf_length(x); double *y = REAL(x); double ss = 0.0;
        for ( int i=0; i<n; i++ ) ss += y[i];
        return Rf_ScalarReal(sqrt(ss));
    }")
system("R CMD SHLIB f_C.c")
dyn.load("f_C.so")
.f_C <- getNativeSymbolInfo("f_C", "f_C")$address
f_C <- function(x) .Call(.f_C, x)

f_cargo <- cargo::rust_fn(x, '
    let ss = x.slice_double().unwrap().iter().fold(0.0, |s,z| s + (*z)*(*z));
    Rval::new(ss.sqrt(), &mut pc)
')

rextendr::rust_function('fn f_rextendr1(x: Robj) -> Robj {
    let ss = x.as_real_slice().unwrap().iter().fold(0.0, |s,z| s + (*z)*(*z));
    Robj::from(ss.sqrt())
}')

rextendr::rust_function('fn f_rextendr2(x: &[f64]) -> f64 {
    let ss = x.iter().fold(0.0, |s,z| s + (*z)*(*z));
    ss.sqrt()
}')

.f1 <- getNativeSymbolInfo("wrap__f_rextendr1", "librextendr1")$address
f_rextendr1_cached <- function(x) .Call(.f1, x)

.f2 <- getNativeSymbolInfo("wrap__f_rextendr2", "librextendr2")$address
f_rextendr2_cached <- function(x) .Call(.f2, x)

x <- rnorm(10)
microbenchmark::microbenchmark(f_C(x), f_cargo(x), f_rextendr1(x), f_rextendr2(x),
    f_rextendr1_cached(x), f_rextendr2_cached(x), times=1000000)
```

A summary of the performance is included below. Notice that the implementation based on **cargo** is competitive with the C version and faster than the **rextendr** implementations.

```
Unit: nanoseconds
                   expr  min   lq      mean median   uq      max neval
                 f_C(x)  349  504  616.3403    545  597  2329566 1e+06
              f_cargo(x)  389  523  684.6221    563  625 54627019 1e+06
          f_rextendr1(x) 4918 5818 6282.8506   6011 6226  3925556 1e+06
          f_rextendr2(x) 3789 4504 4941.8917   4707 4918  6396105 1e+06
 f_rextendr1_cached(x) 4244 5197 5606.6939   5391 5586  6669919 1e+06
 f_rextendr2_cached(x) 3145 3849 4256.3246   4070 4278  6602903 1e+06
```

Summary

The hope is that this paper contributes to interest in developing R packages with Rust. The paper highlights idiosyncrasies of R and Rust that must be addressed by any integration. The **cargo** framework provides a Rust interface for commonly used parts of the R API that can easily be extended to cover other parts of the R API. The framework minimizes the runtime overhead and seeks to be transparent on how it interfaces R and Rust.

Bibliography

D. B. Dahl. *cargo: Run Cargo, the Rust Package Manager*, 2021. URL https://CRAN.R-project.org/package=cargo. R package version 0.1.37. [p1]

D. B. Dahl, J. Andros, and J. B. Carter. *caviarpd: Cluster Analysis via Random Partition Distributions*. URL https://CRAN.R-project.org/package=caviarpd. R package version 0.2.17. [p2]

D. B. Dahl, D. J. Johnson, and P. Müller. *salso: Search Algorithms and Loss Functions for Bayesian Clustering*, 2021. URL https://CRAN.R-project.org/package=salso. R package version 0.2.23. [p2]

A. Dmytrenko. *Linear Assignmment Problem solver using Jonker-Volgenant algorithm*, 2020. URL https://crates.io/crates/lapjv. Rust crate version 0.2.0. [p7]

R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987. [p7]

B. W. Kernighan and D. M. Ritchie. *The C programming language*. 2006. [p1]

H. W. Kuhn. The Hungarian method for the assignment problem. *Naval research logistics quarterly*, 2 (1-2):83–97, 1955. [p7]

J. Ooms. *gifski: Highest Quality GIF Encoder*, 2021. URL https://CRAN.R-project.org/package=gifski. R package version 1.4.3-1. [p2]

A. Pelletier, V. Andréassian, and O. Delaigue. *baseflow: Computes Hydrograph Separation*, 2021. URL https://CRAN.r-project.org/package=baseflow. R package version 0.13.2. [p2]

J. Silverman. *RcppHungarian: Solves Minimum Cost Bipartite Matching Problems*, 2019. URL https://CRAN.R-project.org/package=RcppHungarian. R package version 0.1. [p7]

C. O. Wilke, A. Thomason, M. M. Reimert, I. Kosenkov, H. Yutani, and M. Barrett. *rextendr: Call Rust Code from R using the 'extendr' Crate*, 2021. URL https://CRAN.R-project.org/package=rextendr. R package version 0.2.0. [p2]

H. Yutani. *string2path: Rendering Font into 'data.frame'*, 2021. URL https://CRAN.R-project.org/package=string2path. R package version 0.0.2. [p2]

David B. Dahl
Brigham Young University
Provo, Utah
University States of America
(ORCiD: 0000-0002-8173-1547)
dahl@stat.byu.edu