



Borg MOEA User Guide

Version 1.8

Updated September 23, 2014

Dr. David Hadka

Dr. Patrick M. Reed

Table of Contents

Introduction 3

 Copyright and License Notice..... 3

 Patent Notice..... 3

Enhancements..... 4

Compiling the Borg MOEA 5

 Compiling on Unix/Linux/Cygwin 5

 Compiling on Windows (Visual Studio) 6

A Simple Example..... 10

Supported Programming Languages 14

 Initial Setup 14

 Python..... 16

 Java..... 17

 C# (Visual Studio) 18

 C# (Mono)..... 22

 Matlab 22

 R..... 23

 Speed Comparison..... 23

Troubleshooting 25

Additional Reading Materials 27

Borg MOEA License 29

Introduction

The Borg Multiobjective Evolutionary Algorithm (Borg MOEA) is a state-of-the-art optimization algorithm developed by Dr. David Hadka and Dr. Patrick Reed¹ at the Pennsylvania State University. This package contains the C implementation of the Borg MOEA along with several libraries/plugins for other popular programming languages. The Borg MOEA is characterized as a multi-operator, auto-adaptive algorithm for solving multi- and many-objective problems. The multi-operator aspect refers to the inclusion of multiple search operators (i.e., crossover and mutation operators) and the adaptive selection of these operators at runtime. By adaptively preferring operators that contribute new, innovative solutions to the population, the Borg MOEA learns at runtime which operators or combination of operators performs best. For full details of this algorithm, please refer to the following journal paper:

Hadka, D., and Reed, P.M. "*Borg: An Auto-Adaptive Many-Objective Evolutionary Computing Framework*." *Evolutionary Computation*, 21(2):231-259, 2013.

Please cite this reference in all academic papers and other publications which use or are based on these codes.

Copyright and License Notice

Copyright 2012-2014 The Pennsylvania State University

This software was written by Dr. David Hadka and others.

The use, modification and distribution of this software is governed by The Pennsylvania State University Research and Educational Use License. You should have received a copy of this license along with this program. If not, contact <dmh309@psu.edu>.

Patent Notice

Patent protection for the Borg MOEA is being pursued by The Pennsylvania State University including, but not limited to, United States patent 8,856,054.

¹ Dr. Reed is now a professor in the School of Civil and Environmental Engineering at Cornell University.

Enhancements

Version 1.8 of the Borg MOEA introduces many new features requested by our users, including:

1. **Required epsilons** – Previous versions of the software used a default ϵ value of 0.01, potentially affecting the performance of the algorithm if the ϵ values were scaled poorly. Users are now required to define the ϵ values for their optimization problem, otherwise the program exits with an error message.
2. **Checkpointing** – Saves the current state of the algorithm to a file. For long-running jobs, this allows you to resume an interrupted run (e.g., power outage) at the last saved checkpoint.
3. **Guarding against NaNs** – NaNs, or not-a-number, are special values that can appear in a computer program as a result of invalid mathematical operations (e.g., division by zero). NaNs are particularly bad for Pareto-based MOEAs since any solution with a NaN will always be non-dominated. Since NaNs are typically caused by errors in the model, we now check for and report an error if a NaN is encountered.
4. **Progress display** – When using `borg.exe` to solve a problem, it will now display a progress bar along with the elapsed time, remaining time, and number of function evaluations. The progress bar only appears when the output is saved to a file (`-f out.txt`).

```
./borg.exe -n 1000000 -v 11 -o 2 -e 0.01,0.01 -f out.txt python dtlz2.py  
[====>] 12% E: 00:00:13, R: 00:01:30, N: 126370/1000000
```

5. **Shorthand for lower/upper bounds** – Previous versions of `borg.exe` required specifying the lower and upper bounds for each decision variable.

```
./borg.exe -n 100000 -v 11 -o 2 -l 0,0,0,0,0,0,0,0,0,0,0 -u 1,1,1,1,1,1,1,1,1,1 -e 0.01,0.01  
python dtlz2.py
```

You can now assign the remaining values using the shorter `...` notation:

```
./borg.exe -n 100000 -v 11 -o 2 -l 0,... -u 1,... -e 0.01,0.01 python dtlz2.py
```

6. **Windows Support** – `borg.exe` can now be used on Windows.
7. **Support for Python, Java, C#, Matlab, R** – Libraries for each of these programming languages has been written, allowing you to optimize your models written in any of these languages.
8. **User manual** – With all these new features, this user manual was written to provide detailed instructions for using the Borg MOEA.

Compiling the Borg MOEA

This chapter describes the steps needed to compile the Borg MOEA on commonly-used operating systems. Please find the section appropriate for your operating system and follow the instructions.

Compiling on Unix/Linux/Cygwin

A makefile is provided to help automatically build the Borg MOEA code and examples on Unix, Linux, Cygwin, or any other Linux-like system. You will need to have a compatible C compiler installed on the system. We recommend using `gcc`, the GNU C compiler. Other compilers known to work with the Borg MOEA include `g++` (GNU C++ compiler), `icc` (Intel C compiler), and `pgcc` (Portland Group C compiler). If you are using a compiler other than `gcc`, you will need to edit the `makefile` and change the following line to reference the appropriate compiler:

```
CC = gcc
```

To build the code and examples, simply **open a terminal, navigate to the Borg MOEA directory, and run the command `make`**. The makefile will automatically build the three executables distributed with the Borg MOEA: `borg.exe`, `dtlz2_serial.exe`, and `dtlz2_advanced.exe`. To test the build, you can then attempt to run either `dtlz2_serial.exe` or `dtlz2_advanced.exe`:

```
-sh-4.1$ make
gcc -O3 -Wl,-R,\. -o dtlz2_serial.exe dtlz2_serial.c borg.c mt19937ar.c -lm
gcc -O3 -Wl,-R,\. -o dtlz2_advanced.exe dtlz2_advanced.c borg.c mt19937ar.c -lm
gcc -O3 -Wl,-R,\. -o borg.exe frontend.c borg.c mt19937ar.c -lm
-sh-4.1$ ./dtlz2_serial.exe
0.287133441382752985 0.49999999725423383 0.499999991596498938 0.50000000052315844
99 0.500000007824245696 0.499999996299798632 0.500000000869960548 0.499999997809
12957 0.5000000005137748449 0.5000000029749478547 0.5000000007685016179 0.8999999414
499789818 0.435891103258643942
0.202802245956361149 0.500000002239317065 0.499999991565778512 0.5000000005220852
417 0.500000001229395585 0.499999997051475187 0.5000000004790669084 0.49999999722
6814652 0.5000000005252678181 0.5000000029758462028 0.5000000007831332915 0.9496870
89198264611 0.313200307487284557
0.843667660682277942 0.49999999971861292 0.499999992739417531 0.50000000056565158
11 0.5000000003684745953 0.499999995117124951 0.499999998480931351 0.5000000002142
245159 0.5000000003726163489 0.5000000030142853658 0.5000000012744572064 0.24310564
0061708816 0.969999818438224803
0.422525969680879743 0.499999999563241926 0.499999991562927293 0.5000000005304657
269 0.5000000007932606239 0.499999996874185726 0.500000001449839915 0.49999999862
667438 0.5000000003919938263 0.5000000029650530253 0.499999988370272597 0.78771691
6200122852 0.616037385174123098
0.69859127177701864 0.499999998787471867 0.499999993070275706 0.5000000003126645
942 0.500000002770568885 0.499999994322044616 0.500000000550638646 0.49999999869
6544046 0.5000000003029364426 0.5000000029372994814 0.5000000004998635283 0.4559610
20694064911 0.889999745846945856
```

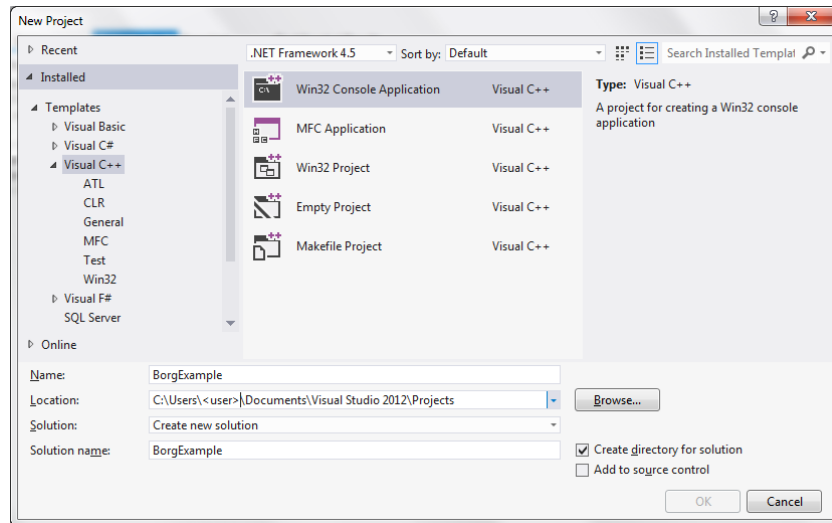
If the `make` program is not available or you need to manually compile the code, the following three commands can be used to build the examples:

```
gcc -O3 -o dtlz2_serial.exe dtlz2_serial.c borg.c mt19937ar.c -lm
gcc -O3 -o dtlz2_advanced.exe dtlz2_advanced.c borg.c mt19937ar.c -lm
gcc -O3 -o borg.exe frontend.c borg.c mt19937ar.c -lm
```

