Splitting your program into multiple files

It is a general practice to divide your code into different categories or files. This section will teach you how to do this correctly.

We'll cover the following

The programs we have seen so far have all been stored in a single source file. As your programs become larger, and as you start to deal with other people's code (e.g. other C libraries) you will have to deal with code that resides in multiple files. Indeed you may build up your own library of C functions and data structures, that you can re-use in your own scientific programming and data analysis.

Here we will see how to place C functions and data structures in their own file(s) and how to incorporate them into a new program.

We saw in the section on functions, that one way of including custom-written functions in your C code, is to simply place them in your main source file, above the declaration of the `main()` function.

A better way to re-use functions that you commonly incorporate into your C programs is to place them in their own file, and to include a statement above `main()` to **include** that file.

When compiled, it's just like copying and pasting the code above `main()`, but for the purpose of editing and writing your code, this allows you to keep things in separate files. It also means that if you ever decide to **change** one of those re-usable functions (for example if you find and fix an error) that you only have to

change it in **one place**, and you don't have to go searching through all of your programs and change each one.

Header files

A common convention in C programs is to write a header file (with **.h** suffix) for each source file (**.c** suffix) that you link to your main source code. The logic is that the .c source file contains all of the code and the header file contains the function **prototypes**, that is, just a declaration of which functions can be found in the source file.

This is done for libraries that are provided by others, sometimes only as compiled binary "blobs" (i.e. you can't look at the source code). Pairing them with plain-text header files allows you see what functions are defined, and what arguments they take (and return).

The #include statement

The `#include` statement is used to link a header file or a library with a source file. We've already seen a very prominent example of this.

`#include <stdio.h>` indicates that we want to include all the content of `stdio.h` into our source file. So, for any source file, the template for using the `#include` statement is:

`#include <file name>` or `#include "file name"`

An example

Here is a program that computes the preferred direction of a neuron recorded in primary motor cortex of rhesus macaques, during a whole-arm reaching task (e.g. from Gribble & Scott, 2002). The monkey moved his hand from a central start target to one of 8 peripheral targets around the circumference of a circle. Movements to each of the 8 targets were repeated 5 times for a total of 40 movements. The order of target directions was randomized.

The data for each neuron are represented by two sets of values: first, an array of 40 values that indicate for each movement, the **direction** of the movement (specifically, the direction, in radians of the velocity vector at peak hand velocity), and second, another array of 40 values that indicate for each movement, the mean number of **spikes** per second averaged over a window starting 150ms before

movement onset and ending at peak hand velocity.

For each neuron, the goal is to compute a **preferred direction**. That is, the direction of movement for which the neuron fires most enthusiastically (most spikes per second). The details of how these computations are done are not so important to consider here, but imagine for the moment that we have already written (or someone has provided us with) a C function called `compute_PD()` that performs this calculation, and that the code is stored in a file called `neuron.c`, with a header file `neuron.h`.

Here is our main program which is called `go.c`:

```
// gcc -std=c99 -o go go.c neuron.c -lm

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "neuron.h"

int main(int argc, char *argv[]) {

  int ncells = 100;                     // # cells to process
  char fnum[4], fname[128];              // filename strings
  double celldirs[40], cellspks[40];    // data for each cell
  double PD[ncells], plate_out[9];      // store cell PDs

  // loop to process each cell
  int i;
  for (i=1; i<=ncells; i++) {

    // construct the numeric part of the filename
    if (i<10) sprintf(fnum, "00%d", i);
    else if (i<100) sprintf(fnum, "0%d", i);
    else sprintf(fnum, "%d", i);

    // read in a dirs data file
    sprintf(fname, "data/cell_dirs_%s.txt", fnum);
    readcell(fname, celldirs, 40, 0);

    // read in a spks data file
    sprintf(fname, "data/cell_spks_%s.txt", fnum);
    readcell(fname, cellspks, 40, 0);

    // compute PD
    PD[i-1] = compute_PD(celldirs, cellspks, 40);
  }

  // print vector of PDs to screen and write to a file
  show_double_vec(PD, ncells);
  write_double_vec(PD, ncells, "PDs.txt");

  return 0;
}
```

go.c

What we can see is that on line 6, we use an `#include` statement to include the neuron.h header file (shown below). This is a way of essentially telling the compiler, that these functions (in neurons.h) will be described fully later ... and for now, "trust" that they are defined, and in this particular way (inputs and outputs).

On line 1 (commented out) I show the gcc command to compile this program, and you can see that we simple add neuron.c to the list of files to send to the compiler. This is where the compiler actually integrates the code in neuron.c into our program. We also need the `-lm` flag, to instruct the compiler to load the standard C math library (since the `#include <math.h>` directive appears in `neuron.h`).

