How to Compile Simple EPSO C++ code – version 2009

This document isn't dealing with fundamentals nor theory of evolutionary computation or EPSO. To do so, one is instructed to read other available works related to EPSO algorithm. Instead, this is just a practical "walkthrough" to setting up the compilation of EPSO in C++.

Hrvoje Keko hkeko@inescporto.pt

Introduction, requirements and specifics

There are three ingredients for a successful compilation of EPSO:

- EPSO code itself, of course
- STL-compliant C++ compiler
- Boost libraries

This EPSO implementation can be compiled on virtually any modern C++ compiler.
It relies on Standard Template Library, a ready made set of common classes, that is included in the C++ standard.
http://en.wikipedia.org/wiki/Standard_Template_Library

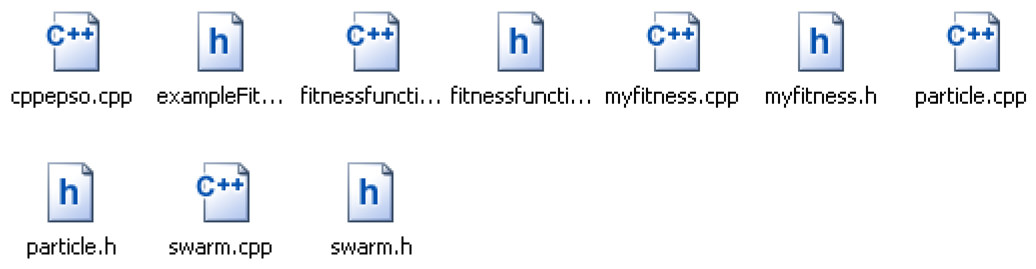
A good introductory STL tutorial can be found here, on a very resourceful CodeProject community website:
http://www.codeproject.com/KB/stl/stlintroduction.aspx

Even though this EPSO implementation was mostly developed by using Microsoft's excellent Visual C++ 2005 development environment, it doesn't use any Microsoft platform specifics. Thus virtually any newer C++ compiler could be used in compiling EPSO.

The only compiler requirements are related to STL and Boost libraries. This is why some older compilers, like Microsoft's Visual C++ 6.0 from 1998, may have some difficulties. So it may be more convenient to use a more recent compiler. Even more so since newer ones (like MS Visual C++ 2002 and newer) largely facilitate debugging of STL code.

In this document two examples of setting EPSO compilation are shown: setting up a project in Microsoft Visual Studio 2008 and in Bloodshed DevC++.

EPSO 2009 C++ files overview



The EPSO 2009 consists of several C++ files. There are three base classes of EPSO, some examples on how to write a fitness function, and a base program that instantiates and runs the EPSO.

- cppepso.cpp
  this is the **main program**, that instantiates the swarm object and runs the basic algorithm loop calling the methods of class swarm
- swarm.h and swarm.cpp
  the **swarm class** controls the particle movements, communication between particles, and basically encapsulates the whole EPSO algorithm
- particle.h and particle.cpp
  these files define **particle class:** this class implements movement equation, mutation of strategic parameters etc – all of these are controlled inside the swarm class – so the swarm class is "responsible" for managing particles
- fitnessfunction.h and fitnessfunction.cpp
  this class is the abstract class (i.e. it can not be instantiated) for the fitness function, it has the virtual fitness function
- exampleFitness.h
  this file contains several classic example functions – these are extensions of the abstract fitness function class above that implement the Rosenbrock, Alpine, etc...
- myFitness.h and myFitness.cpp – the simplest example possible on how to develop a fitness function suitable for EPSO.

The rationale is as follows – the *swarm* and *particle* classes can be used *as-is,* but the user has to develop own fitness function that will extend and implement the calculation of fitness. To make it as simple as possible, some examples are included in exampleFitness and in myFitness files, and described later in this document.

Even though it may not be the most elegant solution, the EPSO algorithm only instantiates a single instance of fitness function object and then in the swarm class passes around just a pointer to that particular instance of fitness function object. This was chosen instead of instancing the fitness function with each particle, since in real-life fitness functions can be "heavy".
In this 2009 version most of the functionality is encapsulated within the swarm class. Curious users might find the insides of swarm and particle classes more interesting.

