The Unfit tutorial

By Dr Martin Buist

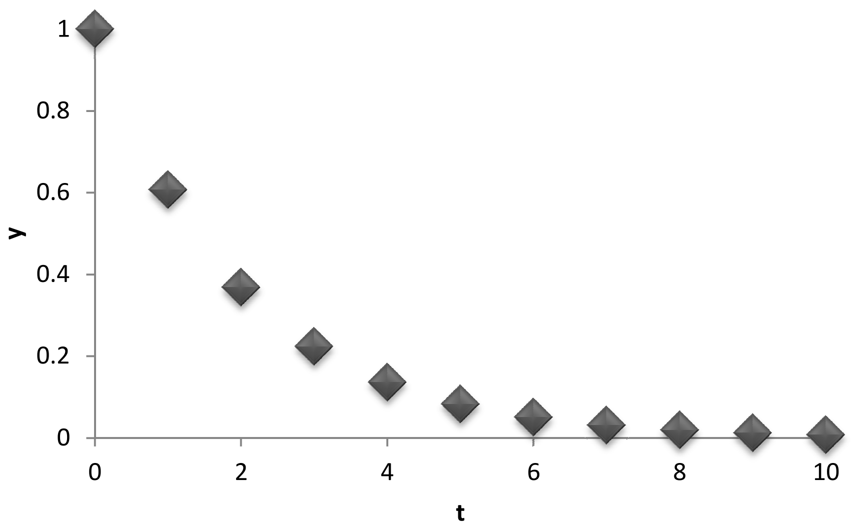
Initial version: November 2013

Unfit version: 2 or later

This tutorial will show you how to write your own cost function for Unfit using your own model and data. Here we will start with some data which consists of observations ***y*** made at times ***t***.

y	1.0	0.6065	0.3679	0.2231	0.1353	0.0821	0.0498	0.0302	0.0183	0.0111	0.0067
t	0	1	2	3	4	5	6	7	8	9	10

If we plot these data they look like this:



To this we will try to fit a two parameter exponential model of the form:

$$y(t) = Aexp(-Bt)$$

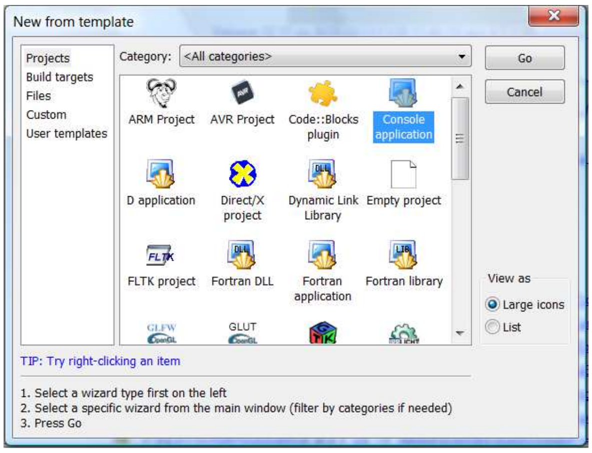
Our goal is to find the values of the two constants in this equation, A & B, such that we get the best possible fit to our data.

Getting Started

The easiest way to get started with your own fitting problem is to create a new Code::Blocks project. If you are familiar with adding search paths etc in Code::Blocks then you can put the new project wherever you like. If you are a newbie, we suggest creating your project in the directory where you placed Unfit. Through the menus at the top of Code::Blocks, simply go

File -> New -> Project

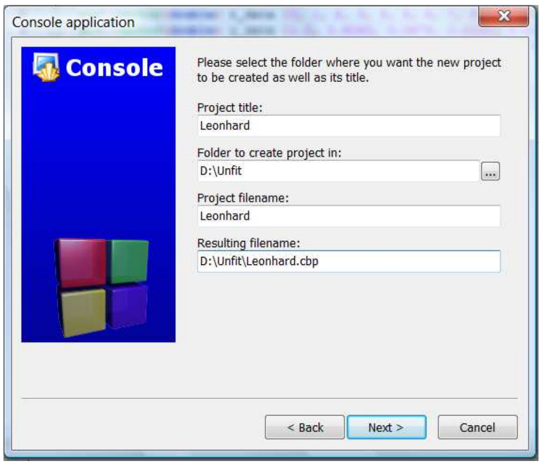
This will bring up a dialogue box, and we want a Console Application.



