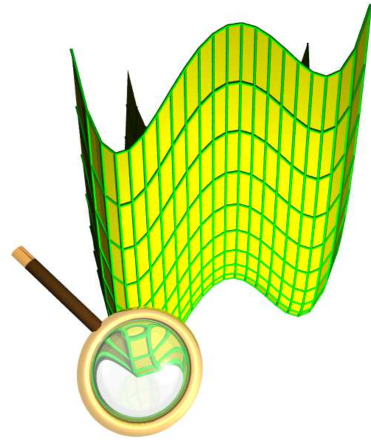


Reactive Search: Continuous Optimization

User Manual

Version 3.1— August 21, 2009

Reactive Search: Learning on the Job

Reactive Search is a robust and efficient method for solving difficult optimization problems. The word *reactive* hints at a ready response to events *while* alternative solutions are tested. Its strength lies in the introduction of high-level skills often associated to the human brain, such as learning from the past experience, learning on the job, rapid analysis of alternatives, ability to cope with incomplete information, quick adaptation to new situations and events.

The main features of the Reactive Search techniques are:

Learning on the job Real-world problems have a rich structure. While many alternative solutions are tested in the exploration of a search space, patterns and regularities appear. The human brain quickly learns and drives future decisions based on previous observations. This is the main inspiration source for inserting online *machine learning* techniques into the optimization engine of Reactive Search.

Rapid generation and analysis of many alternatives Often, to solve a problem one searches among a large number of alternatives, each requiring the analysis of *what-if* scenarios. The search speed is improved if alternatives are generated in a strategic manner, so that different solutions are chained along a trajectory in the search space *exploring* wide areas and rapidly *exploiting* the most promising solutions.

Flexible decision support Crucial decisions depend on factors and priorities which are not always easy to describe before starting the solution process. Feedback from the user in the preliminary exploration phase can be incorporated so that a better tuning of the final solutions takes the end user preferences into account.

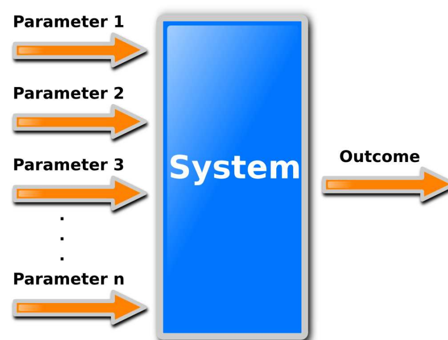
Diversity of solutions The final decision is up to *you*, not the machine. The reason is that not all qualitative factors of a problem can be encoded into a computer program. Having a set of diverse near-best alternatives is often crucial for the decision maker.

Anytime solutions You decide when to stop searching. A first complete solution is generated rapidly, better and better ones are produced in the following search phases. The more you run, the bigger the possibility to identify excellent solutions, but if you want a solution fast you are going to get it!

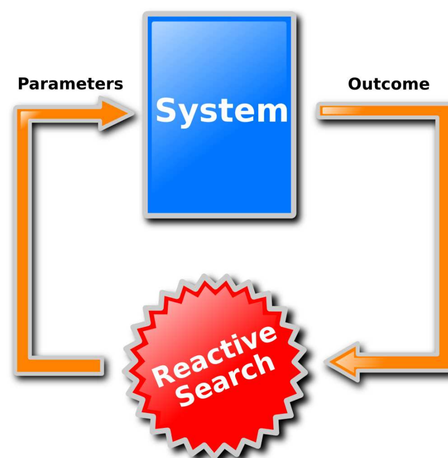
Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved

Continuous optimization

A scenario for applying Reactive Search is where a *system* (an electronic system, a production plant, a fleet of trucks, a business process) requires you to set some operating *parameters* (knobs, switches, travel plans, procedures) to improve its functionality. Depending on how the parameters are set, the system is going to give a better or worse *outcome* (measured by production speed, net income, fuel consumption, customer satisfaction. . .), as shown by the following figure.



In order to optimize the outcome, a simple loop is performed: set the parameters, observe the outcome, then change the parameters in a strategic and intelligent manner until a suitable solution is identified.



