Create your own optimizer in Unfit

By: Dr Martin Buist
Initial version: August 2014
Unfit version: 3.0 or later

This guide was written post Summer of Code 2014 when I was adding the ParticleSwarm optimizer to Unfit in preparation for the Unfit 3.0 release. This is the process that I followed, and the thought I had along the way.

Getting Started:

So you have discovered a flash new optimization method that you would like to try within the Unfit framework? This is the document you need. The first thing to do is to start CodeBlocks on your computer and load the Unfit-devel.cbp project file. If you have not used Unfit before, please see the introductory documentation to see how to set up UnitTest++ etc. Once you can compile and run the existing Unfit test suite you are ready to get started.

Each new optimization method is encapsulated in two files:

> <Unfit home>/include/MyNewMethod.hpp
> <Unfit home>/src/MyNewMethod.cpp

Obviously, you should put your definitions in the header (hpp) file and the implementations in the source (cpp) file. We **STRONGLY** recommend adding an additional file:

> <Unfit home>/unittests/TestMyNewMethod.cpp

This file contains the unit tests you need to test that your new method is working properly. If you would like us to consider adding your method to the main version, this is mandatory. Finally, we would also recommend adding a fourth file:

> <Unfit home>/examples/TestExamplesMyNewMethod.cpp

Here you should adapt the Unfit example suite for use with your new method. How does it perform? Is it faster than Levenberg-Marquardt? More reliable than Differential Evolution?

Step 1: Creating the Files

Here we assume that you are using Codeblocks, as that is what we use. The process for other environments is probably similar. First, start Codeblocks and load the Unfit-devel project (Unfit-devel.cbp). Don't worry about Unfit.cbp, we will update that later once we are sure everything is working.

Assuming that you want to produce a brilliant, high quality implementation, we need to add the four files listed above. If you are just interested in hacking up a prototype, just the first two will do. To create a file and include it in the project, just go

> File -> New -> Empty File

I always choose the empty file option so Codeblocks does not try to be overly helpful. You will be asked if you want to add the file to the active project and the answer is of course, yes. Please

remember to save your files in the correct directories. You will also be asked which targets the new file should belong to. Unfit has three targets, Debug, Release, and Library. The main difference is that the Library version contains no tests, so uncheck the Library button for the last two files.

MyNewMethod.hpp	Debug, Release & Library
MyNewMethod.cpp	Debug, Release & Library
TestMyNewMethod.cpp	Debug, Release
TestExamplesMyNewMethod.cpp	Debug, Release

Step 2: Class Outline

First we will put some essentials into the header file (MyNewMethod.hpp). Perhaps the easiest way to go about this is to copy-paste from an existing method, but care is needed to update everything. Around all of the code in MyNewMethod.hpp we will need a header guard to prevent multiple inclusions:

```
#ifndef UNFIT_INCLUDE_MYNEWMETHOD_HPP_
#define UNFIT_INCLUDE_MYNEWMETHOD_HPP_

<code will go here>

#endif
```

Please stick to our naming convention if you want us to consider including your method. Next, we need to include the necessary header files. The three below are the minimum set, and you can add more as required. We have the C++ headers first, followed by our headers, with each sorted alphabetically.

```
#ifndef UNFIT_INCLUDE_MYNEWMETHOD_HPP_
#define UNFIT_INCLUDE_MYNEWMETHOD_HPP_

#include <vector>
#include "GenericCostFunction.hpp"
#include "GenericOptimizer.hpp"

namespace Unfit
{

<code will go here>

}  // namespace Unfit

#endif
```

There are good reasons why you need these headers. Your optimizer must derive from our GenericOptimizer class to ensure compatibility, and therefore must take in a GenericCostFunction which returns a std::vector. You will notice that something else was added along with the header files and that is the Unfit namespace. By putting all of our code into the Unfit namespace we will hopefully avoid naming conflicts between the internals of Unfit and external code.