TIP: Try right-clicking an item

1. Select a wizard type first on the left
2. Select a specific wizard from the main window (filter by categories if needed)
3. Press Go

Hit the Go button. The next screen asks about which language you want to use. We want C++. After that we need the project title and location. Although Unfit was developed on Linux, it also works under Windows. Here Unfit is located in D:\Unfit, and the project we are making is called Leonhard. (The name was chosen because we are fitting an exponential and it is believed the constant ***e*** is named after Leonhard Euler).



Please select the folder where you want the new project to be created as well as its title.

Project title: Leonhard

Folder to create project in: D:\Unfit

Project filename: Leonhard

Resulting filename: D:\Unfit\Leonhard.cbp

The next dialogue asks you to select a compiler. The default is usually fine, so there is no need to change anything. Just click the Finish button. By default you should get a project populated with a main.cpp which looks something like:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

You can try compiling (building) and running this as-is to see that it works by hitting the F9 key. A window should pop up and display the standard “Hello world!”.

Generic Cost Function

In order to do any fitting or optimization in Unfit, your cost function must derive from our GenericCostFunction, which can be found in include/GenericCostFunction.hpp. This is important as it defines the interface through which the optimizers access your cost function(s). In particular, there is one line that we must obey:

```
virtual std::vector<double> operator()(const std::vector<double> &x) = 0;
```

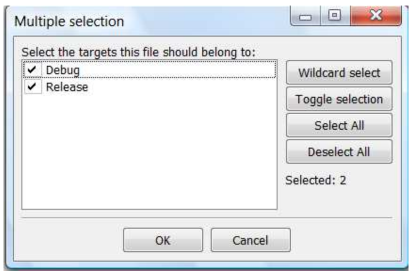
The technical description is that this is a pure virtual function, which overloads **`operator()`**making this class is a functor, or function object. If this makes no sense to you, don't worry. The non-technical description is that we need to include this method in our class, and this is where we put our model.

Your Own Cost Function

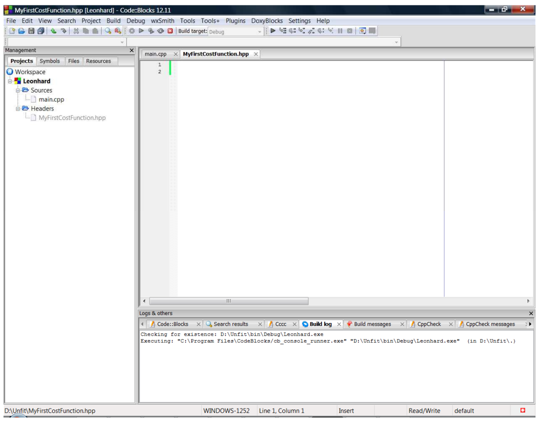
First we must make a new header file for our cost function class. It is good practice to make the name of the header file match the name of the class it contains. (Some languages even force this behavior). Here we will make a class for our cost function called MyFirstCostFunction, so in Code::Blocks we can select

File -> New -> Empty File

A dialogue will pop up asking us if we want to add this file to the project. The answer is yes. Next we will set the file name to be MyFirstCostFunction.hpp, while checking that it is going to put the file in our Unfit directory. After that it will pop up a dialogue asking about targets, e.g.



Just check all of the boxes and click OK. You should now be staring at an empty file in Code::Blocks that looks something like this:


MyFirstCostFunction.hpp [Leonhard] - Code::Blocks 12.11
File Edit View Search Project Build Debug wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Build target: Debug
Management
Projects Symbols Files Resources
Workspace
Leonhard
Sources
main.cpp
Headers
MyFirstCostFunction.hpp
main.cpp
MyFirstCostFunction.hpp
Logs & others
Code::Blocks Search results Cccc Build log Build messages CppCheck CppCheck messages
D:\Unfit\MyFirstCostFunction.hpp WINDOWS-1252 Line 1, Column 1 Insert Read/Write default

```
Checking for existence: D:\Unfit\bin\Debug\Leonhard.exe
Executing: "C:\Program Files\CodeBlocks/cb_console_runner.exe" "D:\Unfit\bin\Debug\Leonhard.exe"  (in D:\Unfit\.)
```

In general I try to avoid preprocessor directives whenever possible (especially macros), but header guards are the one place where they are the gold-standard. To avoid our header file from being included more than once, we should wrap our code with the following:

```
#ifndef MYFIRSTCOSTFUNCTION_HPP_
#define MYFIRSTCOSTFUNCTION_HPP_

 // All of our code

#endif
```

Note that the capitalized term (`MYFIRSTCOSTFUNCTION_HPP_`) needs to be a unique identifier which is why we based it on the name of the file/class.