Boost libraries

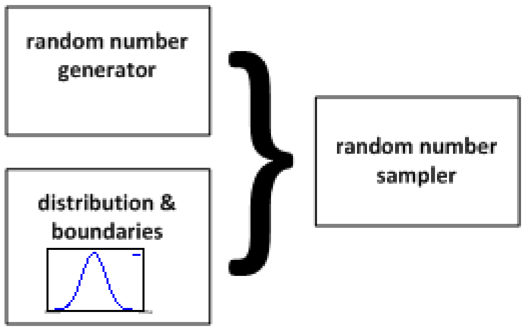
One thing in particular is specific to this code. Besides using the STL libraries, it uses Random library from an excellent set of Boost libraries (www.boost.org).

Boost is set of peer-reviewed, high quality C++ libraries that conveniently extend the functionality of plain C++. Boost provides a broad set of applications, and heavily relies on C++ templates and generic programming. It is also pretty extensively and meticulously tested.

It is well known that the performance of metaheuristic algorithms is affected by quality of a random number generator being used. Thus this version of EPSO is “complicated” by using the Boost's **Random** library.

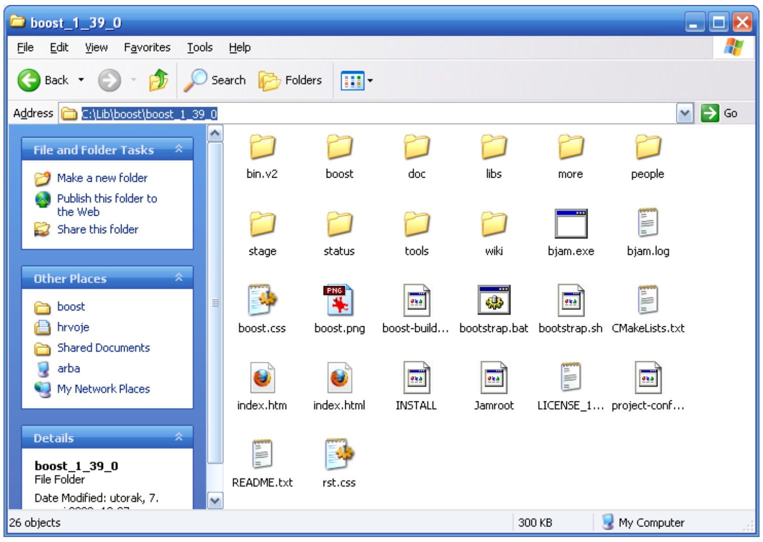
Another benefit of the Boost Random library is availability of random number distributions like Gaussian or log-normal. So using Boost Random means using a convenient usage of very good random number generator coupled with a variety of distributions.

Although this is not entirely necessary for pure usage of EPSO, this is how Boost Random is used in EPSO code.
A basic example of its usage is here :
http://en.wikipedia.org/wiki/Boost_C%2B%2B_Libraries#Generating_random_numbers_.E2.80.93_Boost.Random



At first, one object of random number generator (RNG) class is instanced.
In the whole EPSO process, there is just a single instance of RNG being used. This is why a reference to RNG is constantly being passed in the swarm class code.
This way one can control the EPSO process – by controlling the RNG. All of this is done inside the swarm class – so the user that only creates a fitness function doesn't need to know a lot about this.
Next, a desired distribution (Gaussian, uniform, log-normal...) is instanced. The distribution object takes care that the random numbers fit into desired distribution and boundaries.
Finally, a sampler object is instanced. It couples the RNG with the distribution, and it is the sampler that gets called for to get the random numbers.