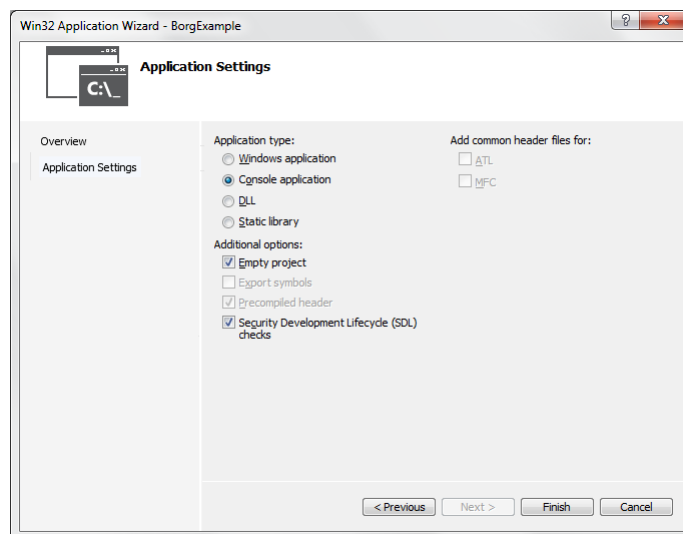
Compiling on Windows (Visual Studio)

The Borg MOEA can be compiled on Windows using Visual Studio 2005 or later versions. The screenshots in this section are from Visual Studio 2012. The steps for setting up a project may differ slightly between versions.

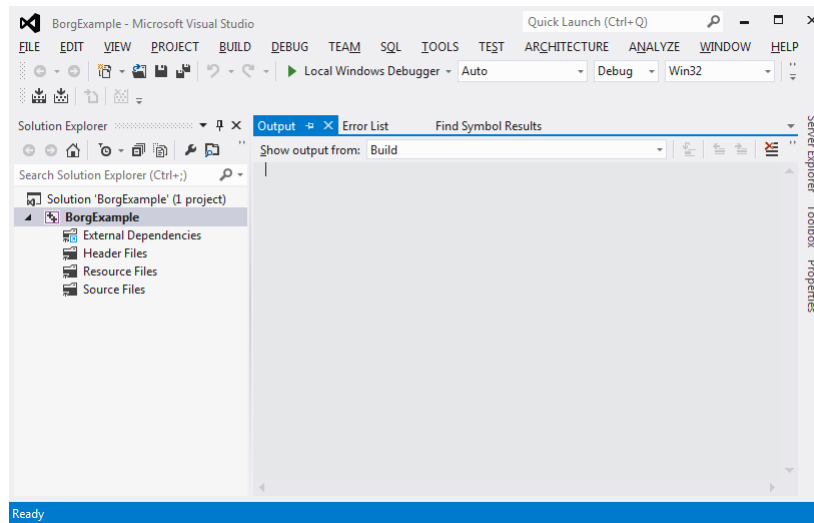
After launching Visual Studio, **create a New Project**. In the window that appears, select the **Visual C++** template and ensure **Win32 Console Application** is selected. You can optionally set the name and location for this project on your computer.



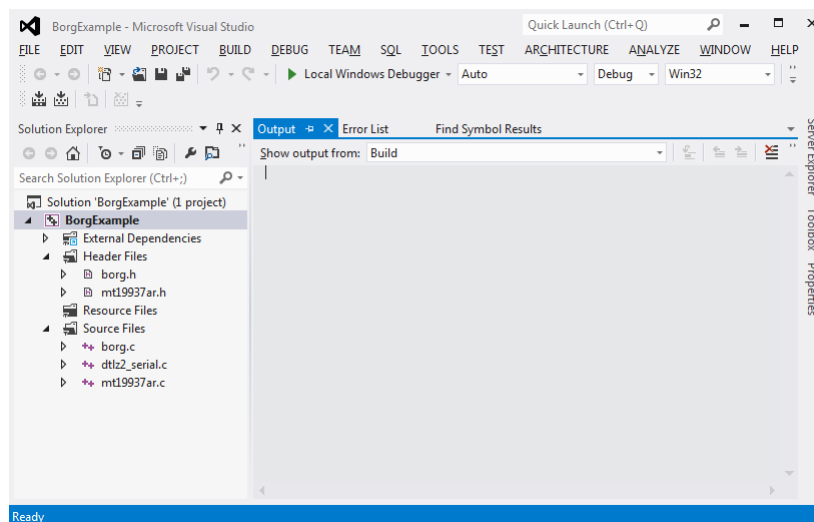
After clicking OK, check the **Empty Project** box as shown in the screenshot below.



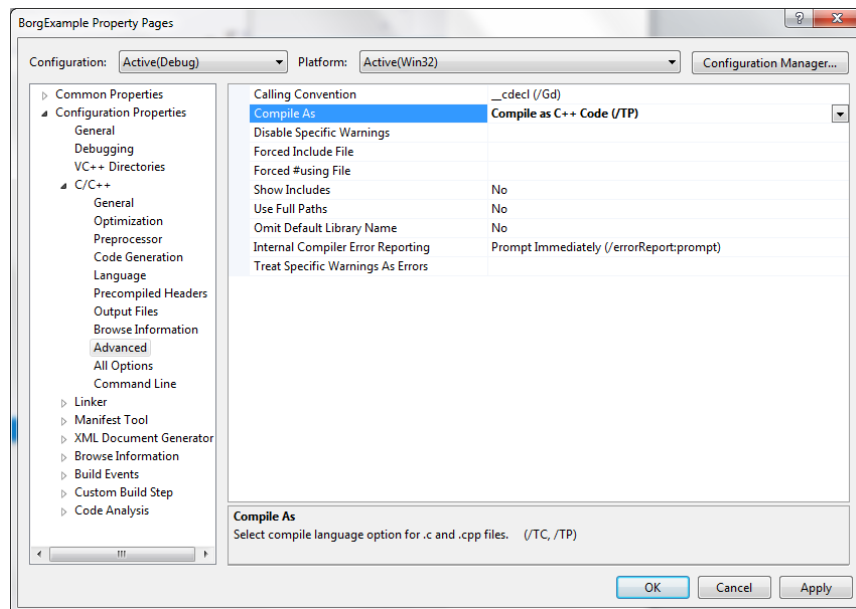
Click Finish. The result is an empty Visual Studio project, as shown below.



We will now add the C files for this example. **Right-click the project and select Add > Existing Item.** This will open a file browser window. **Navigate to the Borg MOEA directory and select the following files: `borg.c`, `borg.h`, `mt19937ar.c`, `mt19937ar.h`, and `dtlz2_serial.c`.** After selecting these five files, **click Add.** The files should appear in your project as shown below.



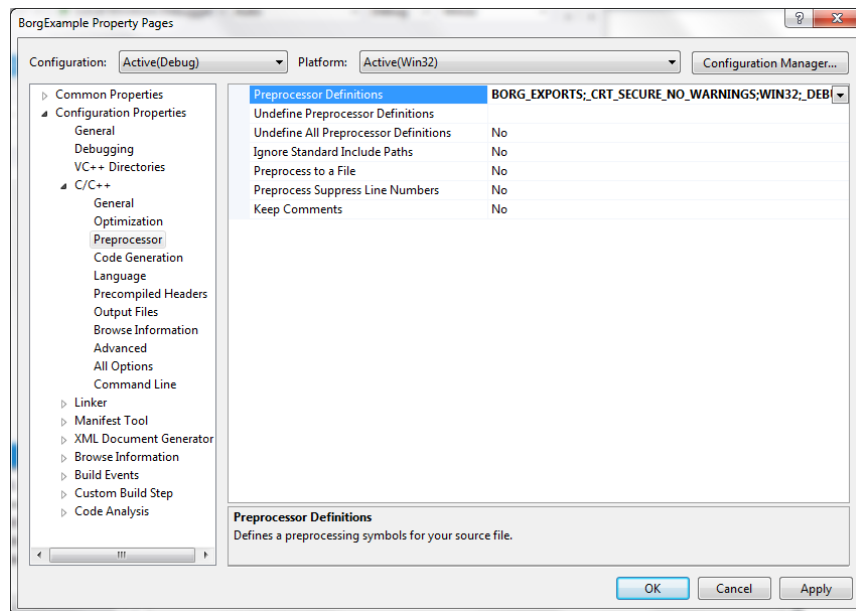
The last step is to configure the compiler settings. Visual Studio's C compilers only supports an older version of the C standard (C89). In order to compile the Borg MOEA, we must force Visual Studio to use the C++ compiler. To do so, **right-click on the project and select Properties.** In the window that appears, find and **click the C/C++ > Advanced** page on the left. Finally, **select the Compile As option, click the down arrow, and select the "Compile as C++ Code (/TP)" option.**