As mentioned earlier, all of our cost function classes must derive from the GenericCostFunction class. The code to do this looks like this:

```
class MyFirstCostFunction : public Unfit::GenericCostFunction
```

The most common newbie mistake is to forget to add the `public`, the `Unfit::`, or both. The `public` keyword states that anything that is public (accessible to anyone) in the GenericCostFunction class should also be public in our cost function class. To keep the core Unfit code separate from your code we have wrapped it in an Unfit namespace, which means to access the Unfit GenericCostFunction, you need to include the `Unfit::` prefix.

Next we need to create our own implementation of the **`operator()`**method which is mandated by the GenericCostFunction class. In terms of code, the starting template looks like this:

```
class MyFirstCostFunction : public Unfit::GenericCostFunction
{
 public:
  std::vector<double> operator()(const std::vector<double> &x)
  {
    // Implement the model here
  }
};
```

The problem statement at the start of this tutorial specified that in this case we have of observations ***y*** at times ***t***. Our model needs to be able to access these data. There are several ways to do this but we would recommend storing the data in two `std::vector` containers, and making these private so nothing can accidentally modify them from outside this class. Here we add three lines to the bottom of our class definition to get:

```
class MyFirstCostFunction : public Unfit::GenericCostFunction
{
 public:
  std::vector<double> operator()(const std::vector<double> &x)
  {
    // Implement the model here
  }
 private:
  std::vector<double> t;
  std::vector<double> y;
};
```

Residuals and the Model

Unfit requires our cost function to return a vector of residuals, which represent the linear distance between the data points we have, and our model's prediction of those data points. If we denote our model by ***M***, then our residuals, ***r***, can be written as

$$r_i = y_i - M(t_i)$$

which is simply saying that for an observation, y_i, the residual associated with that observation, r_i, is simply the difference between the observed value and the value predicted by the model at the same time, $M(t_i)$. Practically, a nice compact way to write this in the code is to break this into two parts.

$$r_i = y_i \text{ then } r_i -= M(t_i)$$

Or in code:

```
    auto r = y;
    for (auto i = 0u; i < y.size(); ++i) {
      auto model = .....; // Implement model here
      r[i] -= model;
    }
```

Here we are making use of the C++11 keyword `auto`, which means less typing and potentially less typos. We have 11 data points in our problem, so we could loop from 0 to 10, but it is better to just use the size of our data vector (`y.size()`) as then we can use the same implementation for data sets of different sizes with no code changes. Finally, we need to implement our model, which is a two parameter exponential. The current estimates of two unknown parameters in our model, A & B, are passed in through the ***x*** vector. It is very important that the size of the ***x*** vector is the same as the number of unknown parameters in the model. Unfit looks at the size of the ***x*** vector to determine how many unknowns there are. Here we will assume that x[0] corresponds to A, and x[1] corresponds to B. The model implementation then looks like:

```
      auto model = x[0]*exp(-x[1]*t[i]);
```

Note that the indices to the ***x*** vector do not change with time, but time does change as denoted by the ***i*** index on the ***t*** vector. The complete method implementation looks like this, remembering that in C++11 return by value is the best option:

```
      std::vector<double> operator()(const std::vector<double> &x)
      {
        auto r = y;
        for (auto i = 0u; i < y.size(); ++i) {
          auto model = x[0]*exp(-x[1]*t[i]);
          r[i] -= model;
        }
        return r;
      }
```

The Data

At this stage we have the code for our residual calculations done, but we still haven't dealt with how to give the data to our cost function. Again there are several ways to do this, but an easy way is to just pass the data in to the cost function class through its constructor. In terms of code, this means adding another method to our class which has the same name as our class, i.e.,

```
public:
 MyFirstCostFunction(const std::vector<double> &t_data,
                     const std::vector<double> &y_data)
 {  }
```

Now all we have to do is to make a local copy of the data. The easiest way to do this looks like:

```
MyFirstCostFunction(const std::vector<double> &t_data,
                    const std::vector<double> &y_data)
{
  t = t_data;
  y = y_data;
}
```

Tips for experts: It is more efficient to use an initializer list, so if you know how to do that then go for it. If you do this, then you can also add the **`const`** prefix to your data vectors (t, y) which adds the additional guarantee that you can't overwrite/alter your data accidentally while you are calculating your residuals. You could also sink your data into the cost function using something like a `std::unique_ptr` if you don't want to make a copy, but this is probably only worth it for large data sets. If you are the alpha male/female in C++, try implementing the cost function without any loops by using `std::transform` and a lambda function.