Now we are ready to define our class. As mentioned above, you must derive your optimizer class from GenericOptimizer. If you have a look at the GenericOptimizer code you will notice a few things. First, there is a lot of functionality there that will make your life easier (e.g. bounds, printing, accessing solutions), so please use what is there, don't waste your time duplicating what is there. Next, you will see that the helpful methods provided in GenericOptimizer are almost all virtual, meaning you can override these with your own methods if it is really necessary. Finally, you will notice that two methods, FindMin and Reset, are pure virtual methods, which means you must override them in your optimizer class. The minimum class definition therefore looks like this:

```
namespace Unfit
{

class MyNewMethod : public GenericOptimizer
{
  friend class TestMyNewMethod;
 public:
  MyNewMethod();
  virtual ~MyNewMethod() = default;
  int FindMin(GenericCostFunction &CostFunction,
      std::vector<double> &coordinates);
  void Reset();
};

}  // namespace Unfit
```

Here the header guards and include files have been omitted for brevity. There are some style points worth noting here.

- The class definition is not indented from the namespace
- The public, protected & private keywords are indented one space
- A regular indent is two spaces
- The maximum allowable line length is 80 characters
- Wrapped lines are indented by four spaces
- Use the std:: prefix rather than “using namespace std”
- No raw pointers
- C++11/14 rocks. Use it rather than legacy C++98 or worse

The definition includes the default constructor (which you will probably want to implement), the default destructor (which you probably won't have to implement), and the definitions of the two pure virtual methods from GenericOptimizer that we are obliged to implement. There is another line too, that defines a friend class for the new method. In general we think that friend classes are evil and should be strongly discouraged, but this is one case when they very useful. If you look at the name of the proposed friend class, you will see that it has the same name as the file we made to contain our unit tests (you did intend to unit test your code – right?). So, this is purely for unit testing purposes – more later. Even if you are not creating unit tests it does no harm to include this line, so you might as well include it.

Step 3: Class Implementation

Now we have the bones of our class declaration, we can have a look at the bones of an implementation. There are only three methods here, because we used the C++11 keyword “default” on the destructor in our header file. Have a look at the code below.

```
#include <vector>
#include "Bounds.hpp"
#include "MyNewMethod.hpp"
#include "Options.hpp"

namespace Unfit
{

MyNewMethod::MyNewMethod()
{}

void MyNewMethod::Reset()
{
  ResetGenericOptimizer();
}

int MyNewMethod::FindMin(GenericCostFunction &CostFunction,
    std::vector<double> &coordinates)
{
  auto rc = 0;
  return rc;
}
}  // namespace Unfit
```

At the top are the four headers (minimum) we need to include to access the inbuild options and bounds functionality. You will notice that all of the code is again wrapped by the Unfit namespace. The constructor is currently blank, and the Reset method contains a call to reset the GenericOptimizer. FindMin is the method that performs the optimization, and this is required to return an integer return code.

If you are wondering if you can add private member variables and methods to your class, the answer is a definite yes. However, you should initialize all of your member variables in the constructor via an initializer list (and reset them in the Reset method) as shown below. Say I want to add a private integer integer and a private boolean to my class (hpp file):

```
class MyNewMethod : public GenericOptimizer
{
  friend class TestMyNewMethod;
 public:
  ....
 private:
  int my_private_member_1_;
  bool my_private_member_2_;
};
```

The implementation should then look like this (cpp file):

```
MyNewMethod::MyNewMethod()
  : my_private_member_1_ {0}
  , my_private_member_2_ {false}
{}
```

and don't forget to keep the Reset method in sync:

```
void MyNewMethod::Reset()
{
  ResetGenericOptimizer();
  my_private_member_1_ = 0;
  my_private_member_2_ = false;
}
```

In the body of the constructor you can override the default values for options from the GenericOptimizer, such as the maximum number of iterations, but try not to, and if you do, these should be ***clearly documented***.

You may have noticed these other style conventions along the way:

- Use CaMel case for class and method namesake
- Use all lower case for variable names, with an underscore separator
- Add a single trailing underscore to member variables
- Use all upper case for preprocessor directives (#define, #ifdef etc)
- Avoid preprocessor directives apart from header guards
- Use auto
- Prefer the uniform initialization syntax {}

Step 4: FindMin

The FindMin is where you put the brains of your optimizer. You are strongly encouraged to have a look at the implementations of existing methods to get ideas as to how to implement yours. Again, utilise the functionality in GenericOptimizer as much as you can. So what does FindMin do, and what does it expect? The calling code will probably look something like this:

```
MyCostFunction cost_func;
MyNewMethod optimizer;
std::vector<double> initial_guess {1, 2, 3, 4};
auto rc = optimizer.FindMin(cost_func, initial_guess);
```

If you want to change any of the options, bounds, etc, you can add, for example:

```
optimizer.options.SetMaxIterations(10);
```

between where you create the optimizer and the call to FindMin.

