Lab: Implementing a simple (but fast) Position Weight Matrix matching algorithm for long DNA sequences

Hervé Pagès Martin Morgan

April 18, 2008

1 Goals

- Create a fully working package (*simpleMatchPWM*) around a simple PWM matching function (`matchPWM`) and some related functions (see below).
- Write the core algorithms in C (for speed purpose) and use the `.Call` interface to call them from the R code.
- Make our `matchPWM` function work on long DNA sequences like full chromosome sequences. The standard container for such sequences in Bioconductor is the `DNAString` container defined in the *Biostrings* package. This will require us to learn a little bit about the *Biostrings* internals and the use of the *Biostrings C interface* but an attempt has been made to keep this as simple as possible.

2 Prerequisites

- A laptop with a recent version of R 2.7 and all the tools required for package development (compilers, Perl, etc...)
- The latest version of the *Biostrings* package from the 2.7 series (the current development series). Note that this document reflects the state of affairs in *Biostrings* 2.7.46 and is guaranteed to be accurate for this version only.
- The *BSgenome.Dmelanogaster.FlyBase.r51* package.
- Some basic knowledge of R package authoring.
- Some C knowledge and previous experience in C programming.
- You should have received 2 package source tarballs: `simpleMatchPWM.stub_-0.0.2.tar.gz` and `simpleMatchPWM_0.99.1.tar.gz`. The former is an incomplete package (it doesn't pass `R CMD check`): you will add the code

that is missing in it. The latter is the fully working version of the former (it passes `R CMD check` without any errors or warnings). Note that these packages, like this document, reflect the state of affairs in *Biostrings* 2.7.46.

3 Checking your installation

Exercise 1
Extract the `simpleMatchPWM.stub_0.0.2.tar.gz` tarball somewhere.

Now look what's under the newly created `simpleMatchPWM.stub` folder: you'll find the layout of files and subfolders that you've already seen in most R packages. Except maybe for the `src` subfolder: this is the standard folder where to put files containing native code (`.c`, `.h`, `.cpp`, `.f` files, etc...). In the *simpleMatchPWM.stub* package, `src` contains only 2 files: `matchPWM.c` and `Biostrings_stubs.c`.

You could have as many files as you want in `src`, and you could mix files written in different languages (proper extensions must be used). Note that, most of the times, you don't need a `Makefile` or any other extra file: by default `R CMD INSTALL` knows what to do with them. First it will invoke the compiler on each of them (skipping the header files (`.h`) and anything else that does not require compilation), then it will link together all the `.o` files produced by the individual compilations into a shared object (`.so` on Unix/Linux/Mac and `.DLL` on Windows).

Exercise 2
Run `R CMD INSTALL` on the `simpleMatchPWM.stub` folder and look at how the compiler is invoked.

What compiler flags are used?

Note that `R CMD INSTALL` doesn't clean after itself: can you see the compilation products in `src`?

On Unix/Linux/Mac, if you are using the gcc compiler, you can turn on the -Wall flag (in order to get warnings about potential problems in your code). This is done by editing `$R_HOME/etc/Makeconf` (e.g. append `-Wall` to the `CFLAGS` line and you'll get the warnings during the compilation of C files).

Exercise 3
Turn on the -Wall gcc flag and run `R CMD INSTALL` again (you should see 2 warnings). Are those warnings really serious?

4 Is folder `src` all I need in order to support native code in my package?

Basically yes. But in the case of the *simpleMatchPWM.stub* package, since it has a namespace, then the following line needs to be added to its `NAMESPACE`

file (generally to the top):

```
useDynLib(simpleMatchPWM.stub)
```

Exercise 4
Check simpleMatchPWM.stub's `NAMESPACE` *file.*

5 Trying to use *simpleMatchPWM.stub*

Exercise 5
Start R, load the simpleMatchPWM.stub package, and look at the man page for the `matchPWM` *function (*`?matchPWM`*).*

Try to run the examples. Anything wrong?

Load the BSgenome.Dmelanogaster.FlyBase.r51 package and display the `Dmelanogaster` *object. Display chromosome 3R. What's its length? What's the class of this object?*

Use `alphabetFrequency` *(from the Biostrings package) on it. Are there any other letters than A, C, G or T in this sequence?*

What about chromosome 3L?

6 A first example using .Call

Before we try to implement the `PWMscore`, `matchPWM` and `countPWM` functions, let's go thru the classic *Hello World* exercise. We'll start by adding a little function to the `src/matchPWM.c` file that prints `Hello world!`. Then we'll take any required step to make this C function callable from R via the .Call interface. In other words, we want to make our C function a *.Call entry point*.

Exercise 6
Open the `src/matchPWM.c` *file and add the* `hello_world` *function near the bottom of the file, right before the section called REGISTRATION OF THE .Call ENTRY POINTS.*

For any *.Call entry point*, the arguments and the returned value must be `SEXP` objects. In the case of the `hello_world` function, we don't need any argument, which is fine, but we must return an `SEXP` object. A common solution is to return `NULL_USER_OBJECT` (symbol defined in `$R_HOME/include/Rdefines.h`) which is the `SEXP` object representing the `NULL` value in R.

