FASTlib Design and Development Manual

version 0.1

Fundamental Algorithmic and Statistical Tools (FAST) Lab
Georgia Institute of Technology
Atlanta, GA

Contents

Chapter 1

Introduction

1.1 Overview

FASTlib is a library of numerical and machine learning methods written in C and C++. It provides:

1. templated classes for matrices and vectors and methods for operating over those.
2. a large number of functions for numerical linear algebra and other mathematical methods.
3. classes API's for machine learning methods so that these can be integrated into larger data analysis models and system implementations.
4. a set of tools written in Python to simplify the build setup and for automatic generation of makefiles etc.
5. a collection of machine learning tools as stand-alone executables that can be used for "out of box" analysis without the need to write any code.

Figure 1.1 shows the overall organization of FASTlib. FASTlib is designed with the goal of being:

- scalable - the code should be able to handle large-scale machine learning problems and programmers should have the ability to
- flexible - the design is modular so that only the core library is necessary and components can be plugged in as needed.
- easy to use - provides simple and consistent interface, a rich set of API, and rich set of compilation and debugging tools to enable ease of use.
- $\cdots$

The Core FASTlib library is meant to provide the basic tools needed to start hands-on with machine learning. Its design goals are to have a low learning curve, allow distributed development, be as fast as possible, yet avoid most of the worst problems of developing in a low-level language.

1.2 Background and Motivation

In the current state of affairs, the best implementation of each algorithm is typically in MATLAB or in a stand-alone C code. Neither situation is optimal in the grand scheme of things. Some things are very difficult to do efficiently in MATLAB whereas they are easy to do in C, whereas others are very difficult to do in C.

For example, MATLAB programmers often resort to creating quadratically-sized matrices in order to exploit vector computation, using inordinate amounts of memory, whereas in C a nested for loop with small memory usage would work. Additionally, combining MATLAB and C code, although possible, is still not yet a wide practice.

On the other hand, standalone C can be quite difficult. Using linear algebra in C usually requires writing your own methods, or going through the effort of integrating a large third-party library. Utilities for reading files have to be hand-rolled, and a lot of file readers have various quirks and expectations. With a core library, these processes can be just one line of code.

A common library can provide the basic necessities for machine learning, while still allowing familiar, efficient programming constructs. FASTlib uses the same linear algebra library, LAPACK/BLAS/ATLAS, that is used by MATLAB, and provides a very easy-to-use wrappers for these (finding singular values in LAPACK requires two function calls of 14 parameters each, whereas our single wrapper needs only two parameters).

1.2.1 Development Model

A core tenet of FASTlib is that it is to be open and extensible not just by a few developers, but by a community. Typically, a centralized code base implies a plethora of politics: Who owns the code? Whose style should be used? Will contributions be rejected on pure stylistic grounds? Will the code have to be forked if different requirements are necessary?

To avoid a lot of these plaguing issues, a staging model is used. Contributed code follows a simple migration path. We'll describe starting from the beginning of the migration path to the end. We want to support a vast array of algorithms. These are part of FASTlib's extended API and must meet a certain standard of quality, but not nearly as high as we would want the core of FASTlib to be, in order to facilitate contribution.

1.2.2 Core API

The core API is extremely well organized and thoroughly reviewed, with a consistent programming style, in order to be easy to use, flexible, and very fast. Every person has their own view of how a problem could be solved, and our API should be flexible so that new ideas can be easily built upon.

The core API has many common data structures, useful frameworks and libraries such as parallelization support, and fundamental numerical and data analysis techniques. The core API is in the form of a linkable library, and not executable files.

1.3 Language and Style

We chose to use a combination of C and C++ that is both fast and flexible. Although C++ can lead to difficult code when used improperly, our code design sticks to simpler features of C++ avoid other pitfalls.

For examples, in unmanaged languages like C or C++, especially in machine learning, it is entirely possible to write an algorithm that appears to work but is actually incorrect because of no run-time checking.

For example, a neural network that computes weight updates for each node (suppose we number this node N), but applies the weight to node N+1, might actually appear to work. FASTlib has a very fast debug mode that will check accesses to data structures, and would immediately catch the neural network error as an out-of-bounds write to an array, saving the developer hours of manually inserting print statements or reading over the code multiple times.

To be fast, we use C++ templates in inner-loop type computations, but employ simple inheritance in other situations. The inheritance tree shouldnt too be too tall to navigate, to avoid distributing logic over many layers of hierarchy.