Findmin expects a cost function and a vector containing an initial guess to be passed in. You should check the size of the initial guess to determine how many dimensions there are in the problem. What the user expects FindMin to do is to perform an optimization of the supplied cost function and

to ***overwrite*** the initial guess with the best solution that was found. At the same time, the user expects to receive an integer return code of ***0*** = ***success***, or ***non-zero*** = ***failure***. In general we try to use negative return codes if something was wrong with the setup (the optimization didn't start), and positive return codes if the optimization started but something went wrong (e.g. hit the maximum allowable number of iterations).

Here are some key points to make sure your FindMin is written with a minimum amount of effort and is consistent with the existing Unfit code.

- Never call the cost function directly. Use the CalculateCost method from the GenericOptimizer, and override this if the provided one does not suit your needs. We use CalculateCost as an easy way to track the number of times you call the cost function.
- If you want to make a multithreaded implementation, use standard C++ futures & async.
- Use the provided methods (PrintInitialOutput, PrintIterationOutput, PrintFinalOutput) to update the user as to the progress of the optimization. The user can control the amount of output they get by the options.SetOutputLevel() command. All you need to do is to put the three print function calls in the right place.
- Use the `iterations_` variable from GenericOptimizer to track the number of iterations. The number of function evaluations will be calculated automatically because you are using CalculateCost.
- Use the IsConverged method from the GenericOptimizer to test for convergence.
- Your optimizer should respect the bounds set by the user, and if your method is capable of going outside the set bounds, check and respect the UseHardBounds option if it has been set.
- Use the existing options whenever possible. For example, use alpha = options.SetAlpha() and options.GetAlpha(), rather than defining your own alpha parameter. This allows the user to control the behaviour of the optimizer. Choose sensible defaults in case the user does not want to control things. If you run out of alpha, beta etc it is okay to add another parameter to the options class.
- A number of our existing optimizers perform the setup work in FindMin then offload the algorithm to a (private) method called ProcessFindMin. This is not mandatory, but can be a good way of keeping your algorithm and its implementation understandable.
- Try very very hard not to use anything that would require Unfit to be linked against an external library (other than the C++ library and our unit test library).

If you have a population based method (most optimizers have some form of population):

- Check whether the user has set the population (options.GetUserSetPopulation()), and if they have, respect their choice. You can reject their population (via a negative return code) if the population is no good, but don't modify the initial population.
- Check if the user wants to include the initial guess in the population where populations are randomly generated.
- If the user has not supplied a population, check if they have set the population size.
- If you require a bounded random starting population, use GeneratePopulation from the GenericOptimizer if at all possible.
- Store the population in `population_` from GenericOptimizer. This is a vector of population members. Each member is a vector, so this is just a vector of vectors. We thought about many other data structures for this, but this is both simple and fast.
- Append the cost of the each population member to that member. Some of the built in functionality assumes this is where the costs are.
- Remember, lots of functionality is already in GenericOptimizer (e.g. Sort, SetPopulation) so don't reinvent the wheel.

A pseudo code for FindMin may look something like this:

```
int MyNewMethod::FindMin(GenericCostFunction &CostFunction,
    std::vector<double> &coordinates)
{
  // Initial population
  if (options.GetUserSetPopulation()) {
    <Check the population the proceed or reject>
  }
  else {
    <Check/set the population size>
    GeneratePopulation(CostFunction, dimensions_);
  }

  // If requested add the initial coordinates to the population
  if (options.GetAddInitialToPopulation()) {
    <do the necessary checks then add or reject>
  }

  ++iterations_;  // The initial population is counted as one iteration
  auto rc = ProcessFindMin(CostFunction);
  return rc;
}
```