Exercise 7
Make `hello_world` *return* `NULL_USER_OBJECT`.

Save, reinstall simpleMatchPWM.stub, restart R, load simpleMatchPWM.stub and try:

```
> .Call("hello_world", PACKAGE = "simpleMatchPWM.stub")
```

The above didn't work because all *.Call entry points* must be registered. This is done by adding an entry for `hello_world` to the `callMethods` array defined at the bottom of the file. Note that the last value of each entry must be the number of arguments of the *.Call entry point*.

Exercise 8
Register `hello_world` *and try to call it again from R as before.*

7 Manipulating *DNAString* objects in C

All the R code in *simpleMatchPWM.stub* has been put in a single file, the `R/matchPWM.R` file.

Exercise 9
Open `R/matchPWM.R` *and find the definition of the* `PWMscore` *function.*

What C function does it call? How many arguments are passed to this C function?

Find the definition of the `PWM_score()` *entry point in* `src/matchPWM.c`.

The 2nd argument of *.Call entry points* `PWM_score()` and `match_PWM()` must be a *DNAString* object. This is enforced at the R level: the callers will check the nature of their `subject` argument and raise an error if it's not a *DNAString* object. Note that this kind of sanity checking could be done at the C level but they are generally much easier to do (and to read, understand and modify) at the R level.

You don't need to know all the details about the *DNAString* class in order to manipulate a *DNAString* object at the C level. Most of the time, all you need to know is the address in memory of its first letter and its length. This information can be retrieved by calling the `get_XString_asRoSeq` function. This function is part of the *Biostrings C interface* which will be introduced now.

8 The Biostrings C interface

The *Biostrings C interface* is defined in the `inst/include/Biostrings_interface.h` file of the *Biostrings* package. By default all R packages are installed under `$R_HOME/library` (see `?install.packages` for more information on this), so `Biostrings_interface.h` should be located in `$R_HOME/library/Biostrings/inst/include/`.

Exercise 10
Consult the `Biostrings_interface.h` *file for more information about the Biostrings C interface.*

In particular, check that the simpleMatchPWM.stub package is set up properly:

- *check the* `Depends:`, `Imports:` *and* `LinkingTo:` *fields in the* `DESCRIPTION` *file;*

- *check the* `import` *directives in the* `NAMESPACE` *file;*
- *check the* `src/Biostrings_stubs.c` *file;*
- *check the* `#include "Biostrings_interface.h"` *line in* `src/matchPWM.c`.

Exercise 11
Copy/paste the definition of the `print_XString_bytes` *function given in* `Biostrings_interface.h` *into your* `src/matchPWM.c` *file, and register it as a .Call entry point. Save, reinstall simpleMatchPWM.stub, restart R, load simpleMatchPWM.stub and use* `.Call` *to call* `print_XString_bytes` *on a DNAString object.*

In fact, *DNAString* objects, like *RNAString*, *AAString* and *BString* objects, are particular kinds of *XString* objects. The `get_XString_asRoSeq` function can be used on any of them.

Exercise 12
Call `print_XString_bytes` *on* `DNAString("ACGTacgt")`, `RNAString("ACGUacgu")` *and* `BString("ACGTacgt")`.

As stated in `Biostrings_interface.h`, there are 2 important things to remember about *XString* objects:

- they are *not* null-terminated like standard strings in C: they can eventually contain the null byte so you should never use the C standard string functions on them;
- *DNAString* and *RNAString* objects have their data *encoded*. This means that a code (different from the ASCII code) is used to represent each nucleotide internally.

Exercise 13
Modify the `print_XString_bytes` *function to make it display (in clear) the nucleotide letters contained in a DNAString object. Try to use this modified version on a BString object.*

9 Specifications of the PWM_score() function

Now we are almost ready to implement `PWM_score()`. But before we start, we need to describe *exactly* what this function will do.

We start by describing `PWM_score()`'s arguments:

- `pwm`: the Position Weight Matrix (PWM). This is an integer matrix with row names A, C, G and T (in this order);
- `subject`: a *DNAString* object containing the subject (or target) sequence;
- `start`: the set of starting positions. This is an integer vector of arbitrary length (`NA`s accepted).

Now what the function will do: given a PWM, a DNA sequence and a set of starting positions, `PWM_score()` must walk thru the set of starting positions, and, for each of them, *place* the PWM such that its first column is aligned with the current starting position and compute a score. For a given starting position, the score is obtained by picking, in each column of the PWM, the coefficient that corresponds to the nucleotide in the DNA sequence that is aligned with the column, and by summing them.

Type of the returned value: given that the `pwm` argument will always be a matrix with integer coefficients (see the caller, `PWMscore`, in `R/matchPWM.R`), then the score can only be an integer too. Hence we want `PWM_score()` to return an `SEXP` object representing an integer vector.