At this point, you should be able to compile the code. To compile, **right-click the project and select Build**. If the build was successful, you should see the text below appear on the screen:

```
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

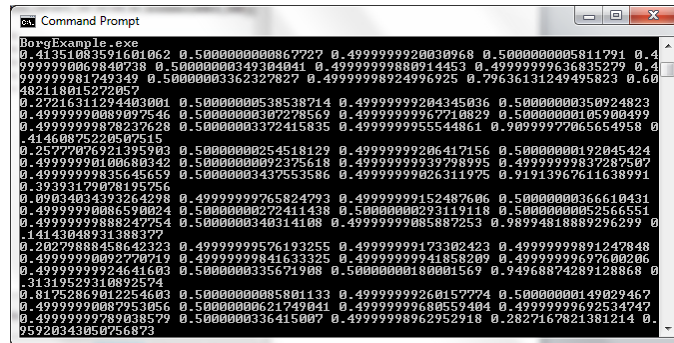
When you compile the code above, it should succeed but you may see a number of warning messages. If you want to get rid of the warning messages, add the `BORG_EXPORTS` and `_CRT_SECURE_NO_WARNINGS` preprocessor definitions in the project properties, as shown below.



You can now go and run the program. If you are unsure where to find the program, look at the output text for a line similar to:

```
1> BorgExample.vcxproj -> C:\Users\<user>\Documents\Visual Studio
2012\Projects\BorgExample\Debug\BorgExample.exe
```


Open a new command prompt window and navigate to the folder containing the executable. We strongly recommend running the program within a command prompt window as you will be able to see the output from the program. If you just double-click the program file, the command prompt window will disappear when the program finishes running. Opening a new command prompt window will retain the program's output. The result should look similar to the following:



```
Command Prompt
BorgExample.exe
0.41351083591601062 0.5000000000867727 0.49999999920030968 0.50000000005811791 0.4
999999990069840738 0.500000000349304041 0.49999999880914453 0.499999999636835279 0.4
999999981749349 0.500000003362327827 0.49999998924996925 0.79636131249495823 0.60
482118015272057
0.27216311294403001 0.500000000538538714 0.499999999204345036 0.500000000350924023
0.499999990089097546 0.500000000307278569 0.49999999967710829 0.500000000105900499
0.49999999878237628 0.500000003372415835 0.4999999955544861 0.909999977065654958 0
.41460875220507515
0.25777076921295903 0.50000000254518129 0.499999999206417156 0.500000000192045424
0.499999990100680342 0.50000000092375618 0.49999999939798995 0.49999999837287507
0.49999999835645659 0.500000003437553586 0.49999999026311975 0.91913967611638991
0.39393179078195756
0.09034834393264298 0.49999999765824793 0.49999999152407606 0.500000000366610431
0.499999990086590024 0.50000000272411438 0.500000000293119118 0.50000000052565551
0.49999999888247754 0.50000000340314108 0.49999999085887253 0.98994818889296299 0
.14143048931388377
0.20279888458642323 0.499999999576193255 0.49999999173302423 0.49999999891247848
0.49999999007270719 0.49999999841633325 0.49999999941858209 0.49999999697600206
0.49999999924641603 0.50000000335671908 0.500000000180001569 0.94968874289128868 0
.131319529310892574
0.01752869012254603 0.50000000085801133 0.499999999260157774 0.500000000149029467
0.499999990087953056 0.500000000621749041 0.49999999680559404 0.49999999692534747
0.49999999789038579 0.50000000336415007 0.499999998962952918 0.2827167821381214 0
.95920343050756873
```

A Simple Example

This chapter will introduce the basic concepts of using the Borg MOEA using a simple toy problem, the 2-objective DTLZ2 problem. The DTLZ2 problem is a popular starting algorithm for testing MOEAs because it is conceptually easy to understand, the Pareto-optimal solutions are known analytically, and it is scalable to any number of objectives. The original DTLZ2 problem for M objectives is defined by the following set of equations:

Minimize:

$$f_1(x) = (1 + g(x_M)) \cos\left(\frac{x_1\pi}{2}\right) \cos\left(\frac{x_2\pi}{2}\right) \cdots \cos\left(\frac{x_{M-2}\pi}{2}\right) \cos\left(\frac{x_{M-1}\pi}{2}\right)$$

$$f_2(x) = (1 + g(x_M)) \cos\left(\frac{x_1\pi}{2}\right) \cos\left(\frac{x_2\pi}{2}\right) \cdots \cos\left(\frac{x_{M-2}\pi}{2}\right) \sin\left(\frac{x_{M-1}\pi}{2}\right)$$

$$f_3(x) = (1 + g(x_M)) \cos\left(\frac{x_1\pi}{2}\right) \cos\left(\frac{x_2\pi}{2}\right) \cdots \sin\left(\frac{x_{M-2}\pi}{2}\right)$$

\vdots

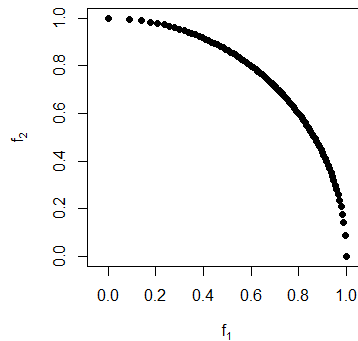
$$f_{M-1}(x) = (1 + g(x_M)) \cos\left(\frac{x_1\pi}{2}\right) \sin\left(\frac{x_2\pi}{2}\right)$$

$$f_M(x) = (1 + g(x_M)) \sin\left(\frac{x_1\pi}{2}\right)$$

Subject to $0 \leq x_i \leq 1 \forall i = 1, 2, \dots, n$

Where $g(x_M) = \sum_{x_i \in x_M} (x_i - 0.5)^2$

The Pareto-optimal solutions correspond to $x_i = 0.5$ for all $x_i \in x_M$. The Pareto front is the first quadrant of a unit sphere centered at the origin, or $\sum_{i=1}^M f_i^2 = 1$. For the 2-objective case that we will be solving, the Pareto front is depicted in the figure below.



The first step for setting up a new optimization problem for use with the Borg MOEA is to define the objective function. In C, the objective function is any function with the form:

```
void functionName(double* vars, double* objs, double* constrs)
```

This function has three arguments, all of which are arrays of double values initialized to the appropriate length. Every candidate solution generated by the Borg MOEA will be evaluated using this function. The decision variables from the candidate solution will be passed into the function in the `vars` argument, and the objective and constraint values are returned using the `objs` and `constrs` arguments, respectively. For example, we can program the DTLZ2 function as follows:

```
int nvars = 11;
int nobjs = 2;

void dtlz2(double* vars, double* objs, double* constrs) {
    int i;
    int j;
    int k = nvars - nobjs + 1;
    double g = 0.0;

    for (i=nvars-k; i<nvars; i++) {
        g += pow(vars[i] - 0.5, 2.0);
    }

    for (i=0; i<nobjs; i++) {
        objs[i] = 1.0 + g;

        for (j=0; j<nobjs-i-1; j++) {
            objs[i] *= cos(0.5*PI*vars[j]);
        }

        if (i != 0) {
            objs[i] *= sin(0.5*PI*vars[nobjs-i-1]);
        }
    }
}
```

Since the DTLZ2 problem can have any number of decision variables and objectives, we define the `nvars` and `nobjs` constants to configure the problem. In this example, we will be solving the 2-objective DTLZ2 problem with 11 decision variables. The function itself is a direct translation of the mathematical equations from above into C code. The key point here, though, is to see how we read the decision variables from the `vars` array and store the computed objective values back into the `objs` array. All of the arrays are initialized to the correct lengths.

All objectives passed to the Borg MOEA will be minimized. It is important to remember that the Borg MOEA only minimizes objectives. This design choice was made for performance reasons: adding additional cases to handle minimized and maximized objectives would decrease performance. Instead, if you have any maximized objectives, you need to negate the value of the objective. Minimizing the negated value is equivalent to maximizing the original value. Also be aware that the Pareto solutions returned at the end of optimization will still have the negated values, so you will want to convert the values back to their original form at the end.

The DTLZ2 problem is unconstrained, so we do not need to modify the `constrs` array. If the problem did have constraints, then we would need to compute and assign the constraint values similar to how we computed and assigned the objective values. **All satisfied constraints must have a value of 0. Any non-zero value is considered a constraint violation.** It doesn't matter if the constraint value is positive or negative, any non-zero value is interpreted as a constraint violations. Also note that the Borg MOEA considers the magnitude of constraint violations. For example, a constraint value of 10 is considered to be worse than a constraint value of 5. Again, the sign does not matter, only the absolute magnitude of the constraint value.

Once the objective function is defined, we can then optimize the function using the Borg MOEA:

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

    BORG_Problem problem = BORG_Problem_create(nvars, nobjs, 0, dtlz2);

    for (i=0; i<nvars; i++) {
        BORG_Problem_set_bounds(problem, i, 0.0, 1.0);
    }

    for (i=0; i<nobjs; i++) {
        BORG_Problem_set_epsilon(problem, i, 0.01);
    }

    BORG_Archive result = BORG_Algorithm_run(problem, 1000000);
    BORG_Archive_print(result, stdout);

    BORG_Archive_destroy(result);
    BORG_Problem_destroy(problem);
    return EXIT_SUCCESS;
}
```

In the above code, we first define the problem we are solving. We first call `BORG_Problem_create` with the number of decision variables, `nvars`, the number of objectives, `nobjs`, the number of constraints, 0 (since this is an unconstrained problem), and the name of the objective function, `dtlz2`. Next, we loop over each decision variable and assign their lower and upper bounds to be 0 and 1, respectively. Finally, we assign the epsilon values of 0.01 for each objective.