and a pseudo code for ProcessFindMin may look something like this:

```
int MyNewMethod::ProcessFindMin(GenericCostFunction &CostFunction)
{
  const auto max_iters = options.GetMaxIterations();
  const auto max_func_evals = options.GetMaxFunctionEvaluations();
  PrintInitialOutput(best_member_[cost_]);

  while (iterations_ < max_iters &&
      function_evaluations_ < max_func_evals) {

    <perform one iteration of your algorithm>

    ++iterations_;
    PrintIterationOutput(best_member_[cost_]);
    if (IsConverged(best_member_)) break;
  }
  PrintFinalOutput();
  if (function_evaluations_ >= max_func_evals) return 1;
  if (iterations_ >= max_iters) return 2;
  return 0;
}
```

As mentioned earlier, have a close look at our existing algorithms (e.g. ParticleSwarm) to see what a complete implementation looks like. That is just about all you need to know. At this point I will

ask you to follow the doctrine of extreme programming and remind you to write your unit tests first. Of course I know you probably will not do this, but it is good practice so give it a try.

Step 5: Unit Testing

In Unfit we use the UnitTest++ testing framework. In our hands it has been tested on both Linux (Ubuntu, our main development system), and Windows (whatever we have lying around). The information given here is not a substitute for the UnitTest++ documentation, which you should also read. Here I will not explain the different types of checks you can use, only the structure of the testing code that you will need. You should put all of your unit tests in the TestMyNewMethod.cpp file in your unittests directory. Apologies for the long code listing, but your skeleton for your unit tests should look something like what is shown below. The length is needed to show you all of the functionality you would normally need to perform your unit tests. The explanation follows below.

```
#include "MyNewMethod.hpp"
#include "TestFunctions.hpp"
#include "UnitTest++.h"

namespace Unfit
{

class TestMyNewMethod : public MyNewMethod
{
 public:
  <type> AccessPrivateMethod(<args>)
  {
    return PrivateMethod(<args>);
  }

  void SetPrivateMember(<args>)
  {
     private_member = <args>;
  }

  <type> GetPrivateMember()
  {
     return private_member;
  }
};

namespace UnitTests
{
SUITE(UnitTestMyNewMethod)
{

TEST(MyNewMethod_TestNameInCamelCase)
{
  <test that needs only public access>
}
```

```
TEST_FIXTURE(TestMyNewMethod, TestNameInCamelCase)
{
  <test that needs to access private member variables or methods>
}

} // suite UnitTestMyNewMethod
} // namespace UnitTests
} // namespace Unfit
```

Let's start at the top and work our way down. First, you will need to include (minimum) the header for your optimizer, our TestFunctions header and, of course, the UnitTest++ header. If you need a cost function as part of your unit testing, use the ones in TestFunctions.hpp. If these are not sufficient, add one to the TestFunctions.hpp file. Don't create others at random locations. Moving down, next we again place all of our code inside the Unfit namespace.

Inside the Unfit namespace you will see a class definition which has the same name as the file we are working in (TestMyNewMethod.cpp). This is derived from the MyNewMethod class which contains your optimizer. By default, you should only be able to access the public and protected members of the MyNewMethod class. However, if you recall our MyNewMethod definition, it had a line that looked like this:

```
friend class TestMyNewMethod;
```

This line allows us to write methods in our TestMyNewMethod class that can access members and methods that are private to MyNewMethod without compromising access restrictions outside our unit tests. The class definition contains an example of how to access a private method, how to set a private member variable and how to get a private member variable. Note the naming conventions; you should substitute the PrivateMethod, PrivateMember, and private_member with the names of the methods and members you are accessing.

After the end of the class definition we are ready to start our unit tests. All of the tests are placed in the UnitTests namespace (so overall in Unfit::UnitTests). All of the tests for your optimizer should be placed in a test suite whose name starts with UnitTest, e.g.

```
SUITE(UnitTestMyNewMethod)
{
....
}
```

SUITE is a macro from UnitTest++. Inside the test suite you should write many many unit tests. There are two main types of tests you will need, and the type of test depends on whether or not you need to access private member methods or variables directly, or not. If you do not need to access anything which is private to MyNewMethod, use the TEST macro. Note the naming convention for the TEST. When using a TEST you need to make yourself an object of MyNewMethod type, and then you can work with that, e.g.

```
MyNewMethod optimizer;
optimizer.DoSomething(....);
```