Finally, C++ is an excellent language for developing parallel code. All the machine's features are there, but C++ templates provide exciting possibilities for simplification: for instance, we use templates to allow painless serialization of data structures.

1.4 Why do I want to use FASTlib?

Maybe there is some machine learning algorithm you'd like to implement, but you want to do this in a fast programming language but have at your fingertips many common tools. With FASTlib, you will alternately be able to apply many machine learning algorithms all from the same package, and integrate them together.

1.5 THOR Component

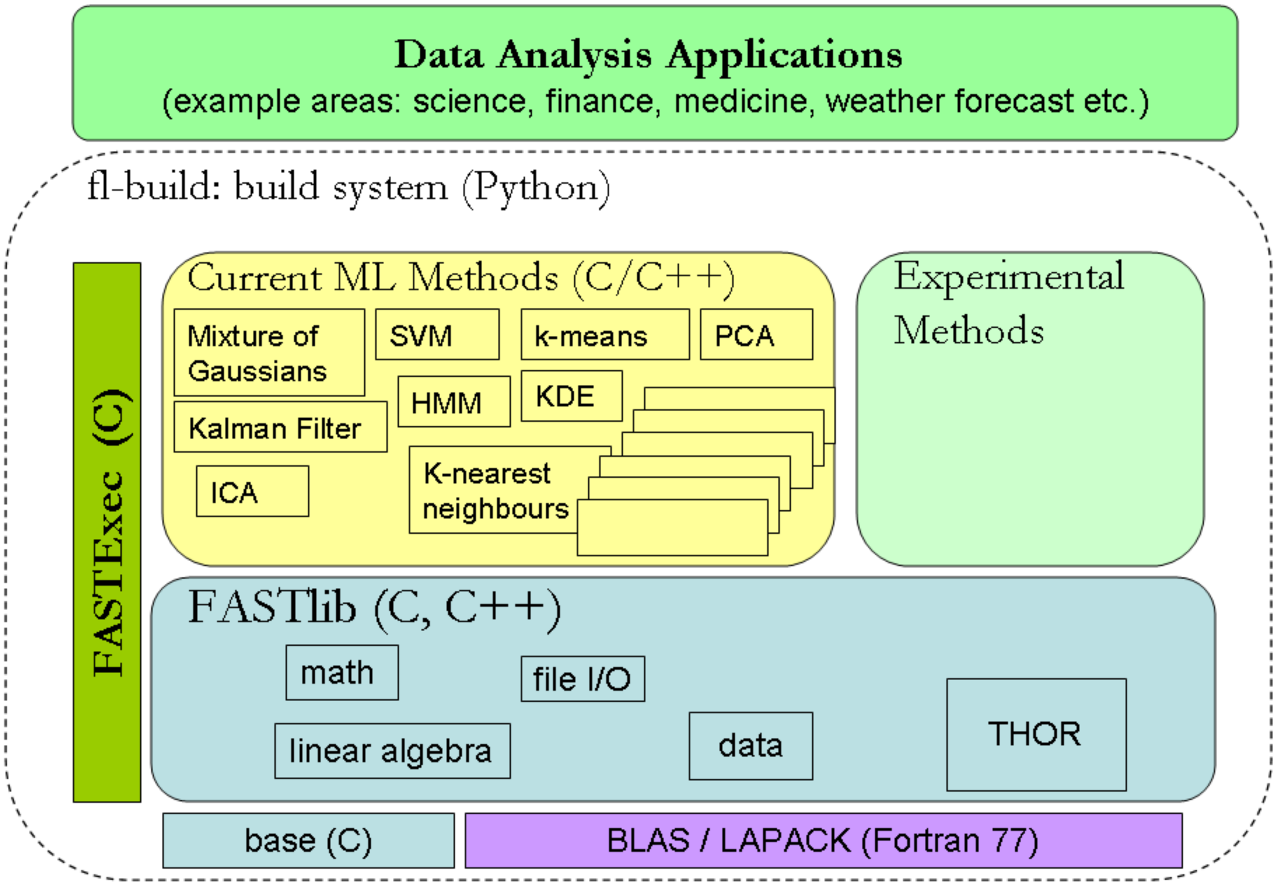
Data Analysis Applications
(example areas: science, finance, medicine, weather forecast etc.)
fl-build: build system (Python)
FASTExec (C)
Current ML Methods (C/C++)
Mixture of Gaussians
SVM
k-means
PCA
HMM
KDE
Kalman Filter
ICA
K-nearest neighbours
Experimental Methods
FASTlib (C, C++)
math
file I/O
linear algebra
data
THOR
base (C)
BLAS / LAPACK (Fortran 77)

