Interfacing an existing software library to R: The example of "RNiftyReg"

Jon Clayden <j.clayden@ucl.ac.uk>

LondonR meeting, 2011-06-15

Health warning

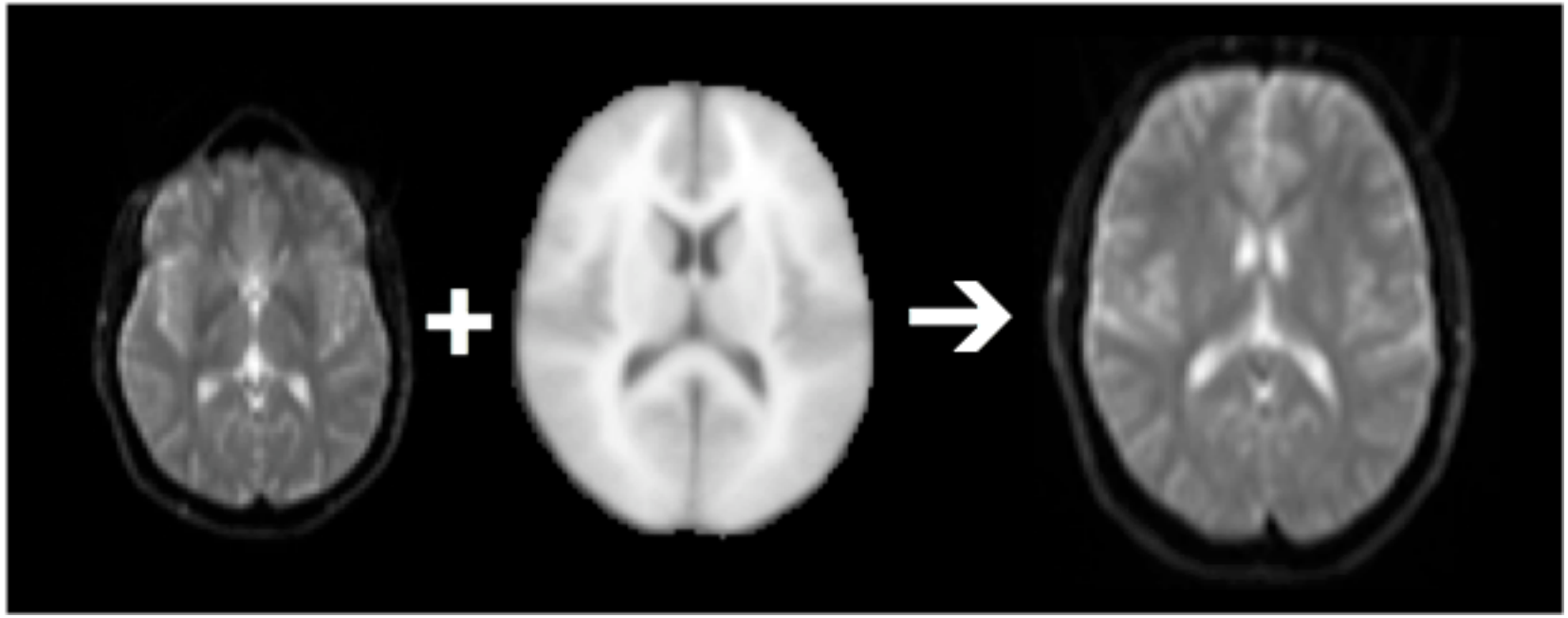
- I will only talk about interfacing C/C++ code
- I try to follow R best practices, but I may be doing something "wrong"!
- What follows is my perspective, and your milage may vary
- Always a good idea to consult *Writing R Extensions* if in doubt (http://cran.r-project.org/doc/manuals/R-exts.html)

A bit of context

- Image registration, or coalignment, is the process of finding an optimal transformation to align one (2D or 3D) image with another
- A key process in medical image processing pipelines, as well as microscopy, etc.
- The process is an optimisation over some space of transformations, followed by application of the final transformation and regridding (including interpolation)
- Typical 3D magnetic resonance image has ~1M voxels (3D pixels) so optimisation needs to be done cleverly to keep run time reasonable
- Registration is a common step, so needs to be as fast as possible
- Interpreted R code is not going to cut it

Illustration

- Typically need to allow for (at least) rotation and translation between source and target image
- A scale and shear are additionally allowed by affine transformations