If you need to access something that is private to MyNewMethod then you will want to use a TEST_FIXTURE. This macro takes in two arguments, the first is the type of object the test fixture

will instantiate (in our case TestMyNewMethod), and the second is the name of the test – note the naming convention. Say, for arguments sake, you had a private member variable in MyNewMethods called dimensions_ and wanted to prescribe the value of that variable. First, we need to create an access method in the TestMyNewMethod class:

```
class TestMyNewMethod : public MyNewMethod
{
 public:
  void SetDimensions(unsigned dims)
  {
    dimensions_ = dims;
  }
};
```

Then we need to create a TEST_FIXTURE and call our access method:

```
TEST_FIXTURE(TestMyNewMethod, TestSetDimensions)
{
  SetDimensions(12);
}
```

What happens when this is executed is the TEST_FIXTURE creates an object of type TestMyNewMethod, which is just the same as MyNewMethod but it has the extra access methods (such as SetDimensions in this example). You can therefore think of the call to SetDimensions here as something like this.SetDimensions.

Write as many tests as you need to convince yourself (and others) that your implementation is correct. However, you need to be careful if your optimizer has a stochastic component (e.g. random numbers). If you change from one Linux flavour to another, or to your Cray system, or Windows, chances are you are going to get a different implementation of the random number generator and/or a different default seed. This means that your answers could be very different. You don't want to have all of your tests passing on your Ubuntu desktop, only to find that half fail on your Windows laptop (with no code changes). This will affect what you test and the types of tests that you write. Try to write unit tests that are independent of your platform. This is not always easy, but is usually possible.

Step 6: Unfit Example Suite

Another thing we encourage you do to is to test your new method on our example suite to see how yours performs compared to the existing methods. As mentioned earlier, the example suite should be placed in

<Unfit home>/examples/TestExamplesMyNewMethod.cpp

The recommended approach is to simply copy the examples from the most similar existing method. For example, if your new method is derivative based, copy the content from TestExamplesLevenbergMarquardt.cpp. If you have a good idea for a new example that can run in a reasonable period of time please let us know. Here we again utilize the UnitTest++ framework to run our examples. In terms of the included header files, the only thing you should have to change is to replace e.g. LevenbergMarquart.hpp with your MyNewMethod.hpp. The skeleton code looks something like this:

```
namespace  // file scope
{
typedef std::chrono::high_resolution_clock hrclock_t;

std::string TestTime(hrclock_t::time_point t1, hrclock_t::time_point t2)
{
  <leave unchanged>
}
}  // namespace

SUITE(ExamplesMyNewMethod)
{
TEST(MyNewMethod_CardiacAlphaN)
{
  <adapt to use MyNewMethod>
}

TEST(MyNewMethod_Exponential)
{
  <adapt to use MyNewMethod>
}
}  // suite ExamplesMyNewMethod
```

Note that here there are no TEST_FIXTUREs, only TESTs, and please also note the test naming convention. At the top there is what is known as an anonymous namespace, which is just a namespace without a name. This is now the preferred method to indicate that whatever is in that code block has file scope (cannot be seen outside the current cpp file). Here there is a helper function for timing the examples and generating nicely formatted output. This should not need to be altered. We place all of the examples into a SUITE with the name of the optimizer prefixed by 'Examples'. Outlines of the first two tests are shown. The majority of the code can probably remain untouched, but of course you will need to do some work to make sure they call your new optimizer.

Step 7: Code Quality

After spending a lot of time and effort populating the four required files for your new optimizer there are a couple of last things to do to ensure your code is correct and consistent with the Unfit source code. These represent essentially the goals we set for ourselves when implementing new methods. The methods we use here work on our Ubuntu development platform. Our goals are platform-independent, but not all of the tools we use to achieve them will be available on all platforms.

Compiler

It probably does not need mentioning, but your code should not have any compiler errors. We suggest you stick with the set of warning flags for GCC that we are currently using, and with these your code should also not generate any compiler warnings. Not even one. If you have the time, you may like to try other compilers (e.g. clang) as different compilers often pick up different problems.