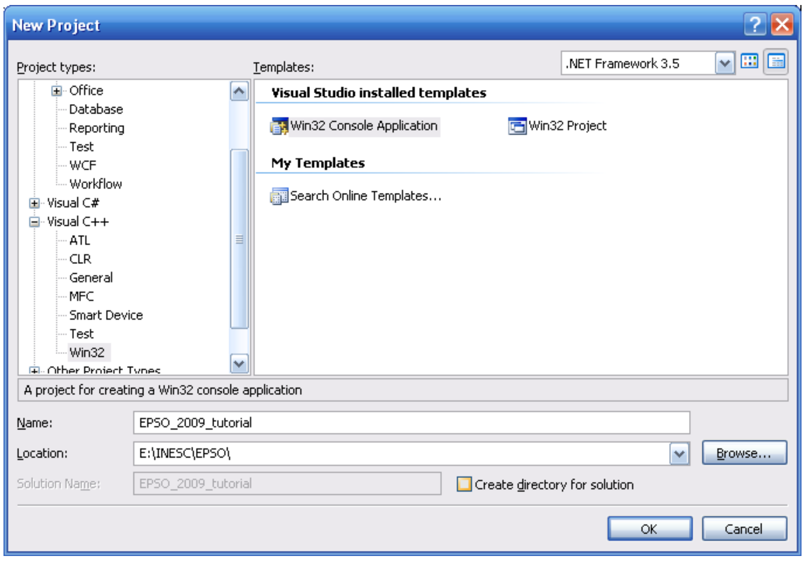
Boost libraries along with their further documentation can be freely obtained from www.boost.org
Once the download is finished, extract them to a folder on your hard drive.
In this example, **C:\Lib\Boost\boost_1_39_0** is used as Boost libraries location.



Compiling EPSO using Microsoft Visual C++ 2008

Several versions of Microsoft Visual Studio are available through University of Porto MSDN Academic Alliance program. More details are on the website of University of Porto (www.up.pt) While the following example shows screen captures from Visual Studio 2008, there's no significant difference in setting up EPSO in other versions of Visual Studio.

Start the Visual C++ environment, and then choose to create a new project (File / New Project). For now, choose the simplest, console application, and be sure to create an empty project, without any automatically generated code.



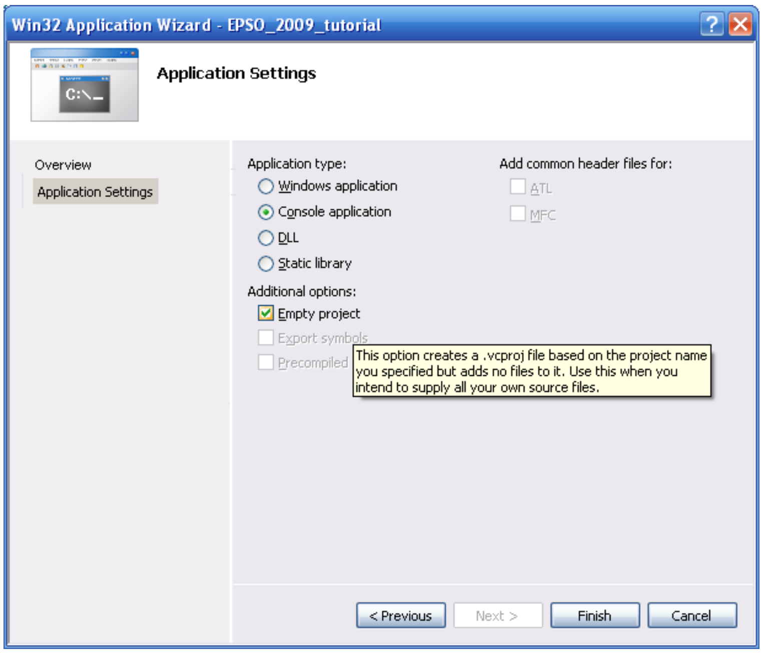
A project for creating a Win32 console application

Name: EPSO_2009_tutorial

Location: E:\INESC\EPSO\

Solution Name: EPSO_2009_tutorial

[ ] Create directory for solution



Application type:

Windows application [✓ Console application] DLL Static library

Add common header files for:

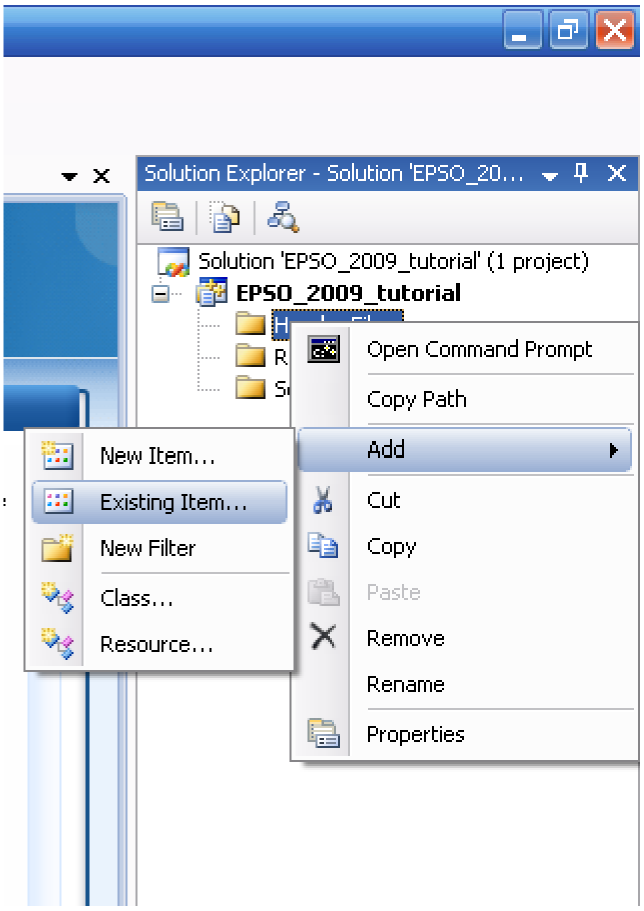
[ ] ATL
[ ] MFC

Additional options:

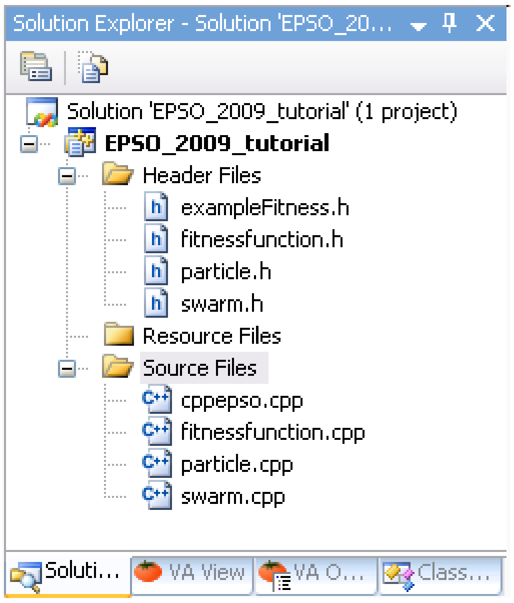
[✓] Empty project
[ ] Export symbols
[ ] Precompiled

This option creates a .vcproj file based on the project name you specified but adds no files to it. Use this when you intend to supply all your own source files.

Now, add the EPSO code to the project, by using Add existing items in Solution Explorer, usually on right hand side of Visual Studio.



After the EPSO code is added to the C++ project, the solution explorer would look somewhat like this:

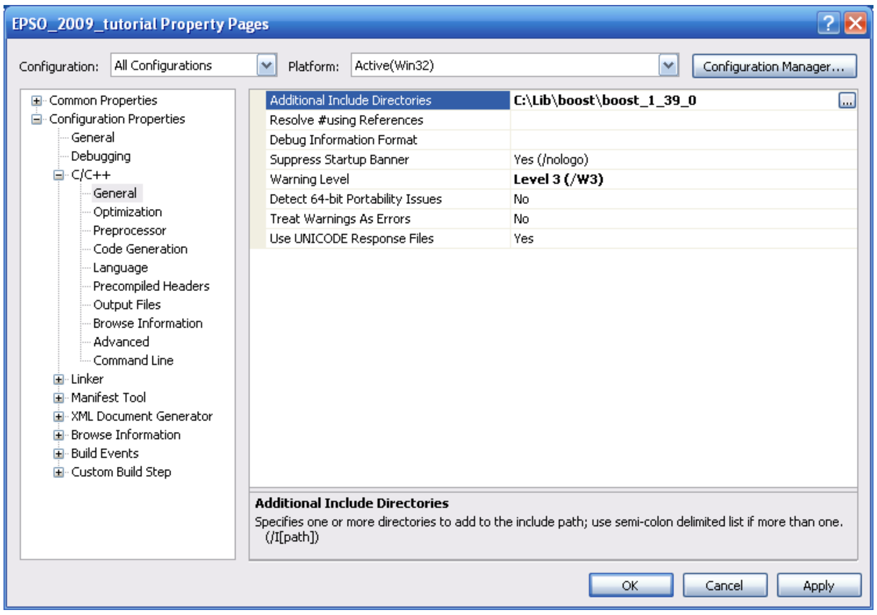


Solution Explorer - Solution 'EPSO_20...

```
Solution 'EPSO_2009_tutorial' (1 project)
EPSO_2009_tutorial
    Header Files
        exampleFitness.h
        fitnessfunction.h
        particle.h
        swarm.h
    Resource Files
    Source Files
        cppepso.cpp
        fitnessfunction.cpp
        particle.cpp
        swarm.cpp
```

The EPSO project properties now have to be changed so that the compiler knows where to find the Boost libraries. In Solution Explorer, right-click the project name – the project name is EPSO_2009_tutorial in the screenshots. Then select Properties.

In Configuration Properties / C++ / General – just like in the following screenshot, add the Boost directory to "Additional Include Directories". By the way, make sure you select "All Configurations" in the property pages, so that you add Boost directory both to Debug and Release compilation settings.



EPSO_2009_tutorial Property Pages

Configuration: All Configurations   Platform: Active(Win32)   Configuration Manager...

```
Common Properties
Configuration Properties
    General
    Debugging
    C/C++
        General
        Optimization
        Preprocessor
        Code Generation
        Language
        Precompiled Headers
        Output Files
        Browse Information
        Advanced
        Command Line
    Linker
    Manifest Tool
    XML Document Generator
    Browse Information
    Build Events
    Custom Build Step
```

Additional Include Directories	C:\Lib\boost\boost_1_39_0
Resolve #using References	
Debug Information Format	
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level 3 (/W3)
Detect 64-bit Portability Issues	No
Treat Warnings As Errors	No
Use UNICODE Response Files	Yes

Additional Include Directories
Specifies one or more directories to add to the include path; use semi-colon delimited list if more than one. (/I[path])

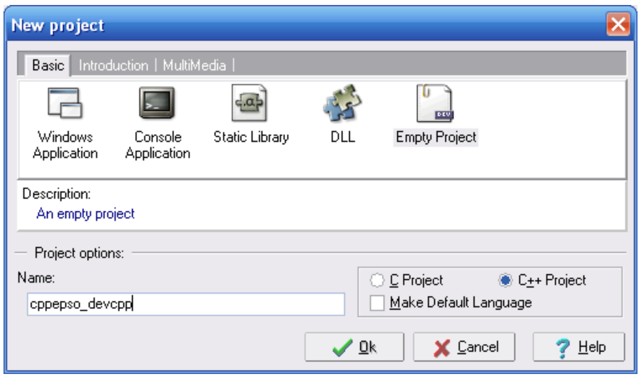
Confirm the change, start the compilation, and EPSO code should now compile without problems. By default, as can be seen from the cppepso.cpp code, the EPSO runs Rosenbrock function optimization.

Now, it's your turn to develop your own fitness function or test the fitness functions bundled with EPSO 2009.

Compiling EPSO using DevC++

Bloodshed DevC++ is a small but versatile, freely available development environment, used widely by collaborators in INESC Porto. It can be downloaded from
http://www.bloodshed.net/devcpp.html