Starting with this version, 1.8, you are now required to specify the epsilon values. The epsilon values are very important to the operation of the Borg MOEA. The epsilons specify the resolution for each objective, which in effect controls the number of Pareto optimal solutions and how far apart they are spaced. Smaller epsilons result in fine-grained sets, whereas larger epsilons produce coarser sets. **Setting appropriate epsilons for your optimization problem is important, as too small or too large epsilons can affect the performance of the algorithm.** Although it may initially take some trial and error to find appropriate epsilons, they provide significant benefits. First, they allow us to prove that the Borg MOEA is convergent (both in proximity and diversity to the true Pareto-optimal set). Second, they allow the algorithm to efficiently scale to higher objectives without experiencing exponential growth in the number of solutions required to represent the set. Finally, we use epsilons to track the progress of the algorithm, requiring that the algorithm improves the set by

at least the minimum resolution. If the Borg MOEA is unable to do so, it then becomes more aggressive and starts to self-adapt to become more effective at generating new, Pareto-optimal solutions.

If you are working on a real-world problem, try setting the epsilons to the minimum change of interest. For example, if you are optimizing a water supply system worth tens of millions of dollars, you are probably not interested in differences of one or two dollars. Setting the epsilon to, for example, ten thousand dollars will hide these negligible differences and only show significant changes in cost savings.

Returning to the code example, after creating the problem we can now use the Borg MOEA to optimize the problem. We next call `BORG_Algorithm_run` providing the problem we just created and the maximum number of objective function evaluations (NFE) that we will allow the algorithm to run.

`BORG_Algorithm_run` will use the default parameter settings for the Borg MOEA. It is possible, with some additional work, to customize these parameters (see `dtlz2_advanced.c`).

When the Borg MOEA finishes, it will return the approximate Pareto-optimal solutions discovered by the algorithm. We can then call `BORG_Archive_print` to print the solutions to the screen or to a file. Each line in the output corresponds to a solution in the approximate Pareto-optimal set. Each value on a line corresponds to the decision variables, objectives, and constraints, in this order. Finally, it is a good practice to always free any objects you create (using one of the `*_create` methods) by using the corresponding `*_destroy` methods.

That's it! See `dtlz2_serial.c` for the full source code for this example. To compile and run this example, follow the instructions from the preceding chapter, *Compiling the Borg MOEA*. For a more advanced example where we set parameter values and use checkpoints, see `dtlz2_advanced.c`.

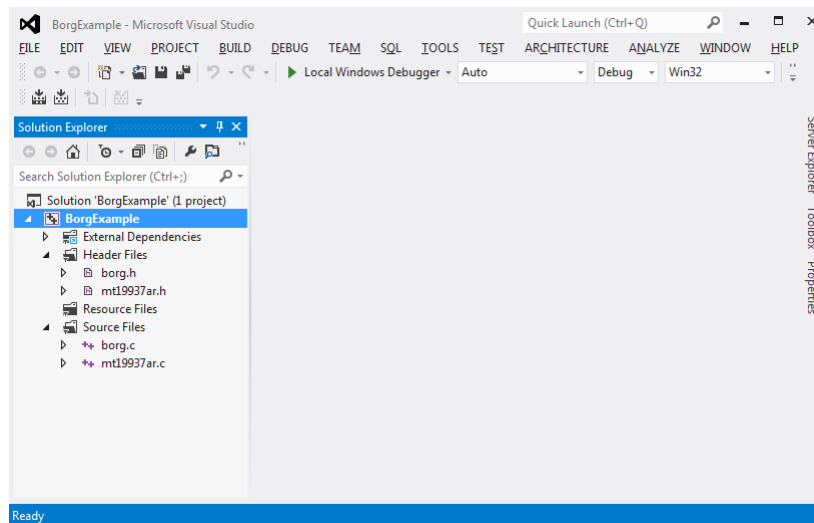
Supported Programming Languages

The Borg MOEA includes libraries for Python, Java, C#, Matlab, and the R statistical language. With these libraries, you can optimize models written in any of these languages using the Borg MOEA. Each library can harness the underlying efficiency and power of the natively-compiled Borg MOEA, but with the flexibility of your favored programming language.

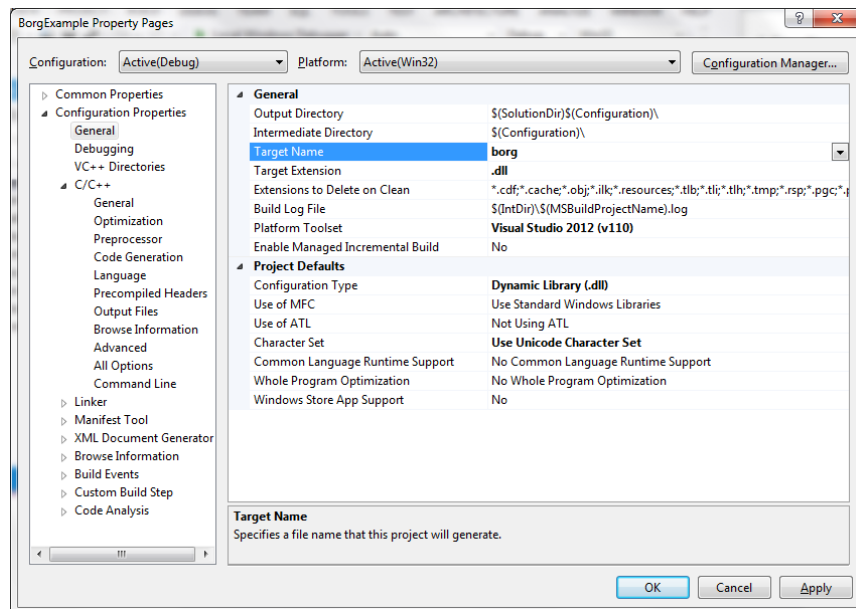
Initial Setup

Regardless of the programming language, you must first compile the Borg MOEA into a shared library. On Windows, the shared library is known as a DLL (dynamic-link library). On Unix/Linux systems, they are called shared objects.

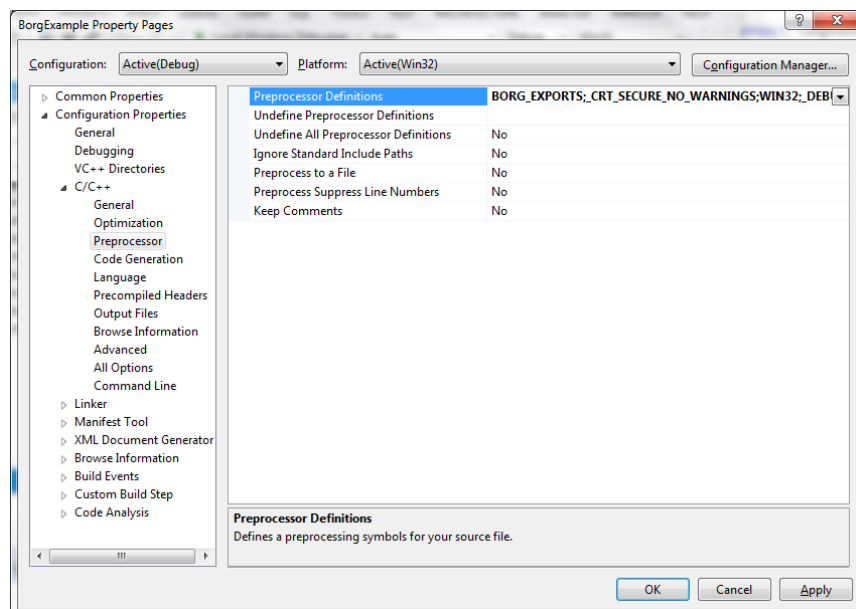
On Windows, you create the Borg DLL (named `borg.dll`) by **creating a new, empty Visual Studio project. Add the following four files to the project: `borg.c`, `borg.h`, `mt19937ar.c`, and `mt19937ar.h`.**



