Writing R Extensions in Rust

by David B. Dahl *[Version 0.2.8]*

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, an unofficial 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, at least, waste time of CRAN maintainers and package contributors.

The paper introduces the **cargo** (Dahl, 2022) 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. The **cargo** package produces an R package structure with the necessary Rust code and scripts. Developers can extend the framework for their own purposes within the generated package structure. Although a source package will obviously depend on Rust, there are no runtime dependencies on Rust, 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 has been submitted for developing subsystems and drivers for the Linux kernel. 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, 2022), 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., 2022b) 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. 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., 2022a) 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, alleviating 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 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 aims to be more comprehensive to provide many high-level conveniences and behind-the-scenes type conversions. 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 the **cargo** package from CRAN using `install.packages("cargo")`. There are are a variety of ways to install Cargo, but we recommend using `cargo::install()` in an interactive session. This function downloads the rustup (`https://rustup.rs`) tool chain installer and runs it. Rust has a six-week release cycle and, if it's installed through `cargo::install()`, the Rust toolchain is automatically updated as needed. 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 particular CRAN server may not have Rust installed or the Rust version may not be recent enough. The original Rust-based packages on CRAN downloaded precompiled static libraries when compiling source packages if a sufficient version of the Rust toolchain could not be found. With a recent revision to CRAN policies, however, this approach is no longer allowed. As such, if a sufficient version cannot be found when compiling the source package, our Cargo Framework embeds the Rust source code in the resulting binary package. The `.onLoad` and `.onAttach` functions of the package can then compile the Rust code if Cargo is available on the end user's machine, or ask the end user to allow it to install Cargo and then compile the Rust code.

One wrinkle with this approach, however, is that examples involving Rust are not guaranteed to run without user intervention if Cargo was not available when installing the source package. Our solution is to conditionally wrap Rust-based R examples in `\dontrun{}`. Consider, for example, the following example for an R function `calculate` that involves calling Rust code.

```
# R_CARGO \dontrun{
# R_CARGO # Example disabled since Cargo not found when installing source package.
# R_CARGO # You can still run this if you install Cargo. Hint: cargo::install().
calculate(x)
# R_CARGO }
```

When configuring the source package for installation, lines starting with "`# R_CARGO`" are deleted if Cargo is available, otherwise only "`# R_CARGO `" is deleted, leaving (for example) `\dontrun{`.

Finally, we emphasizes that, if the computer that compiles the source package (e.g., a CRAN server) has Cargo whose minimum version meets the needs specified in the package's 'DESCRIPTION' file, the end user does not need to install Cargo themselves and the examples do not need to be wrapped in `\dontrun{}`. Again, the easiest way to provide Cargo is to run cargo::install().

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" -> "Cargo Framework: Create R package using Rust (unofficial)". 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
├── configure
├── configure.win
├── DESCRIPTION
├── INSTALL
├── LICENSE
├── NAMESPACE
├── #SRC#
├── R
│   ├── convolve2.R
│   ├── myrnorm.R
│   ├── onLoad.R
│   ├── useDynLib.R
│   └── zero.R
├── man ...
├── src
│   ├── Makevars
│   ├── Makevars.win
│   ├── rust
│   │   ├── Cargo.toml
│   │   ├── roxido ...
│   │   ├── roxido_macro ...
│   │   ├── src
│   │   │   ├── lib.rs
```

```
│   │   │   └── registration.rs
│   │   └── vendor.tar.xz
│   └── shim.c
└── tools
    └── registration.R
```

Several of the resulting files and directories are specific to packages developed with the Cargo Framework. The 'configure' and 'configure.win' files direct R to use the 'tools/configure.R' script to compile the static Rust library defined in 'src/rust' or, as a fallback, to package the Rust code for inclusion in the binary package. Notice that the 'DESCRIPTION' file has an entry 'SystemRequirements: Cargo (>= 1.60) 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/configure.R' script uses `cargo::run` to find and run the Cargo package manager according to CRAN policies by, for example, using no more than two CPU threads. Rather than having Cargo automatically download dependencies (which is its typical behavior), package developers should provided them as part of the package source so as to comply with CRAN's policy that "packages should not write ... on the file system". Vendoring dependencies is accomplished by running `cargo::prebuild(what="vendor")`. Note that other Rust-based packages *not* using the Cargo Framework may still find the `cargo::run` function helpful in their own packages.

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/rust/src/lib.rs':

```rust
mod registration;
use roxido::*;