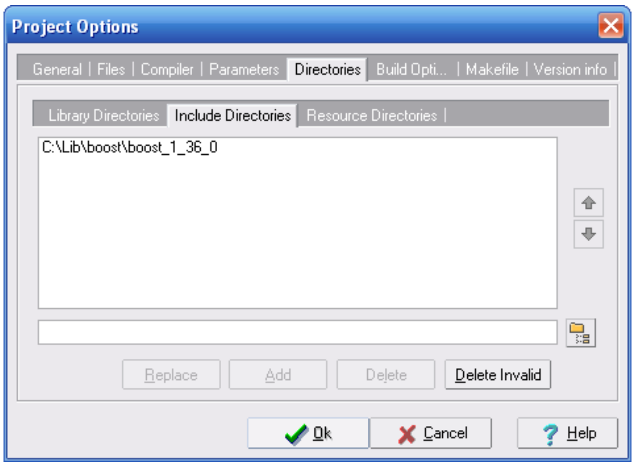
Just like in case of using Microsoft's compiler, the DevC++ compiler also needs to use Boost libraries. To do so, there are two options – download the Boost library from www.boost.org website and
Option 1: Adding manually downloaded Boost library directory to project options
Create a new, empty project and add EPSO files to it.



Description:
An empty project

Project options:
Name: cppepso_devcpp
C Project [✓ C++ Project]
Make Default Language

Choose Project / Project Options and then select Directories.



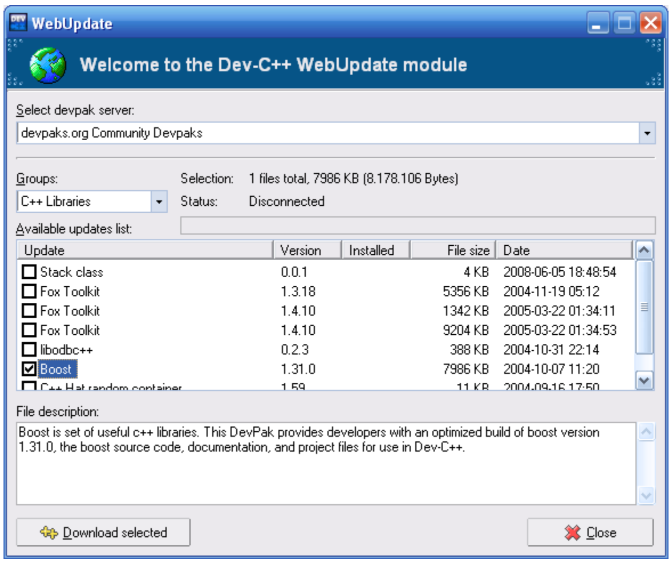
C:\Lib\boost\boost_1_36_0

The directory where you have extracted the Boost library should be added to Include directories, so the compiler can find it. You should now be able to compile EPSO.

Option 2: Downloading Boost libraries from DevC++ package automatically

Besides downloading the libraries from www.boost.org and manually adding them to your EPSO project, DevC++ offers a WebUpdate tool where you can download and install the Boost “package”.

Go to Tools / Check for Updates/Packages, choose devpaks.org Community Devpaks, then C++ Libraries, select Boost and proceed with downloading.



Select devpak server: devpaks.org Community Devpaks

Groups: C++ Libraries

Selection: 1 files total, 7986 KB (8.178.106 Bytes)

Status: Disconnected

Available updates list:

Update	Version	Installed	File size	Date
☐ Stack class	0.0.1		4 KB	2008-06-05 18:48:54
☐ Fox Toolkit	1.3.18		5356 KB	2004-11-19 05:12
☐ Fox Toolkit	1.4.10		1342 KB	2005-03-22 01:34:11
☐ Fox Toolkit	1.4.10		9204 KB	2005-03-22 01:34:53
☐ libodbc++	0.2.3		388 KB	2004-10-31 22:14
☑ Boost	1.31.0		7986 KB	2004-10-07 11:20
☐ C++ Hat random container	1.59		11 KB	2004-09-16 17:50

File description:

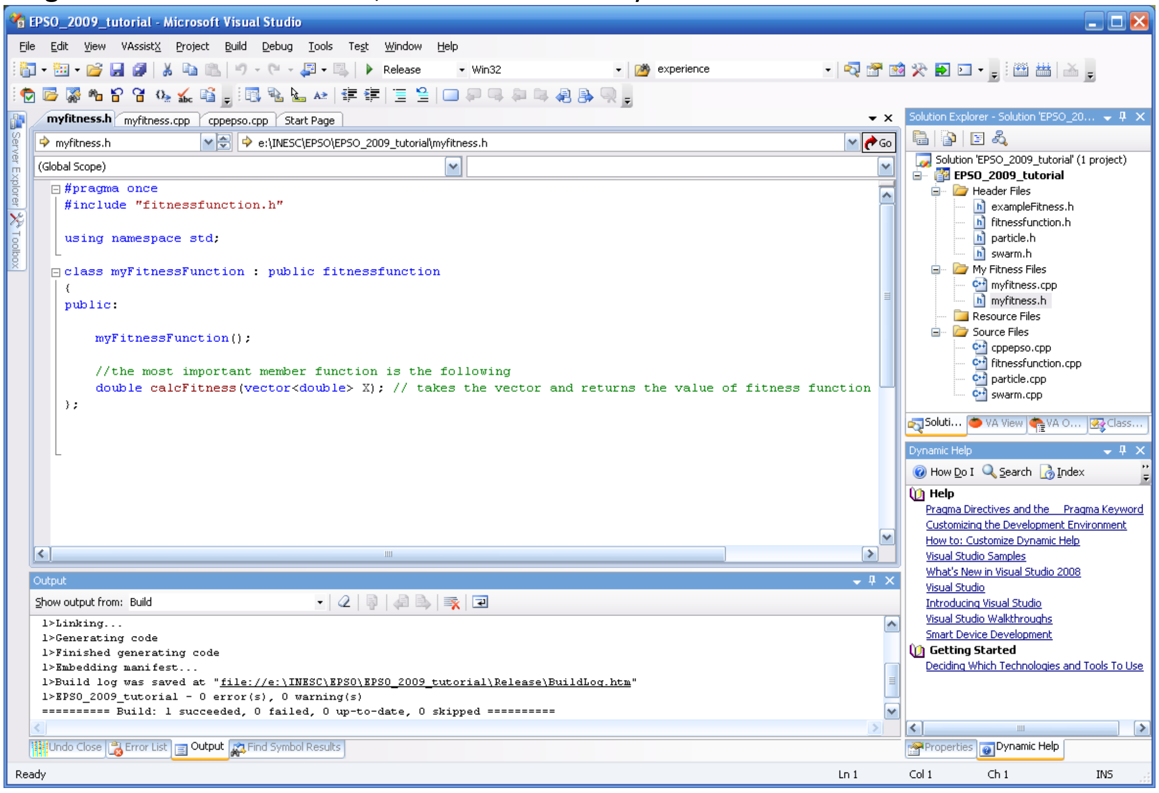
Boost is set of useful c++ libraries. This DevPak provides developers with an optimized build of boost version 1.31.0, the boost source code, documentation, and project files for use in Dev-C++.

Now create a project and add EPSO files to it. You should be able to compile and run EPSO.

Developing Your Own Fitness Function

To develop your own fitness function, you need to develop a class that *public* inherits the abstract fitnessfunction class, and implements the **calcFitness** function, that takes a STL vector of doubles as a parameter. As an example, **myfitness.h** and **myfitness.cpp** are also included in the EPSO 2009 code.