Remember to include the necessary header files at the top of your file (between the #define header guard and class definition):

```
#include <cmath>
#include <vector>
#include "include/GenericCostFunction.hpp"
```

These are needed so we can use the `exp` function, `std::vector`, and the `GenericCostFunction` class, respectively. Apart from intelligent comments, so next year you can still remember what your class does, the cost function class is essentially done. A complete code listing can be found at the end of this tutorial.

The Main Program

The first thing I do the main file (main.cpp) in any new Code::Blocks project is to delete the line which says:

```
using namespace std;
```

This is personal preference, but I think it is useful to be able to recognise exactly what we are using. While this is probably not necessary for things like `cout`, many numerical packages will have, for example, their own way of representing vectors, so I like to see `std::vector` instead of just `vector` for clarity.

We know that our program will need to use vectors, our own cost function, and one or more of the optimizers in Unfit, so we must add three headers in addition to the `iostream` header that is already there:

```
#include <iostream>
#include <vector>
#include "MyFirstCostFunction.hpp"
#include "include/Unfit.hpp"
```

In terms of our data, we can just type in the data as there is not much of it. If you need to read data from a file, Unfit provides a DataFileReader class. Have a look at the source (cpp) files in the examples directory of Unfit to see how to do this. Most of those examples read data from files. Here we can again make use of C++11 initializer lists:

```
int main()
{
  std::vector<double> t_data {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  std::vector<double> y_data {1.0, 0.6065, 0.3679, 0.2231, 0.1353, 0.0821,
                              0.0498, 0.0302, 0.0183, 0.0111, 0.0067};
  ...
```

Next, we need to create a cost function object and pass it the data. Remember that the order of the data vectors must match with the constructor for the cost function. In the example below, remember also that the object we can work with is called `cost_func`, and not `MyFirstCostFunction`, which is the name of the class.

```
MyFirstCostFunction cost_func(t_data, y_data);
```

The other thing we need before we begin optimizing is an initial guess for our unknown parameters. We have two of these, so we must define a vector of length two. Sometimes we have no idea what a good initial guess would be, so we can just try a vector of zeros or ones. Here we can choose zeros.

```
std::vector<double> x_guess {0.0, 0.0};
```

For the last part we must choose which of the optimizers to try. The options are a Nelder-Mead simplex technique, the Levenberg-Marquardt algorithm, Differential Evolution or a Genetic Algorithm approach. Note that the values in the initial guess we provide make a big difference to Nelder-Mead and Levenberg-Marquardt, but do not affect the Differential Evolution and Genetic Algorithm optimizers. Here we will first choose Nelder-Mead so we must create a NelderMead object, which we will call ***nm***.

```
Unfit::NelderMead nm;
```

It is at this point that we can change the way the optimizer behaves, if we want to. This is usually done by adding lines such as:

```
nm.options.XXXX
nm.bounds.YYYY
```

A common (and useful) option is to have the optimizer print out iteration by iteration information and then the final solution. This is achieved with the line:

```
nm.options.SetOutputLevel(3);
```

Here we will just use the default behaviour. Each of the optimizers has a method called ***FindMin***, and this is what we must call for the optimization to happen. The two arguments are simply our cost function object and our initial guess.

```
auto rc = nm.FindMin(cost_func, x_guess);
```