Figure 1.1: Overall Organization of the different components of FASTlib

Chapter 2

Using FASTlib and dveloping code using FASTlib

2.1 Obtaining and Building FASTlib

2.1.1 Before Installation

Systems that we have tested:

- Linux: 32-bit x86, 64-bit x86, and Itanium
- Cygwin (x86 Windows)
- MacOSX (you will need g77, see http://hpc.sourceforge.net/ and search for g77)

Tools you'll need:

- required: gcc C compiler, g++ C++ compiler, and g77 FORTRAN compiler
- required: python 2.2 or later, check by typing python -V, earlier versions will NOT work
- optional: doxygen (to generate local copies of documentation)

2.1.2 Downloading and building FASTlib

Download the FASTlib library from blah ...We are assuming that the user has obtained the library tar ball from some place. Let's now assume that FASTlib has been untarred to `$FASTLIBPATH` . Make sure this is an absolute path (it will start with a "/" at the beginning).

After this, you will need to make sure your environment variables are set up properly. Simply add `$FASTLIBPATH/script` into your `$PATH` environment variable. First find out what shell you are using by typing

```
echo $0
```

and it will be bash, ksh, csh, or tcsh. If you are using the bash or ksh shell, you should add this to your /.bashrc (for bash) or /.kshrc (for ksh), or /.profile if the other doesn't exist, substituting `$FASTLIBPATH` accordingly:

```
export PATH="$FASTLIBPATH/script:$PATH"
```

If you are using csh or tcsh, put the following in your `~/.cshrc` file:

```
setenv PATH "$FASTLIBPATH/script:""$PATH"
```

Close your terminal and re-open it. Check to see if FASTlib is working by typing:

```
fl-build
```

It should give you the help message for FASTlib's build system. If it doesn't, type:

```
echo $PATH
```

and make sure the `$FASTLIBPATH/script` is there and is spelled exactly correctly. Try typing the export or setenv command by hand and see if it starts working afterwards. Now, change into your `$FASTLIBPATH` , and change into `example` . This is where the example code is, and you can try building it by:

```
fl-build main
```

Now run 10-fold cross-validation with K-nearest-neighbors:

```
./main --data=fake.arff
```

You should see p_correct is 1.0 for fake.arff, which was fabricated specifically to do well with nearest-neighbor classification.

2.2 Code Organization

NOTE: This section will have to be reorganized eventually.

FASTlib's core has a major C++ part, but smaller python parts with a few shell scripts. These are laid out in the following way:

- Core C++
  - base/ - basic compiler abstractions and debugging tools
  - fx/ - FASTexec client library, for managing parameters, results, and timers, via the data store (actually, this is C so that you can use it in non-FASTlib code)
  - data/ - dataset utilities and cross-validation
  - math/ - some basic mathematical utilities (we envision growing this)
  - la/ - linear algebra routines (thanks to BLAS and LAPACK)
  - file/ - convenient stateful file readers
  - col/ - collections (dynamic array, heap)
  - fastlib/ - wraps all the packages neatly into one package and header

  - thor/ - template-based dual-tree parallelization
- Community-built C++
  - u/ (user directory)
- Other
  - script/ has scripts that will be in your `$PATH` variable, and a python mini-library used by the scripts
  - bin/ contains files generated by the build system - deleting this is equivalent to make clean
  - bin_keep/ contains other files generated by the build system but aren't meant to be cleaned, such as the linear algebra package

2.3 Developing your own code using FASTlib

2.3.1 Using the build tool

As stated earlier, building is done via the fl-build tool, which stands for FASTlib build. This tool reads through very short files which just have a list of sources (.cc files), headers (.h files), and other sub-packages it depends on, and produces a Makefile, which it runs automatically.