#[roxido]
fn myrnorm(n: Rval, mean: Rval, sd: Rval) -> Rval {
    unsafe {
        use rbindings::*;
        use std::convert::TryFrom;
        let (mean, sd) = (Rf_asReal(mean.0), Rf_asReal(sd.0));
        let length = isize::try_from(Rf_asInteger(n.0)).unwrap();
        let vec = Rf_protect(Rf_allocVector(REALSXP, length));
        let slice = Rval(vec).slice_mut_double().unwrap();
        GetRNGstate();
        for x in slice { *x = Rf_rnorm(mean, sd); }
        PutRNGstate();
        Rf_unprotect(1);
        Rval(vec)
    }
}
```

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/rust/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/rust/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` which takes 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::prebuild("/path/to/package/foo")`. This automatically regenerates the 'src/rust/src/registration.rs' file and does the following three 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/rust/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/rust/src/registration.rs' for examples on calling R's API from Rust.) Third, it rebuilds R documentation if the package's 'DESCRIPTION' file indicates that the package uses **roxygen2**.

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. 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/rust/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 has 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/rust/src/lib.rs' and shown below.

```
21      #[roxido]
22      fn convolve2(a: Rval, b: Rval) -> Rval {
23          let (a, xa) = a.coerce_double(pc).unwrap();
24          let (b, xb) = b.coerce_double(pc).unwrap();
25          let (ab, xab) = Rval::new_vector_double(a.len() + b.len() - 1, 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/rust/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 37, 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 `pc`. The wrapper code provided by the `#[roxido]` attribute includes `let pc = &mut 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 instead of by 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/rust/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", pc);
41      let mut feval = |x: f64| {
42          symbol.assign(rval!(x), rho);
43          f.eval(rho, pc).unwrap().as_f64()
44      };
45      let mut f0 = feval(x0);
46      if f0 == 0.0 { return rval!(x0); }
47      let f1 = feval(x1);
48      if f1 == 0.0 { return rval!(x1); }
49      if f0 * f1 > 0.0 { panic!("x[0] and x[1] have the same sign"); }
50      loop {
51          let xc = 0.5 * (x0 + x1);
52          if (x0 - x1).abs() < tol { return rval!(xc); }
53          let fc = feval(xc);
54          if fc == 0.0 { return rval!(xc); }
55          if f0 * fc > 0.0 { x0 = xc; f0 = fc; } else { x1 = xc; }
56      }
57  }
```

This example shows the creation of new R objects from Rust values (e.g., lines 40, 42, 46, etc.) and extracting Rust values from R objects (e.g., 37, 38, and 43). Note that `rval!(x)` is the macro equivalent to `Rval::new(x,pc)`. Line 43 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.

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 provided by 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, 2022) 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 = "0.2.1"', '
    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(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, 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, pc);
    result.names_gets(rval!(["cost", "pairs"]));
    result.set_list_element(0, rval!(cost));
    result.set_list_element(1, pairs);
    result
')
cost_matrix <- matrix(c(5,9,4,6,8,7,8,6,6,7,9,3,2,3,3,1), nrow=4, byrow=TRUE)
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 0.2.1 of the lapjv crate and is automatically downloaded and compiled by Cargo because of the argument `dependencies='lapjv = "0.2.1"'` 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 took only 0.068 seconds whereas the **RcppHungarian** package found the same solution in 4.866 seconds, i.e., 70 times slower. The claim is not that C++ is slower than Rust, rather that the choice of algorithms can be important and that our **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** approach and the standard mechanism for calling a C function from R. For this benchmark, we use version 0.2.8 of **cargo** and version 0.2.0 of **rextendr**, running in R version 4.2.1. 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] * 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!(ss.sqrt())
')

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())
}', profile="release")

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

.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   lq median   uq neval
                  f_C(x)  425    454  483 1e+06
              f_cargo(x)  491    526  557 1e+06
          f_rextendr1(x) 1368   1426 1507 1e+06
          f_rextendr2(x) 1279   1336 1415 1e+06
   f_rextendr1_cached(x)  822    875  945 1e+06
   f_rextendr2_cached(x)  737    788  858 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*, 2022. URL https://CRAN.R-project.org/package=cargo. R package version 0.2.8. [p1]

D. B. Dahl, J. Andros, and J. B. Carter. *caviarpd: Cluster Analysis via Random Partition Distributions*, 2022a. URL https://CRAN.R-project.org/package=caviarpd. R package version 0.2.31. [p2]

D. B. Dahl, D. J. Johnson, and P. Müller. *salso: Search Algorithms and Loss Functions for Bayesian Clustering*, 2022b. URL https://CRAN.R-project.org/package=salso. R package version 0.3.6. [p2]

A. Dmytrenko. *Linear Assignmment Problem solver using Jonker-Volgenant algorithm*, 2020. URL https://crates.io/crates/lapjv. Rust crate version 0.2.1. [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*, 2022. URL https://CRAN.R-project.org/package=gifski. R package version 1.6.6-1. [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*, 2022. URL https://CRAN.R-project.org/package=RcppHungarian. R package version 0.2. [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 0.0.4. [p2]

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