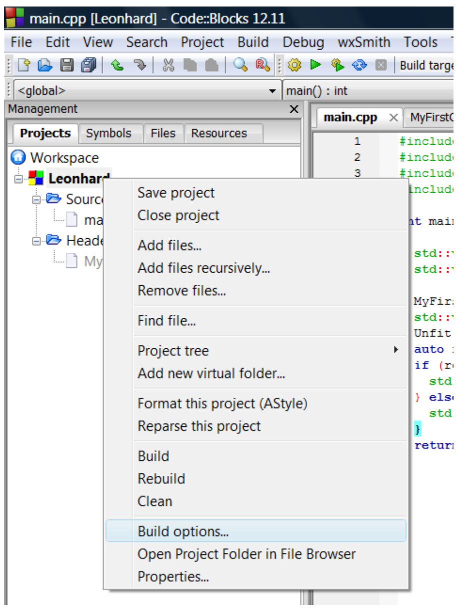
The answer (optimized parameters) will be returned in the x_guess vector. These optimization problems can be tricky and there is no guarantee all, or any of the optimizers will work for your problem. Therefore it is important to check the return code from ***FindMin***. It will be zero if there were no problems. If it is non-zero you will need to check the documentation for that particular optimizer to see what the return code means. Please also note that even if the return code is zero, that does not guarantee that a good solution has been found. Sometimes the algorithms get stuck at a local minimum and think they have converged when in reality the fit is not that good. Our main function should look something like this:

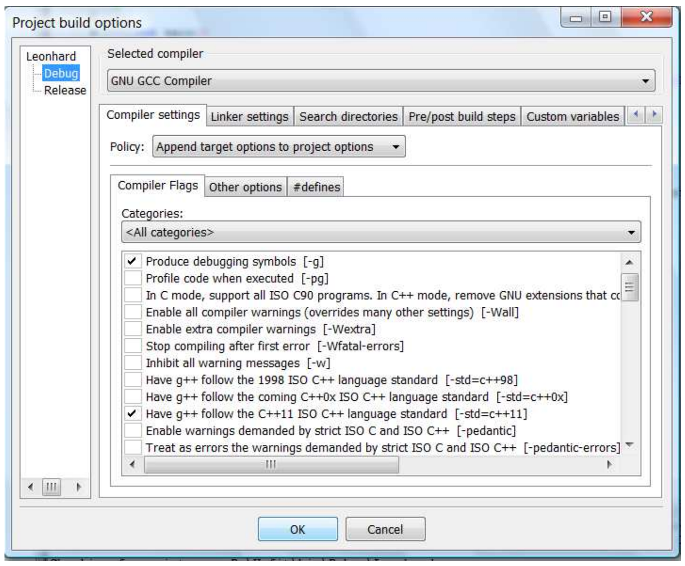
```
std::vector<double> t_data {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::vector<double> y_data {1.0, 0.6065, 0.3679, 0.2231, 0.1353, 0.0821,
                            0.0498, 0.0302, 0.0183, 0.0111, 0.0067};
MyFirstCostFunction cost_func(t_data, y_data);
std::vector<double> x_guess {0.0, 0.0};
Unfit::NelderMead nm;
auto rc = nm.FindMin(cost_func, x_guess);
if (rc == 0) {
  std::cout << "Answer: " << x_guess[0] << " " << x_guess[1] << std::endl;
} else {
  std::cout << "Oh dear. RC = " << rc << std::endl;
}
return rc;
```

Again a complete code listing can be found at the end of the tutorial.

Compile and Run

Two need to be done before the code can compile and run. The first is that by default, at least for the GCC compiler, C++11 is not enabled by default. To enable it you need to go to the project name, right-click and select "Build options…" as shown below (left). This will bring up a dialogue box as shown below (right). You need to check the box next to the "Have g++ follow the C++11 ISO C++ language standard". On older versions of Code::Blocks (e.g. 10.08) this option is not available, but the C++0x option should work.


main.cpp [Leonhard] - Code::Blocks 12.11
File Edit View Search Project Build Debug wxSmith Tools
Management
Projects Symbols Files Resources
Workspace
Leonhard
Save project
Close project
Add files...
Add files recursively...
Remove files...
Find file...
Project tree
Add new virtual folder...
Format this project (AStyle)
Reparse this project
Build
Rebuild
Clean
Build options...
Open Project Folder in File Browser
Properties...