There are a handful of compilation modes supported by fl-build, specified by the =–mode=mode = parameter. Each mode has a use, and enables certain flags:

- verbose: tracking the execution of a program when it's infeasible to step through manually. disables optimizations, enables printing of verbosity messages
- debug: allow best use with gdb for tracking bugs. disables optimization, enables all debug checks, enables highest level of gdb.
- check (default mode): regular development. enables code optimizations and leaves in debug checks, at a 25
- fast: timing runs, for the fairest timing comparisons. debug symbols still enabled but may be inaccurate.
- unsafe: optimizations that might alter correctness, and often will slow the program down.
- profile: speed profiling. compile with –mode=profile, run your program, and run gprof ./binaryfile — less to see what the bottleneck is.

The command line flags are listed in script/buildsys.py.

Thus, we could re-build the example, but turn off all debug checks:

```
fl-build main --mode=debug
```

To add custom flags, add `--cflags`. To get increased performance on Pentium 4 and define a macro called `COAGULATE`, you would use:

```
fl-build mybinary --mode=fast --cflags="-march=pentium4 -DCOAGULATE"
```

Writing build files

The fl-build tool looks in the current directory for a file called build.py. The build file is executed as straight Python code, with access to a few specific functions defined by the build system, that correspond to "meta build rules". In processing these, the build system recursively pulls build files from other directories and resolves the dependencies. A Makefile is then created in your current directory, which fl-build automatically runs for you.

Here is a sample build rule that will build an executable test compiled from test.cc and test.h, and link it against the fastlib front-end library:

```
binrule(
   name = "test",
   sources = ["test.cc"],
   headers = ["test.h"],
   linkables = ["fastlib:fastlib"]
   )
```

Each string you see is actually processed by the build system, which figures out what you are referring to. If there is no colon ":" character, it looks for a file in the same directory as the build file. If it finds a ":", such as "fastlib:fastlib", it looks in the directory "fastlib" for a rule named "fastlib" – the left part of the colon is the directory, the right part is the rule name. For example, the full path to the rule for the main executable in `example` would be `‘‘example:main"` .

Next, suppose you want to create a small reusable library. Taking a look at the build file `u/example/build.py` ,

```
librule(
    name = "example",              # this line can be safely omitted
          # (since this is u/example, the build system
          # will automatically name it "example" if you omit it -- most
          # other sub-packages will omit the name)
    sources = ["helper.cc"],       # files that must be compiled
    headers = ["helper.h"],        # include files part of the ’lib’
    deplibs = ["fastlib:fastlib"]  # depends on fastlib core
    )

binrule(
    name = "main",                 # the executable name
    sources = ["main.cc"],         # compile main.cc
    headers = [],                  # no extra headers
    linkables = [":example"]       # depends on example in this folder
    )
```

2.3.2 Use of C++

FASTlib is C++, but only to an extent. If you are familiar with C, you will have no problem. The things we use from C++ is:

- Templates, for pluggability (used very extensively in THOR)

- Classes, for coupling data and operations over the data
- Destructors, to avoid unnecessary clean-up code, and to better support templates

FASTlib mostly avoids inheritance, virtual functions, and operator overloading. It also notably avoids most constructors due to their pickiness about when they are called. Here's an example of what this looks like. In particular, we'll create a multiplication table:

```
Matrix mult_table;

mult_table.Init(10, 10); // make it 10 rows by 10 columns

for (index_t i = 0; i < 10; i++) {
  for (index_t j = 0; j < 10; j++) {
    mult_table.set(i, j, i * j);
  }
}
// the matrix does not have to be freed
```

Some quick notes before we move on. The matrix must be initialized before it is usable, but keep in mind everything is freed by default. Caution – if you declare a matrix and never initialize it, the program will crash at the end of the function. In debug mode, many FASTlib classes will let you know that this is happening. Next, the `index_t` type is usually an regular int, but is wired through the system to become 64-bit if you tell it to.

2.3.3 Copying and Aliasing

C++ is infamous for its desire to make copies of everything. If you forget to pass a parameter as a constant reference (const Classname&), everything will be copied, but sometimes, there is just no way to get around of it. By avoiding constructors, we find copying is usually not needed. To avoid accidental copies of your class, put `FORBID_COPY` at the beginning like this:

```
class X {
  FORBID_COPY(X);
  ...
}
```

Sometimes, though, you really need to make copies, or at least things like copies. Many classes support a Copy method to explicitly do so, through the Matrix and Vector.