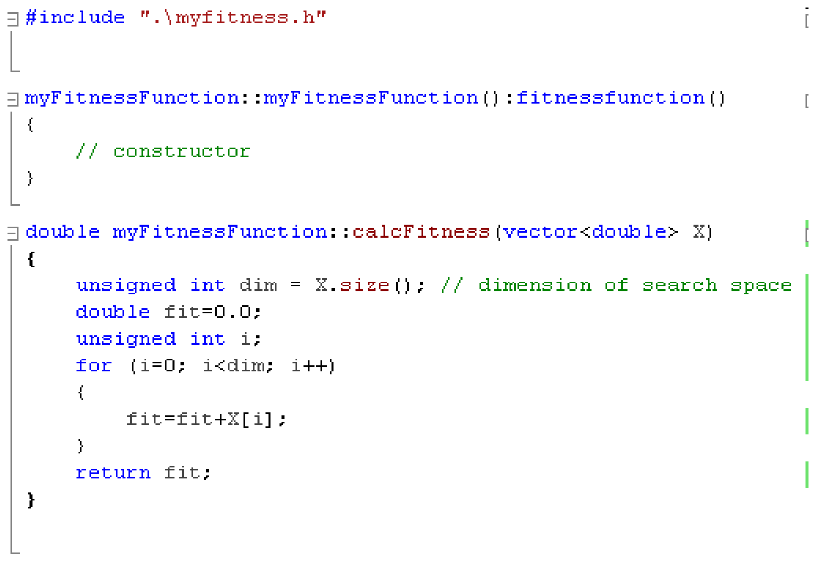
Starting with EPSO 2009 tutorial, one can add the myFitness class to the code.



```
#pragma once
#include "fitnessfunction.h"

using namespace std;

class myFitnessFunction : public fitnessfunction
{
public:

    myFitnessFunction();

    //the most important member function is the following
    double calcFitness(vector<double> X); // takes the vector and returns the value of fitness function
};
```

```
1>Linking...
1>Generating code
1>Finished generating code
1>Embedding manifest...
1>Build log was saved at "file://e:\INESC\EPSO\EPSO_2009_tutorial\Release\BuildLog.htm"
1>EPSO_2009_tutorial - 0 error(s), 0 warning(s)
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

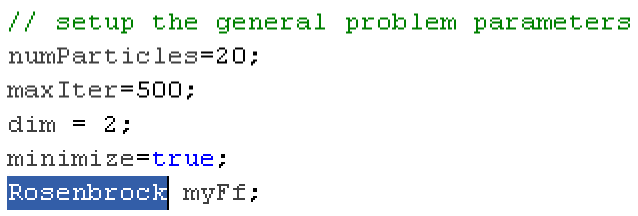
A simple sum of the vector X is the new fitness function implemented as calcFitness member of myFitnessFunction class:



```
#include ".\myfitness.h"

myFitnessFunction::myFitnessFunction():fitnessfunction()
{
    // constructor
}

double myFitnessFunction::calcFitness(vector<double> X)
{
    unsigned int dim = X.size(); // dimension of search space
    double fit=0.0;
    unsigned int i;
    for (i=0; i<dim; i++)
    {
        fit=fit+X[i];
    }
    return fit;
}
```

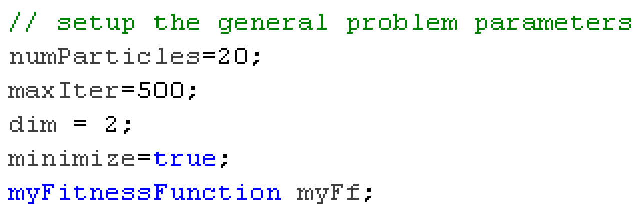
Now one only needs to instantiate the myFitnessFunction object instead of Rosenbrock object in the cppepso.cpp code. The problem parameters can be changed in the same file if one wants to do so.

To use the newly written myFitnessFunction, this



```
// setup the general problem parameters
numParticles=20;
maxIter=500;
dim = 2;
minimize=true;
Rosenbrock myFf;
```

needs to be changed to...



```
// setup the general problem parameters
numParticles=20;
maxIter=500;
dim = 2;
minimize=true;
myFitnessFunction myFf;
```

Of course, the cppepso.cpp also needs to “#include” the myfitness.h file.
That's it, you now have your own fitness function.

In a similar way, you can use other pre-defined fitness functions from exampleFitness.h file (Alpine, Sphere...)

A note on swarm class constructor

Note that you can set the dimensions of search space, not only in this way:

```
// setup search space limits
vector <double> minPos, maxPos;
```

The search space limits can also be set as a plain **double** values, not vectors of doubles:

```
double minPos=0;
double maxPos=100;

// now declare the swarm
swarm epsoSwarm (numParticles, minPos, maxPos, dim, minimize, myFf);
```

The swarm constructor is overloaded:

- if the swarm constructor is called with a **vector** of doubles, that means the search space limits are different for each dimension
- on the contrary, if you call the swarm constructor with **doubles** in place of minPos and maxPos, this means **all the dimensions of search space** are limited in the same manner, i.e the search hyperspace is [minPos, maxPos]^dim

From here, you can continue developing your own EPSO-based application.

Hrvoje Keko
Porto, July 2009
hkeko@inescporto.pt