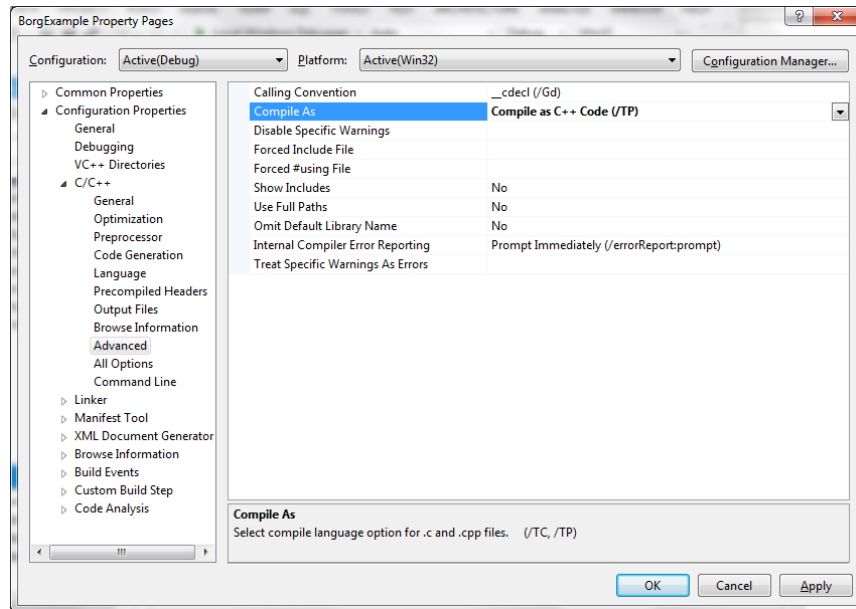
Next, you will need to edit the project properties. **On the general page, change Target Name to `borg`, Target Extension to `.dll`, and Configuration Type to “Dynamic Library (.dll)”.**



Switch to the C/C++ > Preprocessor page and add the **BORG_EXPORTS** and **_CRT_SECURE_NO_WARNINGS** preprocessor definitions, as shown below.



And finally, switch to the C/C++ > Advanced page and change the Compile As option to “Compile as C++ Code (/TP)”.



Once these properties are set, you may **click OK to save the changes and build the DLL**. This will generate the `borg.dll` file. If you are unsure where the file is saved, look at the output text for the path.

On Unix/Linux systems, creating the shared object is much easier. Simply run the following command in a terminal from the Borg MOEA folder:

```
gcc -shared -fPIC -O3 -o libborg.so borg.c mt19937ar.c -lm
```

Once you have generated the `borg.dll` or `libborg.so` file, you may continue to the section below appropriate for your programming language.

Python

The Borg Python library includes two files: `borg.py` and `test.py`. We will focus on `test.py`, which includes an example problem. The contents of `test.py` are shown below for your convenience.

```
from sys import *
from math import *
from borg import *

nvars = 11
nobjs = 2
k = nvars - nobjs + 1

def DTLZ2(*vars):
    g = 0

    for i in range(nvars-k, nvars):
        g = g + (vars[i] - 0.5)**2

    objs = [1.0 + g]*nobjs

    for i in range(nobjs):
        for j in range(nobjs-i-1):
            objs[i] = objs[i] * cos(0.5 * pi * vars[j])
            if i != 0:
                objs[i] = objs[i] * sin(0.5 * pi * vars[nobjs-i-1])

    return objs

borg = Borg(nvars, nobjs, 0, DTLZ2)
borg.setBounds(*[[0, 1]]*nvars)
borg.setEpsilons(*[0.01]*nobjs)

result = borg.solve({"maxEvaluations":1000000})

for solution in result:
    print solution.getObjectives()
```


The objective function is DTLZ2. This is a function with a single argument, `vars`, storing the decision variables. The result of the function, `objs`, is a list of the objective values. In this example, the problem is unconstrained. For a constrained problem, you would return a tuple storing the objective and constraint values, such as:

```
return (objs, constrs)
```

To optimize this function, we create a new instance of the Borg class and specify the lower and upper bounds for each decision variable as well as the epsilon values:

```
borg = Borg(nvars, nobjs, 0, DTLZ2)
borg.setBounds(*[[0, 1]]*nvars)
borg.setEpsilons(*[0.01]*nobjs)
```

In this case, we are defining all decision variables to have a lower bound of 0 and an upper bound of 1, and each objective with an epsilon of 0.01. We could have also specified the bounds and epsilons for each decision variable or objective separately, such as:

```
borg.setBounds([0, 1], [-1, 1], [0.25, 0.75], [2000, 5000])
borg.setEpsilons(0.01, 10.0, 2.5)
```

Once these parameters are set, we are ready to optimize the function. In Python, we invoke the optimization routine with the following command:

```
result = borg.solve({"maxEvaluations":1000000})
```

and print the results to the output:

```
for solution in result:
    print solution.getObjectives()
```

To run this example, first copy `borg.dll` or `libborg.so` into the folder containing `borg.py` and `test.py`. Then, from a command prompt window or terminal, run the command

```
python test.py
```

If all goes well, you will see a bunch of numbers print to the window showing the objective values for each solution.

There are many other features supported by the Python library. If you are interested in learning more about these features, see the code and documentation in `borg.py`.

Java

First, you will need to compile the Borg DLLs for your target operating system. For your convenience, the script `build-native.sh` is provided that will compile the Borg DLLs for your current system. This script will place the generated DLL within the `native/` folder.

Next, to compile the Java library, you will need the Java Development Kit (JDK 7 or 8) and Apache Ant, available for download from <http://ant.apache.org/>. You will also need to download the MOEA Framework compiled binaries from <http://moeaframework.org/> and copy the contents of the MOEAFramework-X.X/lib/ folder into the Borg-1.8/plugins/Java/lib/ folder. Once these dependencies are installed, open a new command prompt or terminal window, navigate to the Java folder, and run the command ant. This will create the borg-1.8.jar file inside the dist/ directory.

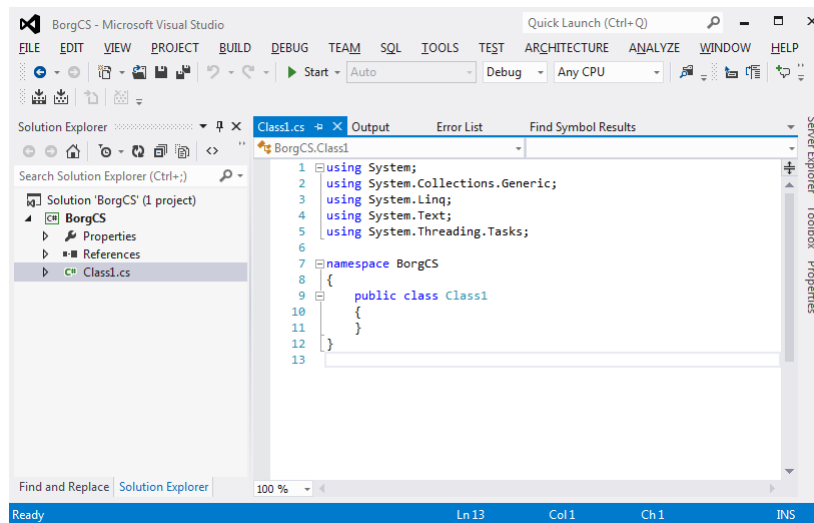
To use the compiled JAR file, you need to import the two libraries, jna-4.1.0.jar and borg-1.8.jar, within your Java project. See the src\Test.java file within the Java folder for example usage.

If you see an error indicating the JNA library was unable to locate the native Borg library for your system, you may need to compile the DLL separately (see the first section in this chapter) and place it somewhere on the Java library path (e.g., java -Djava.library.path=/path/to/borg.dll ...).

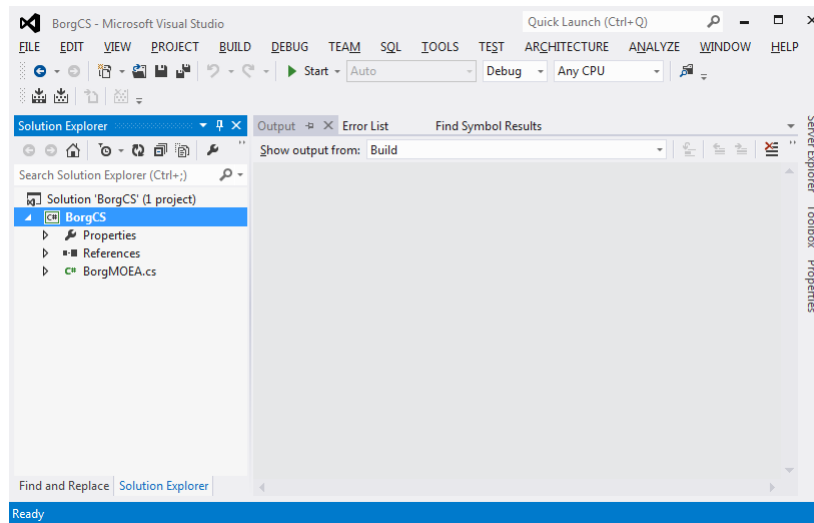
C# (Visual Studio)

The Borg C# library works with both Visual Studio and Mono, an open source implementation of the .NET framework. This section covers the use of Visual Studio. The subsequent section discusses Mono.