The Vector and Matrix classes support a concept of aliasing. A vector can be a vector on its own, or it could also point to some column within a matrix; a matrix can also refer to a subset of another matrix's columns. The concept is simple, but there is one caveat: without garbage collection, somebody eventually has to free the memory. Each Vector and Matrix then knows whether it is the owner of the memory it points to, and if it is, it will free the data automatically; otherwise, it is only an alias.

An example is shown here:

```
   Matrix original;
   original.Init(5, 5);
```

```
...
for (int j = 0; j < 3; ++) {
  Matrix weak_alias;
  weak_alias.Alias(original); // this matrix now aliases the original
  weak_alias.set(j, j, 999.0); // this modifies the original matrix
  // the destructor of weak_alias is called, but the memory is not freed
}
Matrix newowner;
newowner.Own(&original); // newowner
Matrix copy;
copy.Copy(original); // a completely new copy
original.Destruct(); // newowner is still valid
original.Init(99, 99);
```

2.3.4 Debugging

C++, and equally C, can sometimes make it easy to shoot yourself in the foot. To help you avoid this, we made debugging an important part of FASTlib.

The build system by default will enable all checks in the form of `#ifdef DEBUG` (we'll get back to this later). These checks and safeguards, such as bounds checking and memory poisoning, are present in nearly all core FASTlib classes. If you are debugging and find `0xDEADBEEF, 2146666666, or 0x7FF388AA, or NaN` , it probably means you forgot to initialize something; if you go beyond the end of a Vector, it will let you know. In practice, these have a 20

To use debugging yourself, plaster your code with the following:

- `DEBUG_ASSERT_MSG(condition, format, ...)` - If the condition is false, your program will print the formatted message to stderr and die.
- `DEBUG_GOT_HERE(3)` - If verbosity is ¿ 3 and verbose-mode is compiled on, it will print the function and line number.

You can safely leave these in at no cost in non debug mode. To set verbosity level to 3.0, you would specify the command line argument: `--debug/verbosity_level`3.0 =.See the Doxygen for base/debug.h for more information.

2.3.5 FASTexec - Command-line parameters and experimentation

We felt it is important that machine learning researchers can run a lot of experiments without too much trouble. Parameter passing is integral to the experimentation process, so we unified these.

The core idea behind the system is that within one run of your program, a hierarchial data store is created. In some ways it is similar to a "Windows registry" except it only has a very brief existence. In fact, it more like an XML document tree, except without attributes, and has a "stateless" output format.

Basic command-line parameters

Let's look at a non-trivial example. We're measuring how different strided memory access patterns affect cache performance (something you probably won't care about, but is trivial to code):

```
#include "fastlib/fastlib.h"
int main(int argc, char *argv[]) {
  fx_init(argc, argv); // initialize command-line parameters and the data store
  int count = fx_param_int_req( // get a REQUIRED integer parameter
      NULL, // directly from command line (not from a sub module)
      "count"); // called "count", as in --count=num
  int stride = fx_param_int(NULL, "stride", 1); // defaults to stride 1
  double factor = fx_param_double(NULL, "factor", 1.0); // default value 1.0

  ArrayList<double> values;
  values.Init(count);

  fx_timer_start(NULL, "strides"); // timers have names
  // Let's see how stride affects cache performance
  for (int s = 0; s < stride; s++) {
    for (int j = s; j < count; j += stride) {
      values[j] = factor * j;
    }
  }
  fx_timer_stop(NULL, "strides");
  fx_format_result(NULL, "success", "%d", 1); // store some result
  fx_done();
}
```

The useful functions we used were `fx_param_int` (to get an integer), `fx_param_double` (to get a double-precision value), `fx_timer_{start|stop}` to have timers, and `fx_done()` to output the results. I ran this example with the parameters:

```
./test --count=10000 --factor=2.0
```