Selected compiler: GNU GCC Compiler

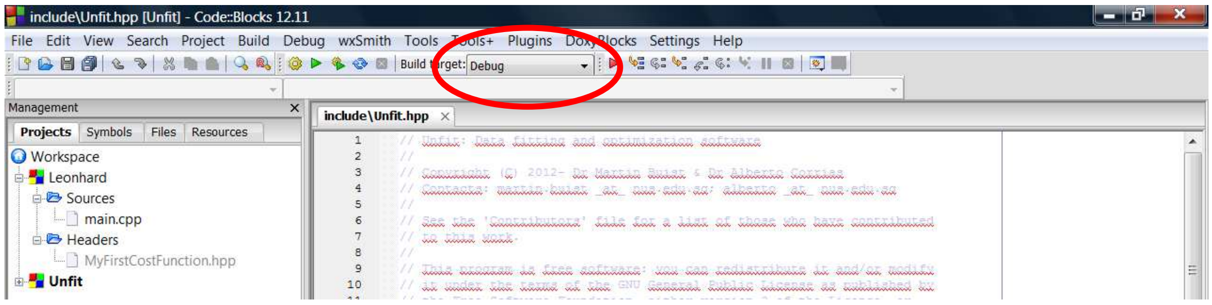
Policy: Append target options to project options

Categories: <All categories>

- [✓] Produce debugging symbols [-g]
- [ ] Profile code when executed [-pg]
- [ ] In C mode, support all ISO C90 programs. In C++ mode, remove GNU extensions that co
- [ ] Enable all compiler warnings (overrides many other settings) [-Wall]
- [ ] Enable extra compiler warnings [-Wextra]
- [ ] Stop compiling after first error [-Wfatal-errors]
- [ ] Inhibit all warning messages [-w]
- [ ] Have g++ follow the 1998 ISO C++ language standard [-std=c++98]
- [ ] Have g++ follow the coming C++0x ISO C++ language standard [-std=c++0x]
- [✓] Have g++ follow the C++11 ISO C++ language standard [-std=c++11]
- [ ] Enable warnings demanded by strict ISO C and ISO C++ [-pedantic]
- [ ] Treat as errors the warnings demanded by strict ISO C and ISO C++ [-pedantic-errors]

The other thing you need to do to get the code to compile (or more technically, link) is to tell it where to find the Unfit library. At this point we will assume that you have not built the Unfit library and briefly explain how to do it.

To build the Unfit library, in Code::Blocks, simply go File -> Open and open the Unfit.cbp project file. Somewhere at the top of the screen there will be a drop-down list that will say either Debug or Release:

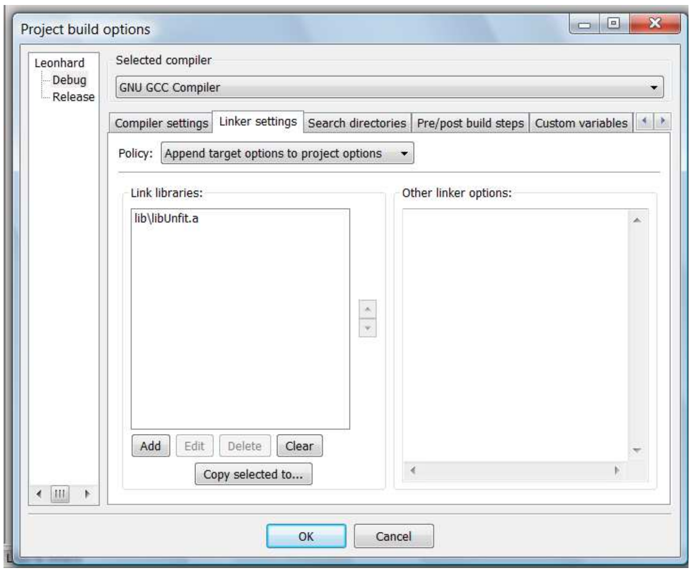

include\Unfit.hpp [Unfit] - Code::Blocks 12.11

Build target: Debug

```
// Unfit: Data fitting and optimization software
//
// Copyright (C) 2012- Dr Martin Buist & Dr Alberto Corrias
// Contacts: martin.buist _at_ nus.edu.sg; alberto _at_ nus.edu.sg
//
// See the 'Contributors' file for a list of those who have contributed
// to this work.
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
```

Change this to Library, then you can press CTRL-F9 to compile. This will create libUnfit.a in a directory called lib (which will be in the same place as Unfit.cbp). You can then close the Unfit project.

To tell our project where to find the library we must again right-click on the project name and bring up the Build options. The resulting dialogue has a tab named Linker Settings that we need to select.