The first step is to compile the Borg C# library. First, **start Visual Studio and create a new project. Select the Visual C# template and select Class Library. Change the name to BorgCS. Click Ok.**

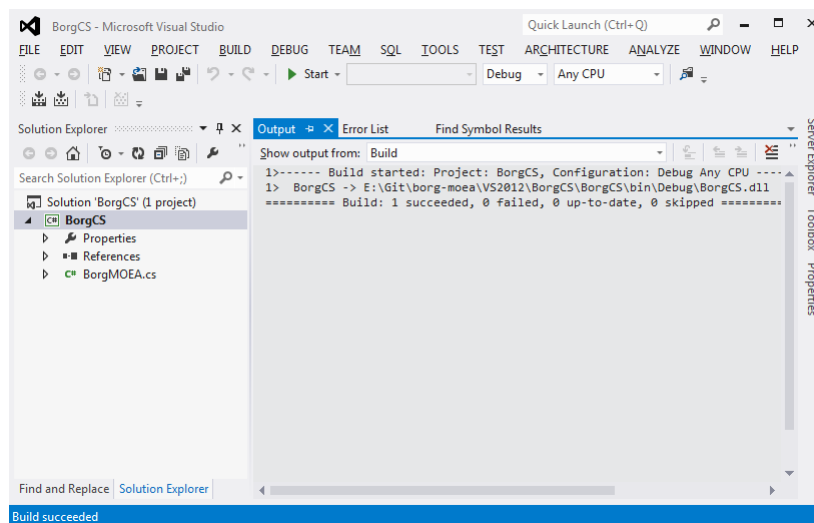


Visual Studio creates a new project containing a boilerplate file called `Class1.cs`. We want to remove this file and add our own. First, **right-click `Class1.cs` and select Delete**. This removes the unwanted `Class1.cs` file. Next, **add the file `BorgMOEA.cs` from the C# folder by right-clicking on the project folder and selecting Add > Existing Item**. Once you have located and selected `BorgMOEA.cs` within the file browser, click Add.

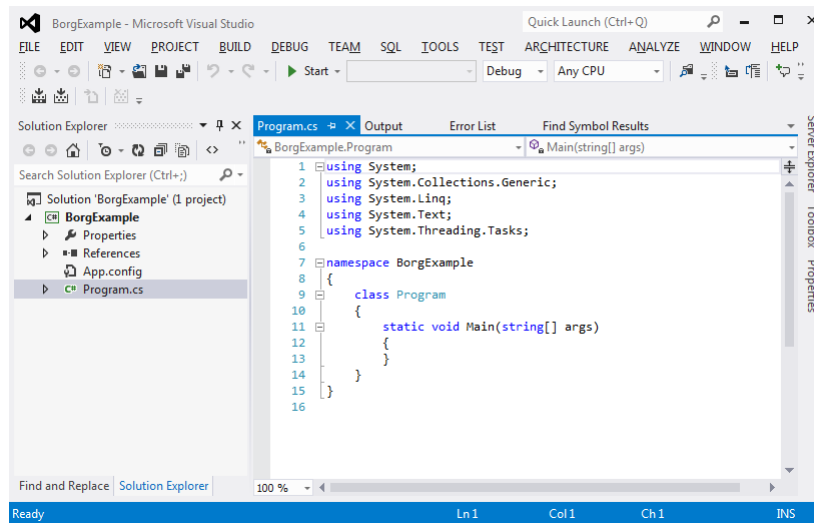


Right-click the project folder again and select Properties. In the window that appears, **click the Build tab and check XML documentation file.** This will enable helpful tooltips with documentation when you are using the Borg C# library within Visual Studio. **Click the save icon.**

Finally, we can build the C# library. **Right-click the project folder and select Build.** If successful, the end of the output will show “Build: 1 succeeded, 0 failed, ...”. This creates the file `BorgCS.dll`. You will need this file along with `borg.dll` whenever you are using the Borg C# wrapper.

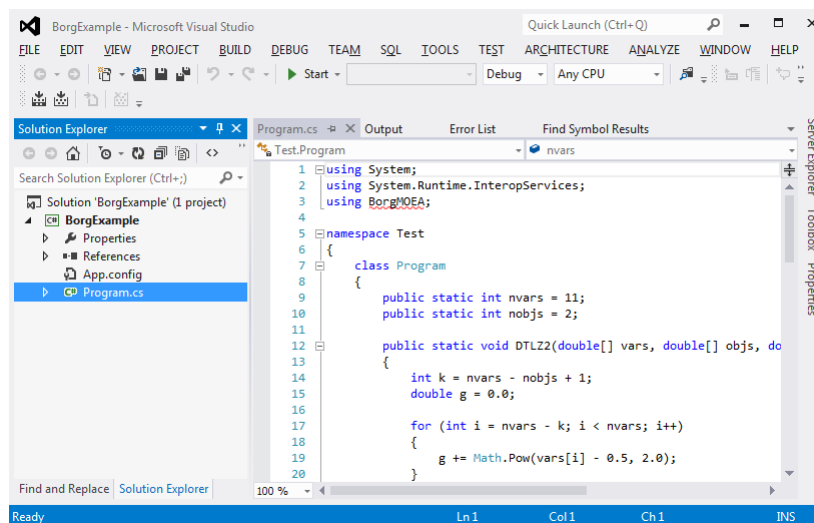


Now we will demonstrate using the Borg C# library to optimize the DTLZ2 example problem. The example code is in `Program.cs`, which you can find in the C# folder. To start, **open a new Visual Studio session and create a new project. Select Console Application from the Visual C# template. Change the name to BorgExample and click Ok.**

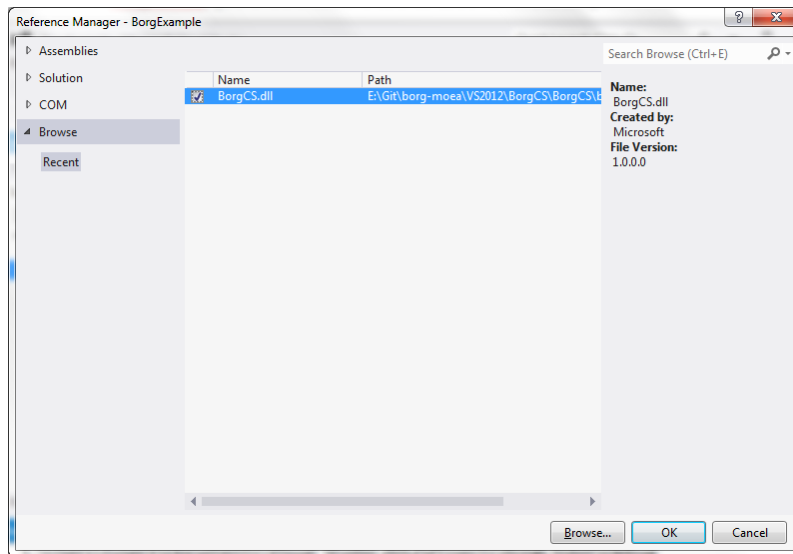


Like before, a new Visual Studio project contains a boilerplate starting class, in this case it's called `Program.cs`. Delete this file by **right-clicking `Program.cs` and selecting Delete**.

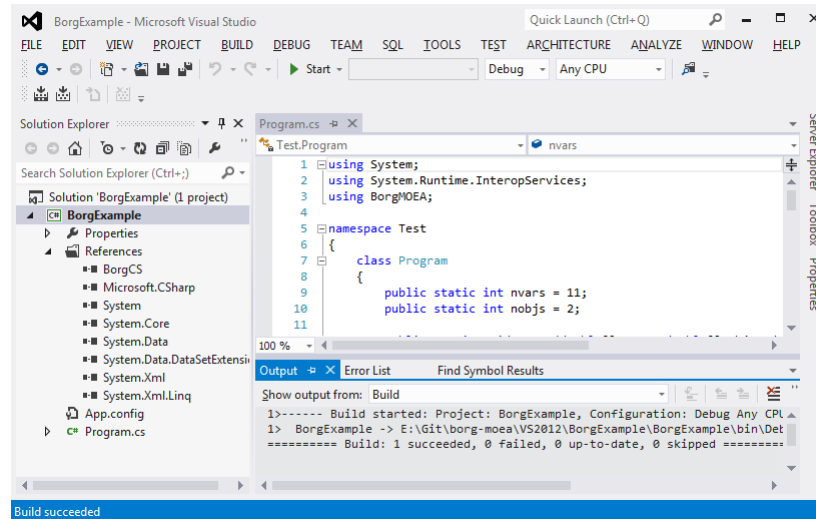
Next, add our example code, `Program.cs`, stored in the C# folder. You can add this file by **right-clicking the project folder and selecting Add > Existing Item**. Navigate to the **Borg C#** folder, select **`Program.cs`**, and click **Add**. If you double-click `Program.cs`, you will see the source code for this example, as shown below.