and the following text resulted:

```
<code>
/timers/strides/children/sys 0.000000
/timers/strides/children/user 0.000000
/timers/strides/self/sys 0.000000
/timers/strides/self/user 0.000000
/timers/strides/wall/sec 0.000061
/timers/default/children/sys 0.000000
/timers/default/children/user 0.000000
/timers/default/self/sys 0.000000
/timers/default/self/user 0.000000
/timers/default/wall/sec 0.000354
/params/stride 1
/params/fx/timing 0
/params/debug/print_notify_headers 1
/params/debug/pause_on_nonfatal 0
```

```
/params/debug/abort_on_nonfatal 0
/params/debug/print_warnings 1
/params/debug/print_got_heres 1
/params/debug/verbosity_level 1
/params/factor 2.0
/params/count 10000
/results/success 1
/info/rusage/children/nivcsw 0
/info/rusage/children/nvcsw 0
/info/rusage/children/nsignals 0
/info/rusage/children/msgrcv 0
/info/rusage/children/msgsnd 0
/info/rusage/children/oublock 0
/info/rusage/children/inblock 0
/info/rusage/children/nswap 0
/info/rusage/children/isrss 0
/info/rusage/children/idrss 0
/info/rusage/children/ixrss 0
/info/rusage/children/maxrss 0
/info/rusage/children/majflt 0
/info/rusage/children/minflt 0
/info/rusage/children/stime 0.000000
/info/rusage/children/utime 0.000000
/info/rusage/children/WARNING your_OS_might_not_support_all_of_these
/info/rusage/self/nivcsw 1
/info/rusage/self/nvcsw 3
/info/rusage/self/nsignals 0
/info/rusage/self/msgrcv 0
/info/rusage/self/msgsnd 0
/info/rusage/self/oublock 0
/info/rusage/self/inblock 0
/info/rusage/self/nswap 0
/info/rusage/self/isrss 0
/info/rusage/self/idrss 0
/info/rusage/self/ixrss 0
/info/rusage/self/maxrss 0
/info/rusage/self/majflt 0
/info/rusage/self/minflt 393
/info/rusage/self/stime 0.002999
/info/rusage/self/utime 0.002999
/info/rusage/self/WARNING your_OS_might_not_support_all_of_these
/info/system/kernel/build %231%20SMP%20Tue%20Jan%2023%2012%3A49%3A51%20EST%202007
/info/system/kernel/release 2.6.9-42.0.8.ELsmp
/info/system/kernel/name Linux
/info/system/arch/name x86_64
/info/system/node/name shannon.cc.gatech.edu
```

We admit this is a lot of text, but when you later comb through the results, you can select only the portions you care about. Briefly, each portion of the tree:

- params/ - Parameters you were called with. Default parameters are explicitly stored in case you later change what the default values are.
- timers/ - Each timer. There will always be a default timer, which runs between `fx_init` and `fx_done`.
- results/ - Results that you decided to emit.
- info/rusage/ - System information related to rusage. See the manpage for getrusage(2).
- info/system/ - Information on the system run on. See manpage for uname(2).

This output could indeed just be an s-expression, or it could be an attribute-less XML file. We chose this path-value dump format because it is stateless – i.e. anyone can process it with a little bit of grep. Corruption in the file will only affect it until the next newline.

Running automated experiments and collecting results

Running experiments and collecting results is made simpler using FASTexec. After using the FASTexec code to store output variables in the datastore, you can use FASTexec to run multiple experiments. If you type the command:

```
fx-run | less
```

you will see the help screen for fx-run. This program will run your executable with all combinations of the parameters you give it, and save results in a directory called fx that is created in the same directory you run the tests. After that, gather results using fx-csv to make an Excel-compatible comma-separated-values file, or fx-latex to create a properly escaped LaTeX table.

You can try an example with the K-nearest-neighbors classifier example in u/example:

```
  cd $FASTLIBPATH/u/example && fl-build main
  fx-run knn_k ./main --knn/k=1,2,3,4,5 --data=../../../../fake.arff
  fx-csv knn_k ./main /params/knn/k /kfold/results/p_correct
  fx-latex knn_k ./main /params/knn/k /kfold/results/p_correct --preview
```

2.3.6 Little gotchas

Success and failure

The `success_t type` in `base/common.h` defines our standard for indicating success or failure. Rather than assuming 1 or 0 or -1 indicates something or other, we explicitly return `SUCCESS_PASS` (succeeded), `SUCCESS_FAIL` (failed), or `SUCCESS_WARN` (something was suboptimal).

Basic types

You will notice heavy use of `index_t` (in `base/scale.h`) rather than integers. For practical purposes, `index_t` is a signed integer – signed so you can loop over ¿= 0. On 32-bit and 64-bit Intel/AMD machines, this will be 32-bits, which is the fastest int for both systems and is relatively compact. But if you want to operate on datasets larger than a few gigabytes, you can define the `SCALE_LARGE` macro, and these will instantly switch to the largest size your computer can address.