Here is the header file `neuron.h`:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846264338327 95
#endif

void show_int_vec(int vec[], int n);

void show_double_vec(double vec[], int n);

void write_double_vec(double vec[], int n, char fname[]);

void showmat(double mat[], int nrows, int ncols);

void readcell(char fname[], double data[], int n, int msg);

void oneplate(double r1, double r2, double a1, double a2, double output[6]);

void platemethod(double a[], double r[], int n, double output[9]);

int mycomp(const void *a, const void *b);

double getangle(double x, double y);

int mycomp_struct(const void *a, const void *b);

double compute_PDr(double celldirs[], double cellspks[], int ntrials, int ndirs);

double compute_PD(double celldirs[], double cellspks[], int ntrials);
```

neuron.h

What you can see is that this is simply a listing of the functions that are defined in neuron.c, and we simply list the function prototypes, not the "meat" of the

functions themselves. When we write the #include “neuron.h” statement at the top of go.c (line 6), these function prototypes are loaded in, so that the code in the main program “knows about” these functions. Note that there are some functions in here that we do not use in the `go.c` code above.

An important point to remember is that as long as we can look at the header file, we have all the information about what functions are defined in the source file, and how they are used (what their input arguments are and what output arguments, if any, they return). If we want to look “inside” those functions, we can look at the source file. The header file then can be thought of as an **interface** between the source code and the programmer.

To summarize then, in order to include external code in your C program (code that is located in a separate file), you need to make sure two things happen:

1. The external C source code is passed to the compiler at compile time
2. Your own C program has access to the function prototypes associated with the external code

Another example

Another small example: let’s say you want to write a program that takes at the command-line one integer input, and determines whether that integer is a prime number or not. Now let’s say that you don’t want to write your own code for determining primality, so you ask your friend, who you know has written such a function already. She sends you a pair of files (`primes.h` and `primes.c`). Her header file (`primes.h`) looks like this:

```
int isPrime(int n); // returns 0 if n is not prime, 1 if n is prime
```

primes.h

So we know a couple of things from this header file. It declares a function prototype for `isPrime()` . We can see this function takes a single integer as an input argument, and returns an integer value: 0 if the input value is a prime number, and 0 if it is not. Now we know all we need to know in order to use this function (without even looking at the function’s source code, which resides in `primes.c`).

Here is what the `primes.c` file look like:

```
int isPrime(int n) {

  // returns 0 if not prime, 1 if prime

  if (n<2) return 0;         // first prime number is 2
  if (n==2) return 1;        // ensure 2 is identified as a prime
  if ((n % 2)==0) return 0;  // all even numbers above 2 are not prime

  int i;
  for (i=3; i*i < n; i++) {  // test divisibility up to sqrt(n)
    if ((n % i) == 0) {
      return 0;
    }
  }
  return 1;
}
```

primes.c

Here is what our program `go.c` looks like:

```
/* go.c
   Takes one input argument from the command line, an integer,
   and returns 1 if the number is prime, and 0 if it is not.
   Compile with: gcc -o go go.c primes.c
*/

#include <stdio.h>
#include <stdlib.h>
#include "primes.h"

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("error: must provide a single integer value to test\n");
    return 1;
  }
  else {
    int n = atoi(argv[1]);
    int prime = isPrime(n);
    printf("isPrime(%d) = %d\n", n, prime);
    return 0;
  }
}
```

go.c

Here is the result of running the program on a small selection of input values:

```
plg@wildebeest:~/Desktop$ gcc -o go go.c primes.c
plg@wildebeest:~/Desktop$ ./go 1
isPrime(1) = 0
plg@wildebeest:~/Desktop$ ./go 2
isPrime(2) = 1
plg@wildebeest:~/Desktop$ ./go 3
isPrime(3) = 1
plg@wildebeest:~/Desktop$ ./go 63
isPrime(63) = 0
```

```
plg@wildebeest:~/Desktop$ ./go 67
isPrime(67) = 1


plg@wildebeest:~/Desktop$ ./go 12347
isPrime(12347) = 1
```

Search path

The compiler will look in several places for header files that you include with the `#include` directive, depending on how you use it. If you use include with the angled brackets (e.g. `#include <stdio.h>`) then the compiler will look in a series of "default" system-wide locations (see Search Path for details). If you use double-quotes (e.g. #include "neuron.h") then the compiler will look in the directory containing the current file. It's possible to add other directories to the search path by using the `-Idir` compiler option, where `dir` is the other directory. You might have to do this if you link your code to an external C library that is not part of the standard C library, and does not reside in the usual "system" default locations.

External variables

Just as we can include external code in our C programs, we can make a declaration in our C program that a variable exists and has been declared elsewhere (e.g. in some other source file). This is done using the `extern` keyword. See here for more details.

In the next lesson, we'll examine a convenient feature of C know as the `make` utility.