The starting point

- Nifty Reg, an open-source registration toolkit developed at UCL (M. Modat *et al.*; http://sourceforge.net/projects/niftyreg/; BSD licence)
- Written in C/C++
- Pretty clean separation between “work” functions and user interface code

Key tasks

- Strip out command-line interface and other unneeded code
- Convert R data structures to C data structures
- Call the “work” functions and convert the result back to the form R expects
- Write R wrapper function(s)
- Write auxiliary/convenience R functions
- Create Makevars, configure script, etc., if needed

.C vs .Call/.External

- .C is generally simpler, avoids need to deal with R-specific functions and types (e.g. SEXP) in C
- .C duplicates its arguments by default; .Call and .External do not
- Handling R objects and functions with .C is clunky; .Call or .External is generally preferable
- .Call arguments should be considered read-only; .C arguments can be read/write
- I recommend .C unless you need to carry over large or complex R objects (medical images are both large and complex, so RNiftyReg uses .Call)

Data structures

- In R, an image is an S4 object of class “nifti”, defined in the “oro.nifti” package (B. Whitcher *et al.*; http://cran.r-project.org/web/packages/oro.nifti/; BSD licence)
- In C, an image is a struct
- Conversion is tedious but important

```
nifti_image * s4_image_to_struct (SEXP object)
{
    int i;
    nifti_1_header header;

    header.sizeof_hdr = 348;

    for (i = 0; i < 8; i++)
        header.dim[i] = (short) INTEGER(GET_SLOT(object,
install("dim_")))[i];

    header.intent_p1 = (float) *(REAL(GET_SLOT(object,
install("intent_p1"))));
    ...
```

Wrapper function

- Checks data types; deals with default cases; handles metadata and ensures that return value is the correct form

```
niftyreg <- function (source, target, targetMask = NULL, initAffine = NULL, scope =
c("affine","rigid"), nLevels = 3, maxIterations = 5, useBlockPercentage = 50,
finalInterpolation = 3, verbose = FALSE)
{
    if (!require("oro.nifti"))
        stop("The \"oro.nifti\" package is required")
    if (missing(source) || missing(target))
        stop("Source and target images must be given")
    ...
    if (source@dim_[1] == target@dim_[1])
    {
        returnValue <- .Call("reg_aladin", .fixTypes(source), .fixTypes(target),
scope, as.integer(nLevels), as.integer(maxIterations), as.integer
(useBlockPercentage), as.integer(finalInterpolation), .fixTypes(targetMask),
initAffine, as.integer(verbose), PACKAGE="RNiftyReg")

        dim(returnValue[[1]]) <- dim(target)
        dim(returnValue[[2]]) <- c(4,4)
        attr(returnValue[[2]], "affineType") <- "niftyreg"
        ...
```

Summary of process

- Source and target images are read from file or otherwise created using functions in the “oro.nifti” package, or similar
- These are passed to niftyreg(), which will do various checks to ensure that its parameters are sensible and then pass them to C
- In C, images are converted to the data type which the Nifty Reg functions expect
- The registration is performed
- A data vector containing the registered image data, and the transformation matrix, are passed back to R
- The niftyreg() function formats the result, applies the relevant metadata, and returns

Making life easier

- Unless every scrap of speed is important, do type checking (and other “cheap” tasks) in the R wrapper function
- Don’t expose C parameters as R wrapper function arguments if they can be worked out (e.g. vector lengths)
- Remember that you can step through compiled code using

  ```
  R -d gdb
  ```

  (or other debugger, as appropriate), and then loading package and setting a breakpoint
- Look out for use of functions which will cause problems with R (e.g. exit(), which will quit the whole R process)
- Porting memory allocation, output and other functions to the R-specific equivalents (e.g. R_alloc()) is generally worthwhile

RNiftyReg

- Provides 2D and 3D rigid-body and affine registration, including 3D-to-2D and 4D-to-3D
- Includes auxiliary functions for reading, writing and manipulating affine transformations
- Plans to interface to Nifty Reg's nonlinear registration algorithm at some point in the future
- Available on CRAN, or at http://code.google.com/p/rniftyreg/

Conclusions

- Interfacing R to an existing software library requires some work, but needn't be too painful (initial release of RNiftyReg was days' work, not weeks', even with very little experience)
- (If the library changes a lot and/or separation between "work" functions and user interface isn't clear, it might be harder)
- Almost certainly faster than developing a new implementation from scratch in most cases
- Converting between data types can be time consuming, but for simple vectors it's much easier
- R really does provide a lot of facilities to help
- Remember, packages provide interfaces to languages other than C and Fortran