Also, the `base/basic_types.h` file (it will be in bin/ since it is auto-generated) defines standard int16, int32, int64, uint16, uint32, uint64 types.

Printf versus Streams

We operate under the assumption that more of our audience is familiar with C I/O than with C++ I/O, especially when it comes to fancy floating-point formatting. The caveat is that printf only works with native types like int, short, long. To print an `index_t`, you use:

```
printf("Iteration %"LI"d complete.", some_index_var);   // no special formatting
printf("Iteration %04"LI"d complete.", some_index_var); // with formatting
```

The LI macro is a string that will have the suitable ”l” modifier for `index_t` (see man sprintf for more details). For other types:

- int16 and uint16: L16
- int32 and uint32: L32
- int64 and uint64: L64

Alternately, you can just cast the variable to a native type like (int) (short) or (long), if you want to be lazy.

Chapter 3

Complete code development walkthrough - Dual-Tree k-NN

This chapter is intended for people who have read the FASTlib Tutorial and successfully compiled and run the example code. You should also understand how a dual-tree all nearest neighbors algorithm works, since this is assumed.

This should help you get started on understanding the basic features of the library. For complete documentation, please see the Doxygen file.

3.1 Walkthrough

3.1.1 build.py

The build.py file is a necessary part of any FASTlib directory. It tells the fl-build script how to compile your code and link to the rest of the library. Essentially, the build.py functions like a Makefile, but is much easier to understand and write.

There are two important kinds of entry in build.py files: binrules and librules. The distinction between these is that binrules create stand-alone executables (somewhere in their code is the main function) while librules are for code used in linking (no main). It is usually a good idea for the bulk of your project to be compiled with a librule, linked to by a simiple binrule in the same build.py:

3.1.2 Librule

```
librule(
  name = "allnn",
  sources = ["allnn.cc"],
  headers = ["allnn.h"],
  deplibs = ["fastlib:fastlib"],
  tests = ["allnn_test.cc"]
)
```

name - the name of the library being created. If this line is omitted, fl-build will use the name of the directory as a default.
sources - The .c or .cc files necessary. This line can be omitted if there are none.
headers - The .h files necessary. This line can also be omitted if there are none.
deplibs - Other libraries linked to by the current one. The name before the colon is the directory within fastlib that contains the library, the name after the colon is the name of the library itself. :$LIBNAME indicates that the library is in the same directory as the build.py file. tests - A file containing unit tests. If this line is in the librule, you can compile the unit tests with fl-build allnn_test.

3.1.3 Binrule

```
binrule(
  name = "allnn_main",
  sources = "allnn_main.cc",
  headers = "allnn_main.h",
  deplibs = [":allnn"]
)
```

The tags are similar as above. Running "fl-build allnn_main" will create an executable called "allnn_main".

3.1.4 allnn_main.cc

Our main file contains only the main function. In general, FASTlib code should have very simple main functions. Our main function reads in the command line arguments, loads the data, and saves the results. All of the computation is done by an AllNN object, defined in allnn.h.

3.2 FASTexec

See FASTexec Tutorial for a more detailed discussion of FASTexec. FASTlib code starts by calling:

```
fx_init(argc, argv);
```

which initializes the FASTexec system. It must be accompanied by:

```
fx_done(argc, argv);
```

at the end of the main function.

3.3 Reading and writing data

Our function determines the files containing the query and reference data. We accomplish this with:

```
const char* queries_file_name = fx_param_str_req(NULL, "q");
Matrix queries;
data::Load(queries_file_name, &queries);
end{verbatim}
```

```
The only other parameters our program reads are a boolean:
\begin{verbatim}
int do_naive = fx_param_bool(NULL, "do_naive", 0);
```

and the output filename:

```
const char* output_filename = fx_param_str(NULL, "output_filename", "output.csv");
```

In general, data can be read in from the command line using

```
fx_param_$TYPE_req(module, "name");
```

for required parameters (the program will terminate with an error if it is not specified). Alternatively, one can specify a default value with:

```
fx_param_$TYPE(module, "name", $DEFAULT_VALUE);
```

In both cases, the parameter is given on the command line as

```
--name=$VALUE"
```

3.4 Modules

The first argument to each of these functions indicates a module. Modules contain parameters, results, and timing information for various portions of the program. For a detailed explanation of modules, consult the FASTexec Tutorial.