Project build options

Selected compiler: GNU GCC Compiler

Policy: Append target options to project options

Link libraries: lib\libUnfit.a

Other linker options:

From there, we must add a link library. Click on Add and it will bring up a dialogue from which you can navigate to find libUnfit.a. Click OK when the library has been successfully added. You should now go up to the File menu in Code::Blocks and choose the “Save Everything” option to make sure we don’t lose these changes.

Now you can press F9 to compile and run the code. That is all there is to it. The answer should be approximately (1.0, 0.5).

Extras

If you want to try the Levenberg-Marquardt algorithm, just replace the lines:

```
Unfit::NelderMead nm;
auto rc = nm.FindMin(cost_func, x_guess);
```

with

```
Unfit:: LevenbergMarquardt lm;
auto rc = lm.FindMin(cost_func, x_guess);
```

Both Differential Evolution and Genetic Algorithm require bounds, and the default bounds are out at +/- the maximum number the computer can store, so we need to set some sensible bounds. For Differential Evolution, we can set sensible bounds for this problem at +/- 10 and run using:

```
Unfit::DifferentialEvolution de;
de.bounds.SetBounds(0, -10.0, 10.0);
de.bounds.SetBounds(1, -10.0, 10.0);
auto rc = de.FindMin(cost_func, x_guess);
```

To try Genetic Algorithm we can use:

```
Unfit::GeneticAlgorithm ga;
ga.bounds.SetBounds(0, -10.0, 10.0);
ga.bounds.SetBounds(1, -10.0, 10.0);
auto rc = ga.FindMin(cost_func, x_guess);
```

For this tutorial all of the algorithms can perform the fit but this is often not the case. Both Nelder-Mead and Levenberg-Marquardt are local minimizers that rely on having a good initial guess. In general we find Levenberg-Marquardt faster for our problems and Nelder-Mead more robust. Differential Evolution and Genetic Algorithm are global optimizers and are usually significantly slower than the other two. Often Genetic Algorithm requires a lot of parameter tuning to get it to work well, thus we more commonly use Differential Evolution. Remember that these two are stochastic in nature and may converge to different solutions when using different sequences of random numbers (different seeds). Because the global optimizers are more likely to find a global minimum and the local optimizers are faster at getting there a hybrid approach has become popular. The equivalent driver code for a hybrid Genetic-Algorithm-Levenberg-Marquardt approach is simply:

```
Unfit::GeneticAlgorithm ga;
ga.bounds.SetBounds(0, -10.0, 10.0);
ga.bounds.SetBounds(1, -10.0, 10.0);
auto rc = ga.FindMin(cost_func, x_guess);
Unfit:: LevenbergMarquardt lm;
rc = lm.FindMin(cost_func, x_guess);
```

…and you have your answer.

MyFirstCostFunction.hpp:

```
#ifndef MYFIRSTCOSTFUNCTION_HPP_
#define MYFIRSTCOSTFUNCTION_HPP_

#include <cmath>
#include <vector>
#include "include/GenericCostFunction.hpp"

class MyFirstCostFunction : public Unfit::GenericCostFunction
{
 public:
  MyFirstCostFunction(const std::vector<double> &t_data,
                      const std::vector<double> &y_data)
  {
    t = t_data;
    y = y_data;
  }

  std::vector<double> operator()(const std::vector<double> &x)
  {
    auto r = y;
    for (auto i = 0u; i < r.size(); ++i) {
      auto model = x[0]*exp(-x[1]*t[i]);
      r[i] -= model;
    }
    return r;
  }
 private:
  std::vector<double> t;
  std::vector<double> y;
};

#endif
```

main.cpp:

```cpp
#include <iostream>
#include <vector>
#include "MyFirstCostFunction.hpp"
#include "include/Unfit.hpp"

int main()
{
  std::vector<double> t_data {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  std::vector<double> y_data {1.0, 0.6065, 0.3679, 0.2231, 0.1353, 0.0821,
                              0.0498, 0.0302, 0.0183, 0.0111, 0.0067};
  MyFirstCostFunction cost_func(t_data, y_data);
  std::vector<double> x_guess {0.0, 0.0};
  Unfit::NelderMead nm;
  auto rc = nm.FindMin(cost_func, x_guess);
  if (rc == 0) {
    std::cout << "Answer: " << x_guess[0] << " " << x_guess[1] << std::endl;
  } else {
    std::cout << "Oh dear. RC = " << rc << std::endl;
  }
  return rc;
}
```