Unit tests

Our primary goal is for each MyNewMethod.cpp (and MyNewMethod.hpp, if applicable), to be 100% covered by the UnitTestMyNewMethod test suite. By this I mean 100% line coverage and 100% function coverage. We use the --coverage flag on GCC and lcov to achieve this via a shell script we call CheckCoverage. In addition, you should think carefully about whether you are checking your edge cases. As an example, say you have set the maximum number of iterations to 10. You should have unit tests that check what happens when your iteration counter is equal to 9 (just inside the “edge”), 10 (on the “edge”), and 11 (just outside the “edge”) to make sure your code produces the expected behaviour.

GCC with the --coverage flag also lets you check for branch coverage. The simplest example of this is an 'if' statement. Say your code has a line that looks like this:

```
if (bob > 0) {
```

There are two possibilities here; either bob is greater than 0, or bob is not greater than 0. This is what we call a branch. Your test should cover both cases. However, it is not practical to aim for 100% branch coverage as the compiler sometimes creates branches for you when it is parsing your code. These branches you cannot see from what you have written. What we aim for then, is to cover all of our visible branches, which, like the one-liner above, can be seen from the code we wrote.

Documentation

As much as possible, use names that are self-explanatory, and aim for simple understandable code whenever possible. Document implentation notes with // comments in the cpp file. The majority of the documentation, however, should be in the hpp files. We use doxygen to parse this documentation and produce html output. When you run doxygen (our own shell script for this is called CreateDocumentation), it will produce output based on your formatted comments plus a list of documentation errors. Please go through and fix these errors; this will ensure all of your code has at least some documentation. Look at the header files for exisiting methods for ideas on what to include.

Memory management

If you have written your code in C++11, memory management is not as big an issue as it once was. You should not have used any “new” or “delete” calls. If you have, go and fix your code up now! We have a shell script called CheckMemory which essentially calls valgrind. This will pick up most memory allocation and memory access errors. The drawback is that it takes a very long time to run, so get everything ready to go and run this at the end. It is worth noting that for multi-threaded applications valgrind does not take a very long time, it takes a very, very long time. There should be zero valgrind errors from your unit test suite and your example suite.

Code formatting

The other thing we have done to check for code consistency is adapted cpplint to check Unfit. This is a python based style checker. We check the code and parse the output via a shell script we call CheckStyle. At the time of writing there are one or two things it doesn't do very well, for example it sometimes flags errors for perfectly formed lambda functions. Having said that, the majority of your errors will be genuine and should be fixed.

Step 8: Two Last Things

Congratulations, you have made it to the last step, but you haven't finished yet. There are two last

things still to do. First, there is another header file we need to update. It is called Unfit.hpp and can be found in the include directory. Those wanting to use Unfit as a library will compile libUnfit.a and include Unfit.hpp in their project. You need to add your new header file (MyNewMethod.hpp) to the Unfit.hpp file. The other thing Unfit.hpp contains is the main page for the Unfit doxygen documentation. You should also add your new method to the list of methods there.

Finally, Unfit actually has two CodeBlocks project files. The one you have been working with to date is called Unfit-devel.cbp, and contains the Unfit-devel project which includes all of the unit tests, examples etc. The second project file is called Unfit.cbp and is designed to be a simple example of how to use the optimizers in Unfit without having to worry about any testing. You will need to open Unfit.cbp and add the source and header file for your new method, e.g.

<Unfit home>/include/MyNewMethod.hpp
<Unfit home>/src/MyNewMethod.cpp

Of course these already exist; there is no need to recreate them, just add them to the Unfit project.

In parallel with these two projects, Unfit has two “main” files. In the src directory, Unfit-devel includes main-devel.cpp, which basically just runs the test suites. Also in the src directory, Unfit (non-devel) includes main.cpp instead. If you look at the contents of that file, you will see that it is a simple parabolic fitting problem and there is a block of example code for each optimizer. You should add your optimizer here. Note that this example should run with the default options if at all possible. Non-default bounds are, however, allowed, but try to use the same bounds as the other methods so the user has a fair comparison. Remember to do a File --> Save Everything in CodeBlocks so the addition of the four files to Unfit-devel and the two files to Unfit will be saved.

```
std::cout << "THE END" << std::endl;
```