For this code, we only utilize a few modules. When NULL is passed as a module, it indicates the root module of FASTexec. So, the line:

```
const char* output_filename = fx_param_str(NULL, "output_filename", "output.csv");
```

indicates that

```
--output_filename="some_file.csv"
```

will appear on the command line (i.e. specified in the root module's /params folder), or else "output.csv" will be used as the default.

We also create a module for our AllNN object. The line:

```
fx_submodule(NULL, "allnn", "allnn_module");
```

creates a new module called "allnn_module", copies all parameters stored under allnn to this module, and places the entire thing under the root directory (NULL).

3.5 Timers

Timers are also handled by FASTexec. They are stored in the /timers section of a module. We can start a timer with:

```
fx_timer_start(allnn_module, "dual_tree_computation");
```

and stop it with

```
fx_timer_stop(allnn_module, "dual_tree_computation");
```

These timers will print as a part of the final output of the program, and are suitable for parsing with the fx-run commands. Timers, modules, and command line parameters can be accessed from any part of the program, not just main.

3.6 Development Walkthrough

You are developing code. What to do, step by step:

Step 0: You should have read the FASTlib tutorial and successfully compiled and run the example.

Step 1: Your code needs a home. You will work on it in your user directory, i.e.: u/plato/allnn

Step 2: You need a build.py file, which tells fl-build what to do to compile your work. It is the equivalent of a Makefile, but a bit easier to understand.

There are two important kinds of entry in build.py files: binrules and librules. The distinction between these is that binrules create stand-alone executables (somewhere in their code is the main function) while librules are for code used in linking (no main). It is usually a good idea for the bulk of your project to be compiled with a librule, linked to by a simiple binrule in the same build.py:

```
librule(
  name = "allnn",
  sources = ["allnn.cc"],
  headers = ["allnn.h"],
  deplibs = ["fastlib:fastlib"],
  tests = ["allnn_test.cc"]
)

binrule(
  name = "allnn_main",
  sources = "allnn_main.cc",
  headers = "allnn_main.h",
  deplibs = [":allnn"]
)
```

Note that ”tests” in the librule allows you to compile your unit tests with ”fl-build allnn_test”.

Step 3: Now we need to start writing the code. We will start with allnn_main.cc by incluidng allnn.h at the top and writing a main function:

```
#include "allnn.h"
int main(int argc, char *argv[]) {
  fx_init(argc, argv);
  ...
  fx_done();
  return 0;
}
```

FASTlib main functions should always begin and end by initializing and finalizing fx, or FASTexec, which manages command line input among other things.

In our particular project, the first logical thing to do is to load the data. We need to get the input file names out of the command line arguments and then to use a library function that reads matrices.

```
const char *q_filename = fx_param_str_req(NULL, "queries");
const char *r_filename = fx_param_str_req(NULL, "references");
Matrix q;
Matrix r;
data::Load(q_filename, &q);
data::Load(r_filename, &r);
```

We organize all of our project's tasks into a class called AllNN. After declaring an object of this class, we initialize it with the two data sets and a submodule, which serves to pass it its own parameters from the command line.

```
struct datanode *allnn_mod = fx_submodule(NULL, "allnn", "allnn_mod");
AllNN allnn;
allnn.Init(q, r, allnn_mod);
```

We must declare a local variable to receive the results of computation.

```
ArrayList<index_t> results;
allnn.ComputeNeighbors(&results);
```

We emit result by printing to a file.

```
const char *o_filename = fx_param_str(NULL, "out", "out.csv");
FILE* o_file = fopen(o_filename, "w");
ot::Print(results, o_file);
```

Step 4: Moving to allnn.h, the first thing to do include the rest of FASTlib within appropriate inclusion guards:

```
#ifndef ALLNN_H
#define ALLNN_H
#include "fastlib/fastlib.h"
...
#endif
```

Make sure you include ”fastlib/fastlib.h” as opposed to ”fastlib.h” so the complier can properly find the file.

Step n: Write your unit tests.

NOTE: This chapter still need more work.

Chapter 4

Examples of some common tasks - FASTlib Cookbook

Chapter 5

Code Style Suggestion for developers

The coding style suggestions in this chapter stem from our own development expereince and the style used to develop FASTlib. It is recommended that you use the suggestions herein for your code development if you would like to contribute your code to be part of the standard FASTlib distribution. You could also choose to skip this chapter.

Chapter 6

Beyond this Document