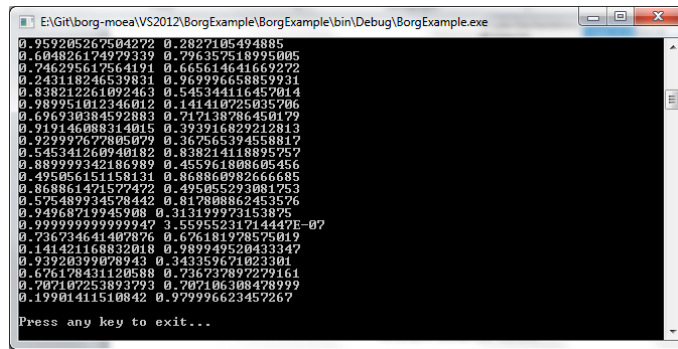
Good. Now we need to add the Borg C# library, `BorgCS.dll`, that we just built. **Right-click References and select Add Reference**. Click the **Browse** button at the lower-right corner of the window and navigate to and select the **`BorgCS.dll`** file. Click **Add**.



Make sure the BorgCS.dll row is checked and **click OK**. At this point, we can compile the example. **Right-click the project folder and select Build**. If successful, the output will show “Build: 1 succeeded, 0 failed, ...”.



At this point, you have successfully compiled BorgExample.exe. In order to run the program, BorgCS.dll and borg.dll must be located alongside the executable. If you navigate to the folder where BorgExample.exe was created, you should notice that borg.dll is missing. **Copy and paste borg.dll into this folder**. Finally, double-click BorgExample.exe to run the program. You should see a command prompt window that will display the objective values after about 20-30 seconds.



C# (Mono)

Mono is an open source implementation of Microsoft's .NET framework used by Visual Studio. Whereas Visual Studio only compiles programs that can run on Windows, Mono allows .NET programs to run on Linux and Mac OS X. Open a new terminal and navigate to the Borg C# folder. The first step is to compile the Borg C# library, called `BorgCS.dll`. To do so, run the following command:

```
mcs -out:BorgCS.dll -target:library BorgMOEA.cs
```

Second, compile the example DTLZ2 problem in `Program.cs` with the following command:

```
mcs -reference:BorgCS.dll Program.cs
```

This will generate the executable, `Program.exe`. Next, copy `borg.dll` to this folder. All three files (`borg.dll`, `BorgCS.dll`, and `Program.exe`) must be contained in the same directory. Finally, you can run the example with the following command:

```
mono Program.exe
```

Matlab

The Borg Matlab library allows you to optimize functions in Matlab using the Borg MOEA. The Matlab function can be one of two forms:

```
function [ objs ] = FunctionName( vars )
function [ objs, constrs ] = FunctionName ( vars )
```

In the first case, the function takes a single argument, `vars`, as an array of the decision variables and returns an array of the objective values, `objs`. In Matlab, an array is just a matrix with one row (e.g., `zeros(1, nobjs)`). If the function includes constraints, the function must return two arrays, `objs` and `constrs`. See the `DTLZ2.m` function in the Matlab folder for an example.

The Borg Matlab library uses a binary MEX file. This MEX file is similar to a DLL, except it is compiled by Matlab's MEX compiler. If you look in the Matlab folder provided in the Borg MOEA distribution, you will see a file called `nativeborg.cpp`. We will first compile `nativeborg.cpp` into the binary MEX file. If you are using Windows, you will need to copy `borg.dll` and `borg.lib` into the Matlab folder. Note that you

need both the `.dll` and `.lib` files on Windows. If you are using Unix or Linux, copy `libborg.so` into the Matlab folder.

To compile the library, first copy `borg.h` into the Matlab directory. Next, start Matlab. Then, if you are using Windows, run the following command within Matlab:

```
mex nativeborg.cpp Borg.lib
```

If you are using Unix/Linux, run the following command:

```
mex LDFLAGS="\$LDFLAGS -Wl,-rpath,\'\' nativeborg.cpp libborg.so
```

This will generate a file like `nativeborg.mexw64` or `nativeborg.mexa64` (possibly ending in 32) depending on your system. Once compiled, you can then run the following command from within Matlab:

```
[vars, objs] = borg(11, 2, 0, @DTLZ2, 10000, [0.01, 0.01])
```

The result of running this command will be two matrices in Matlab, one storing the decision variables and the other storing the objective values. Each row corresponds to a Pareto optimal solution. For additional help, run `help borg` within Matlab.

Note: in order for this to work, you need to have `borg.m`, `DTLZ2.m`, `borg.dll`, and the generated MEX file (e.g., `nativeborg.mexw64`) all in the same folder. Typically, if you start Matlab from the folder containing these four files it will work. If not, you may need to change Matlab's working directory.

R

The Borg R library is fairly straightforward to setup. Copy the `borg.dll` or `libborg.so` file into the R folder contained in the Borg MOEA distribution. Start R, and run `source("test.R")`. After a short amount of time, you will see the Pareto optimal solutions printed to the screen. For details on using the Borg R library, see `test.R` and `DTLZ2.R`.

On Windows, you may need to compile `borg.dll` using the R Tools toolchain, available for download from <http://cran.at.r-project.org/bin/windows/Rtools/>. After installing R Tools, run the following two commands:

```
set PATH=C:\Program Files\Rtools\bin;C:\Program Files\Rtools\gcc-4.6.3\bin
gcc -shared -O3 -o borg.dll borg.c mt19937ar.c -lm
```

You may need to adjust the `PATH` to the correct folder names given your installation of R Tools.

Speed Comparison

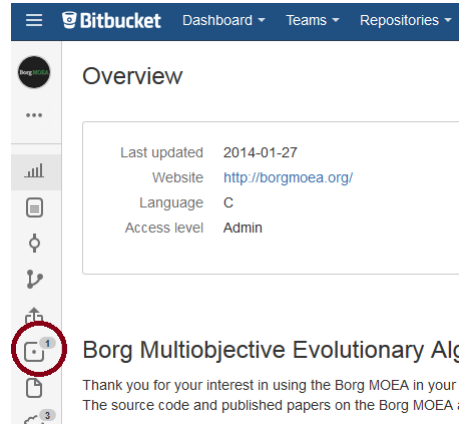
While the availability of these libraries for various programming languages allows you to optimize functions written within that language, it is important to understand the potential downsides. The primary downside is reduced runtime performance. The following table shows the walltime required for one million function evaluations of the DTLZ2 test problem.

Language	Time	Slowdown
C	6.9 sec	
Python	23.9 sec	3.4x
Java	15.9 sec	2.3x
C#	11.3 sec	1.6x
Matlab	49.4 sec	7.2x
R	244 sec	35.4x

You will note that the baseline C program is roughly 6.9 seconds. Most of the programming languages run between 2-7 times slower than the baseline C program. R is the worst with over 35x slowdown. If performance does become a concern, you may use this table to roughly see the kind of speedup you can expect by switching to C. Note that you will probably see even more speedup due to re-writing your problem in C.

Troubleshooting

If you experience any trouble compiling or running the Borg MOEA, feel free to open an issue on our BitBucket page at <https://bitbucket.org/dmh309/serial-borg-moea>. Click the issues tab, highlighted below, to see existing issues or create new issues.



You can also click the Wiki tab, the icon that looks like a piece of paper, to view articles discussing the use of the Borg MOEA. The remainder of this chapter includes answers to some common questions.

Missing Symbols (fabs, sqrt, log, or floor)

When compiling the Borg MOEA, if you see an error message about missing symbols like `fabs`, `sqrt`, `log`, or `floor`, add the `-lm` flag at the end of the command, such as:

```
gcc -o MyProblem.exe problem.c borg.c mt19937ar.c -lm
```

Unable to Compile `borg.exe` on Windows

Unfortunately, due to differences in how Windows and Unix/Linux handle sockets and processes, some functionality is not supported on Windows. Windows users can compile and run Borg directly, such as with the `dt1z2_serial.c` example, but they will not be able to compile the command line interface, `borg.exe`.

`java.lang.UnsatisfiedLinkError: /lib64/libc.so.6: version 'GLIBC_2.14' not found`

If you see an error message like this when using the Java Borg library, then your system uses a different version for the C library than that supported by JNA. To find out what version your system uses, run the command `/lib64/libc.so.6`. You will then need to download an older version of the Java JNA library from <http://mvnrepository.com/artifact/net.java.dev.jna/jna>. The known mapping of JNA versions to GLIBC versions includes:

- 4.1.0 – GLIBC_2.14
- 3.5.2 – GLIBC_2.12

Representing Discrete Decision Variables

While the Borg MOEA only supports real-value decision variables, it is possible to encode certain discrete variables to work with the Borg MOEA.

