Vectorizing Fitness Functions and Calling External Functions from Matlab

Aaron M. Cramer
10/16/06

While executing a genetic algorithm (GA), the fitness function is evaluated many times per generation. By default, GOSET performs these evaluations using a `for` loop. By The MathWorks' own admission the performance of `for` loops in Matlab leaves something to be desired. The Matlab documentation suggests that the matrix-oriented nature of Matlab makes matrix and vector calculations more efficient than the equivalent calculations in a `for` loop. It is possible to improve the performance of GOSET on certain problems by removing this `for` loop from the GA.

One technique for removing this `for` loop is to vectorize the fitness function. In particular, there is a field of the `GAP` structure called `ev_bev`. By default this block evaluation field has a value of `false`. However, if it were feasible to calculate the fitness of each individual at the same time, with one function evaluation, then this value should be set to `true`. Consider Rosenbrock's problem,

$$f(x_1, x_2) = (1 - x_1)^2 + 100\left(x_2 - x_1^2\right)^2 \tag{1}$$

The fitness function for this problem can be described using the function in Listing 1.

Listing 1 Scalar fitness function (`scalarfitness.m`)

```
function fitness = scalarfitness(x)

f = (1-x(1))^2+100*(x(2)-x(1)^2)^2;
fitness = 1/(f+0.001);
```

A script file like that shown in Listing 2 can be used to optimize the Rosenbrock function.

Listing 2 Scalar script (`scalarscript.m`)

```
tic;

GAP = gapdefault;

GAP.fp_ipop = 1000;
GAP.fp_npop = 1000;

%               x1, x2
GAP.gd_min  = [-10,-10];
GAP.gd_max  = [ 10, 10];
GAP.gd_type = [  2,  2];
GAP.gd_cid  = [  1,  1];

[fP,GAS] = gaoptimize(@scalarfitness,GAP,[],[],[],[]);

toc;
```

However, it is possible to perform all fitness function evaluations at one time using Matlab's elementwise arithmetic operators as shown in Listing 3.

Listing 3 Vector fitness function (`vectorfitness.m`)

```
function fitness = vectorfitness(x)

f = (1-x(1,:)).^2+100*(x(2,:)-x(1,:).^2).^2;
fitness = 1./(f+0.001);
```

This vectorized fitness function can be called using a script like that shown in Listing 4. Note that the block evaluation field is set to `true`.

Listing 4 Vector script (`vectorscript.m`)

```
tic;

GAP = gapdefault;

GAP.fp_ipop = 1000;
GAP.fp_npop = 1000;

GAP.ev_bev = true;

%               x1, x2
GAP.gd_min  = [-10,-10];
GAP.gd_max  = [ 10, 10];
GAP.gd_type = [  2,  2];
GAP.gd_cid  = [  1,  1];

[fP,GAS] = gaoptimize(@vectorfitness,GAP,[],[],[],[]);

toc;
```

Note that this technique requires that it is possible to express the fitness function in terms of vector operations. This is possible for many fitness functions, and many tricks associated with performing this vectorization exist. However, there are functions which may require a `for` loop to evaluate the fitness of multiple individuals. It is possible to use `ev_bev` and write a `for` loop in the fitness function, but this does not provide any savings. This effectively moves the `for` loop from GOSET's evaluation function into the fitness function, but both `for` loops will be equally slow. A more appropriate solution may be to move the `for` loop into code that does not incur a performance penalty for executing `for` loops. One such environment is a dynamic link library (DLL) written in C. It is possible to write a function in C that evaluates the fitness of multiple individuals using a `for` loop (in C, not Matlab) that can be called from GOSET using block evaluation. Consider the C file and its header file in Listings 5 and 6.

Listing 5 C implementation of fitness (`rosenbrock.c`)

```
#include "rosenbrock.h"

void populationFitness(unsigned int n, double *x, double *fitness)
{
    unsigned int counter;
    double f;

    for(counter = 0; counter < n; counter++)
    {
        f = (1.0 - x[2 * counter]) * (1.0 - x[2 * counter]) +
            100.0 * (x[2 * counter + 1] –
            x[2 * counter] * x[2 * counter]) *
            (x[2 * counter + 1] –
            x[2 * counter] * x[2 * counter]);
        fitness[counter] = 1.0 / (f + 0.001);
    }
}
```

Listing 6 C header file (`rosenbrock.h`)

```
__declspec(dllexport) void populationFitness(unsigned int, double*, double*);
```

To call this C function a fitness function like that shown in Listing 7 and a script file shown in Listing 8 can be used.

Listing 7 DLL fitness function (`dllfitness.m`)

```
function fitness = dllfitness(x)

N = size(x,2);

fitness = NaN(1,N);

[x,fitness] = calllib('rosenbrock','populationFitness',N,x,fitness);
```

Listing 8 DLL script (`dllscript.m`)

```
tic;

GAP = gapdefault;

GAP.fp_ipop = 1000;
GAP.fp_npop = 1000;

GAP.ev_bev = true;

%               x1, x2
GAP.gd_min  = [-10,-10];
GAP.gd_max  = [ 10, 10];
GAP.gd_type = [  2,  2];
GAP.gd_cid  = [  1,  1];

if ~libisloaded('rosenbrock')
    loadlibrary('rosenbrock','rosenbrock.h');
end

[fP,GAS] = gaoptimize(@dllfitness,GAP,[],[],[],[]);

unloadlibrary('rosenbrock');

toc;
```

To properly choose which of these methods is most efficient, it is necessary to consider several questions. Does the performance penalty associated with the `for` loop outweigh the (usually low) cost of loading and unloading the DLL? Are there many generations? Are there many individuals? There is a performance penalty for each call to the DLL associated with making the function call. Is fitness evaluation the dominant bottleneck in the algorithm? Making the fitness evaluation faster is useless if it was not the problem to begin with. For example, in the simple problem presented above, each method had approximately the same runtime. These considerations should lead to the proper strategy for fitness evaluation.