Vectorization: the `PWM_score()` function is vectorized in respect to the `start` argument. This means that applying the function to the `start` vector is equivalent to applying the same function on each of its elements. Also, as it is usually the case with vectorized functions, we want to allow `NA` values in the `start` object: when a starting position is `NA`, then the corresponding score must be `NA` too.

Finally, we want to raise an error if one of the starting positions is *invalid*. When the first column of the PWM is aligned with the starting position, then the entire PWM should fit within the limits of the DNA sequence, otherwise, the starting position is considered *invalid*.

10 Some notes about the compute_score() helper function

The `compute_score()` helper function is provided so you can use it any time you need to compute the score for a given starting position. This will make implementing `PWM_score()` and `match_PWM()` easier.

The arguments of `compute_score()` are:

- `pwm`: a pointer to the first coefficient of the PWM. In R, a matrix (like any array) is just an atomic vector with a `"dim"` attribute. This means that, at the C level, its coefficients are stored one next to each other in memory. Most importantly, they are stored *column by column*. For example, in the case of the PWM, the first column is stored in `pwm[0]`, `pwm[1]`, `pwm[2]`, `pwm[3]`, the second column in `pwm[4]`, `pwm[5]`, `pwm[6]`, `pwm[7]`, and so on...

- `pwm_ncol`: the number of column of the PWM. Hence the last valid element in `pwm[]` is `pwm[4*pwm_ncol-1]`.

- `S`: a pointer to the first letter in the target sequence.

- `nS`: the length of the target sequence.

- `pwm_shift`: how far the PWM must be shifted along the target. A shift of zero corresponds to a starting position of one and the shift is in fact always the starting position minus one.

Speed: inside `compute_score()`, everytime we move to the next letter in the target sequence (`S`), then we need not move to the next column in the PWM (`pwm`), we need to map this new letter to the corresponding row in the PWM. In order to make this as fast as possible, we use the translation table `DNAcode2PWMrowoffset`. This table must be initialized before `compute_score()` can be used. So don't forget to call `init_DNAcode2PWMrowoffset()` in `PWM_score()` and `match_PWM()` before they call `compute_score()` for the first time.

Exercise 14
Have a close look at `compute_score()` *and try to understand how it works.*

Note that we didn't make `compute_score()` a *.Call entry point* because we never need to call it directly from R. Even more, by declaring this function static (see the use of the `static` keyword at the beginning of the function definition), we restrict its use to the `matchPWM.c` file. That is, only functions defined in the same file can call it. In the case of the *simpleMatchPWM.stub* package, it doesn't really matter whether a C function is declared static or not, because all C functions are in the same file. But, when a program is made of many `.c` files, making some functions statics can make the code easier to maintain, and it helps to keep things organized.

11 Implementing the PWM_score() function

We are finally ready to implement `PWM_score()`!

Exercise 15
Follow the instructions given in the `PWM_score()` *stub to implement the function.*

Tips:

- *Apply* `INTEGER()` *to an* `SEXP` *object representing an integer vector in order to get the address of its first element (remember that in C, the index of the first element in an array is 0).*
- *Use* `PROTECT(ans = NEW_INTEGER(LENGTH(start)));` *in order to allocate memory for the answer object. Then, for example, to assign a value to its first element, use* `INTEGER(ans)[0]` *on the left side of the assignement.*
- *Every time you try to compile your code, try to decipher all compiler errors or warnings. They can sometimes be cryptic, but getting familiar with gcc's jargon becomes essential in troubleshooting times.*
- *If you get stuck, look at the simpleMatchPWM package.*

12 Implementing the match_PWM() function

The `match_PWM()` function must *find* and return all the starting positions in the target sequence that realize a score greater or equal to a given value (the *min score*).

We will use *brute force* for this, that is, we will try all valid starting positions in the target sequence and return only those that lead to a match. This means that our main loop in `match_PWM()` will walk the target sequence from its first letter to a letter close to its last letter (the last `pwm_ncol-1` letters in the target sequence are invalid starting positions). For example, with a chromosome sequence, we will typically have to examine millions of starting positions!

Exercise 16
Make sure you know exactly what the `match_PWM()` *function will do.*

Exercise 17
Follow the instructions given in the `match_PWM()` *stub to implement the function. Tips: you could start by ignoring the* `count_only` *argument and come back to it later.*

Look at the `matchPWM` *function in* `R/matchPWM.R` *and see how it converts the object returned by* `.Call` *into a BStringViews object.*

When you are done, try to run the examples in `?matchPWM`*. Is* `match_PWM()` *fast enough?*

13 Finishing your package

Exercise 18
Set the `Author:` *and* `Maintainer:` *fields in the* `DESCRIPTION` *file.*

Finally, make sure that your package passes `R CMD check` *without any errors or warnings.*

That's it!