- Integers – For an integer between $[a, b]$, encode as a real-value on the range $[a, b+0.999]$. Then truncate the real-value to get the integer:

```
int value = (int)floor(x);
```

- Set of Options – Encode as an integer that specifies the index of the option:

```
const char* colors[] = {"red", "blue", "green"};  
int index = (int)floor(x);  
const char* value = colors[index];
```

- Binary – Encode as a real-value on the range $[0, 1]$ and set to TRUE if $x > 0.5$:

```
bool value = x > 0.5;
```

Additional Reading Materials

This chapter provides references to related publications involving the Borg MOEA for interested readers.

Hadka, D. and P. Reed. *“Borg: An Auto-Adaptive Many-Objective Evolutionary Computing Framework.”* Evolutionary Computation, 21(2):231-259, 2013.

This peer-reviewed journal article introduces the design of the Borg MOEA and compares the performance of the Borg MOEA against six state-of-the-art MOEAs on 33 test problem instances. This preliminary analysis shows the Borg MOEA matches or exceeds the search quality of the competing algorithms on the majority of test problems.

Hadka, D. and P. Reed. *“Diagnostic Assessment of Search Controls and Failure Modes in Many-Objective Evolutionary Optimization.”* Evolutionary Computation, 20(3):423-452, 2012.

Providing a more comprehensive comparison of the Borg MOEA against 8 state-of-the-art MOEAs, including the popular MOEA/D algorithm, this peer-reviewed journal article assessed the performance of the tested algorithms using a comprehensive and rigorous testing methodology. Many studies use preset parameter settings when testing algorithms, but this study demonstrates that an algorithm’s “sweet-spot”, or the parameter settings yielding the best performance, change across problems. Due to the Borg MOEA’s auto-adaptive and multi-operator features, it exhibits a large sweet-spot indicating its performance is less dependent on parameters than other algorithms.

Reed, P., et al. *“Evolutionary Multiobjective Optimization in Water Resources: The Past, Present, and Future.”* (Editor Invited Submission to the 35th Anniversary Special Issue), Advances in Water Resources, 51:438-456, 2013.

This peer-reviewed article extends the prior analyses of the Borg MOEA to include three real-world applications. Real-world applications often exhibit problem characteristics not present in the simple test problems commonly used to assess the performance of MOEAs. The applications include calibrating a rainfall-runoff model, devising a strategy for long-term groundwater monitoring for pollutants, and planning a risk-based water supply portfolio. In all three problems, the Borg MOEA outperformed the competition, expanding the prior results to real applications.

Woodruff, M. et al. *“Auto-Adaptive Search Capabilities of the New Borg MOEA: A Detailed Comparison on Product Family Design Problems.”* 12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Indianapolis, Indiana, 17 Sept. 2012.

The previous studies hint that the multi-operator and auto-adaptive features of the Borg MOEA help reduce the dependency of performance on the algorithm’s parameters, resulting in a large sweet-spot. This conference paper solidifies this observation by using global sensitivity analysis to characterize the

sensitivities of the parameters. Using a NASA design problem, called the General Aviation Aircraft problem, we demonstrated that the performance of many MOEAs tend to be dependent on their crossover/mutation operator parameters, which are difficult to correctly set a priori. The Borg MOEA, on the other hand, depends only on NFE, the number of function evaluations, which is easily controlled by the user.

Reed, P. M. and D. Hadka. *“Evolving Many-Objective Water Management to Exploit Exascale Computing.”* Water Resources Research, 50(10):8367-8373, 2014.

For readers interested in parallel computation to solve expensive optimization problems, two parallel variants of the Borg MOEA have been developed: the master-slave and multi-master Borg MOEAs. The master-slave follows the traditional parallelization strategy of the same name, allowing pure speedup of the algorithm by distributing function evaluations on up to 500-1000 processors, depending on the problem. For running on supercomputers with upwards of 1,000,000 processors, we developed the multi-master Borg MOEA using a modified hierarchical island-model parallelization strategy. The modified version seeks to reduce unnecessary communication among islands in order to avoid communication and computational bottlenecks. Experimental results shown in this paper demonstrate the multi-master Borg MOEA can support Petascale and (theoretically) Exascale computing.

Links to the online details for each paper and, where available, the PDF version are available on <http://borgmoea.org/>.

Borg MOEA License

THE PENNSYLVANIA STATE UNIVERSITY RESEARCH AND EDUCATIONAL USE LICENSE

BE ADVISED: This Agreement is a legal agreement between "LICENSEE" (defined below) and The Pennsylvania State University, a non-profit corporation duly organized and existing under the laws of the Commonwealth of Pennsylvania ("UNIVERSITY"). By installing, downloading, accessing or otherwise using the SOFTWARE (defined below), LICENSEE agrees to be bound by the terms of this Agreement. If LICENSEE does not agree with the terms of this Agreement, do not install, access, or use the SOFTWARE.

WHEREAS, Dr. Patrick Reed and Dr. David Hadka, employees of The Pennsylvania State University, have created and developed certain computer software entitled "BORG: Many-Objective Evolutionary Computing Framework" (the "SOFTWARE");

WHEREAS, patent protection for the SOFTWARE is being pursued by the UNIVERSITY including, but not limited to, United States patent 8856054;

WHEREAS, the UNIVERSITY is dedicated to fostering and advancing scientific research, this SOFTWARE is made available to the noncommercial research and educational community as a public service by the UNIVERSITY. [Please contact the UNIVERSITY Office of Technology Management for information on other licensing arrangements (i.e. proprietary commercial applications)].

WHEREAS, the UNIVERSITY is the owner of and has the right to license the SOFTWARE and the associated patent rights;

NOW THEREFORE, in consideration of the premises as well as the mutual promises and covenants set forth herein and for other good and valuable consideration, the receipt and sufficiency of which is hereby acknowledged, the parties hereby agree as follows:

1. Definitions

"LICENSEE" shall mean the person installing the SOFTWARE if it is solely for personal use by that person on the personal equipment of that person. If the SOFTWARE is being installed on equipment for use by another legal entity, such as a corporation, limited liability company, partnership, or institution, then the person installing the SOFTWARE by proceeding with the installation certifies that he or she has authority to bind that legal entity to this Agreement; and that legal entity shall be considered to be the LICENSEE.

2. Grant and Obligations

Under this License Agreement, LICENSEE is granted nonexclusive and royalty-free rights to practice any patent rights associated with the SOFTWARE, and the SOFTWARE may be used as-is or may be modified, provided that your practice or use is solely for noncommercial, research and education purposes. You may not distribute the original version or modifications of the SOFTWARE for any purpose. Both the original

version and modified versions of the SOFTWARE may be copied and performed provided that the following terms and conditions are met:

2.1) LICENSEE is affiliated with a nonprofit, noncommercial research and educational institution and shall identify the name of the institution and the names(s) of the users of the SOFTWARE to UNIVERSITY via Patrick Reed at the time that LICENSEE acquires this license;

2.2) Modified versions of the SOFTWARE must clearly state that the work is a modification of the SOFTWARE and must prominently display the full SOFTWARE title and appropriate copyright notices. Copyright management information (software identifier and version number, copyright notice and license) shall be retained in all versions of the SOFTWARE.

2.3) If modified, the source code, documentation, and user run-time elements of the SOFTWARE should be clearly labeled by identifying the name of the person(s) and/or organization making the modifications, the date of modification, and a written description of the modifications.

2.4) The UNIVERSITY may make modifications to the SOFTWARE that are substantially similar to modified versions of the SOFTWARE created by others, and may make, use, sell, copy, distribute, publicly display, and perform such modifications, including making such modifications available under this or other licenses, without obligation or restriction.

2.5) Neither the name nor any of the logos of the UNIVERSITY may be used in advertising or publicity pertaining to the use or modification of the SOFTWARE without the specific, prior written permission of an authorized representative of the UNIVERSITY.

2.6) Any publications, reports, course lectures, and/or presentations that employ the SOFTWARE or its derivatives will acknowledge the use of the SOFTWARE or of any of its derivatives by citing the SOFTWARE developed by Dr. Reed and Dr. Hadka.

3. Disclaimer of Warranty

THIS SOFTWARE IS MADE AVAILABLE "AS-IS." THE PENNSYLVANIA STATE UNIVERSITY DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, WITH REGARD TO THIS SOFTWARE, INCLUDING WITHOUT LIMITATION, THE WARRANTY OF NONINFRINGEMENT AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND IN NO EVENT SHALL THE PENNSYLVANIA STATE UNIVERSITY BE LIABLE FOR ANY DAMAGES WHATSOEVER, INCLUDING DIRECT, SPECIAL, INDIRECT, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY USE OR PERFORMANCE OF THE SOFTWARE INCLUDING, WITHOUT LIMITATION, LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION FOR CONTRACT, TORT (INCLUDING NEGLIGENCE) OR STRICT LIABILITY.