The intelligence of the process lies in the Reactive Search (RS) component, which decides about the next step to take based on information collected during the ongoing exploration of alternatives. In order to operate efficiently, Reactive Search uses *memory* and *intelligence*, to recognize ways to improve solutions in a directed and focussed manner.

Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved

The `RS-ContinuousOptimization` library provides a simple way for the user to solve problems with continuous parameters. The programmer defines the system to be optimized with a C++ function and invokes a powerful Reactive Search solver that aims at locating the best values for the operating parameters.

The main library functionalities are first explained by means of a simple tutorial which enables any C++ programmer to quickly insert an optimization task into any program. Then a complete reference to the library's functions and parameters is provided.

Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved

Two steps to solve it!

We assume that the operating parameters and the process outcome are real numbers. Moreover, the system outcome represents a *loss* to be minimized (production time for one item, fuel cost. . .). In other words, we are facing a "real-valued, or continuous, multivariate minimization" problem.

In order to define this problem in a programming language, we need to define three items:

- the number of operating parameters which we are able to set (a.k.a. the *dimensionality* of the problem);
- the range (minimum and maximum value) within which each parameter can be varied (a.k.a. the *domain* of the problem);
- a procedure that, given values for the operating parameters $x_0, \dots, x_{n-1}$, returns the outcome of the system for those particular values (the so-called *objective function*).

For a concrete example, let's assume that the system has three operating parameters (called x_0, x_1, x_2), varying in the intervals $-1 \le x_0 \le 1$, $-2 \le x_1 \le 4$, $0 \le x_2 \le 5$, and that the outcome of the system is equal to the following objective function:

$$\text{myobjective}(x_0, x_1, x_2) = x_0 + x_1^2 + \cos^2 x_2.$$

Step 1: Define the problem

Open a new file, call it `test.cpp`. Start by including the standard `iostream` header and the library's header `ContinuousOptimization.h`:

```
#include <iostream>
#include <cmath>
#include "RS-CO.h"
using namespace std;
```

Define the number of parameters and the range of variability of the variables:

```
const int dimension = 3;
const double minarg[] = { -1.0, -2.0, 0.0 };
const double maxarg[] = { 1.0, 4.0, 5.0 };
```

Note that the name of the variables is up to you, and that two separate arrays are needed to store the minimum and maximum value in the range of each parameter.

Write the objective function to be minimized. The name can be decided by the programmer, provided that the prototype remains the same:

```
double myobjective (const double *x, void*)
{
   double y = cos(x[2]);
   return x[0] + x[1] * x[1] + y * y;
}
```

Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved

Note that the array indices range from 0 to (dimension − 1), as it is standard practice in C and C++. The second parameter can be used to pass additional data to the function without using a global environment. In our case, the second argument is not used.

Step 2: Call the optimizer and solve it

Once the domain and the objective function are defined, deploy the search technique. Within the `main` function, initialize the search algorithm:

```
RSCO_initialize(myobjective, dimension, minarg, maxarg, RSCO_RASH, 0,
NULL);
```

Invoke a 60-second run of the optimization algorithm:

```
RSCO_optimize(60);
```

Retrieve the minimum value of the function and the corresponding values of the parameters:

```
double v;
const double *x = RSCO_best (v);
```

Print the result that you have just retrieved:

```
cout << "Best value " << v << " found at";
for ( int i = 0; i < dimension; i++ )
   cout << 'x' << i << " = " << x[i] << endl;
```

Compile the program and link it with the `libRS-CO.a` library:

```
g++ -o test test.cpp libRS-CO.a
```

Run it and enjoy the results!

Note for Microsoft Windows programmers

The native Windows SDK version of the library is called `libRS-CO.lib`; the compile command must be

```
cl /Fetest.exe /MD /EHsc /GL test.cpp libRS-CO.lib user32.lib
```

However, all options shall be automatically set up for you by the IDE during the project setup.

Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved

Library reference

Implemented Reactive Search techniques

The `ContinuousOptimization` library implements two basic Reactive Local Search algorithms:

RASH (Reactive Affine Shaker) is a fast algorithm which can quickly find a local minimum by continuously adjusting its local search area. Its use is advisable when the system to be optimized is smooth or has a small number of local minima.

C-RTS (Continuous Reactive Tabu Search) is a robust technique for global optimization in the presence of a more structured and complex problem.

When unsure, C-RTS is recommended, however it has higher memory requirements and it takes more time per iteration, so the programmer is encouraged to explore both alternatives.

Definitions

Function type `RSCO_objectiveFunction`

This typedef defines the prototype of the function to be optimized:

```
typedef double RSCO_objectiveFunction (const double* x, void* extra);
```

The function must return the value corresponding to the parameter vector `x` passed as first argument.

The second parameter, `extra`, is a pointer used to pass additional data to the function without using a global environment, with the guarantee that the optimization framework won't modify it; the function can modify the contents of the data pointed by the second parameter. New
See also: parameter `extra` of function RSCO_initialize().

Enumeration type `enum RSCO_SolverType`

The enumeration type has only two possible values. It is used in the `RSCO_initialize()` function in order to specify which solver algorithm must be used by the library:

`RSCO_RASH`
This value refers to the RASH algorithm.

`RSCO_CRTS`
This value refers to the C-RTS algorithm.

When unsure, choose C-RTS.

Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved

Function RSCO_initialize()

The function

```
void RSCO_initialize (RSCO_objectiveFunction f, int dim
                      const double * minarg, const double * maxarg,
                      RSCO_SolverType type, int seed, void* extra)
```

must be called before the optimizer is used.

Parameters

RSCO_objectiveFunction f

The function to be optimized, passed as a pointer to a user-defined C function.

int dim

The number of variables of function f.

const double * minarg

An array with the lower bounds of all function variables.

const double * maxarg

An array with the upper bounds of all function variables.

RSCO_SolverType type

The chosen optimization procedure. Choose RSCO_RASH for a quick local-minimum inspection, RSCO_CRTS for a more robust global optimization.

int seed

The random number generator seed. Use the same value for reproducible results.

void* extra

New

Optional pointer to additional data that will be passed as second parameter to the objective function without modifications by the optimization framework.
See also: parameter extra of function type RSCO_objectiveFunction.

Function RSCO_optimize()

The function

```
void RSCO_optimize (int time)
```

is used to invoke an optimization run with a given duration in seconds.

Parameters

int time

The integer number of seconds allotted for the optimization run.

Observations for UNIX developers

This function runs for the specified amount of time. It schedules a SIGALRM signal after the given time, so the user program can safely interrupt it by sending a SIGALRM signal to itself.

If the user program is using SIGALRM for any other purpose, calling this function may disrupt it. In this case, repeated calls to the RSCO_step() functions (see below) provide the same functionality.

Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved

Function RSCO_best()

The function

const double * RSCO_best (**double** * v)

provides the best value of the objective function up to the moment it is invoked and the parameters values that yield it.

Parameters

double * v

A pointer to a double variable to be filled with the best value found by the library up to the current step. Set to NULL if the value is not needed.

Return value

The function returns a pointer to a constant array containing the coordinates corresponding to the value stored into v.

Observations

Consider that a constant array pointer is returned, so the user code should not be allowed to modify its contents.

Function RSCO_step()

The function

void RSCO_step (**double** * v)

is used to invoke a single step of the solver algorithm. An optimization run can be implemented by repeatedly calling this function.

Parameters

double * v

A pointer to a double variable to be filled with the best value found by the library up to the current step. Set to NULL if the value is not needed.

Function RSCO_set()

New

The function

void RSCO_set (**const double*** x)

resets all parameters and sets the initial step of the RASH solver to the x vector.

Parameters

const double * x

Initial coordinates of the next RASH step.

Observations

The function is only effective for the RSCO_RASH local optimizer, and it does not affect the RSCO_CRTS global optimizer.

Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved

Using the library functions

Please refer to the program described in the "two steps" example given before.

Algorithm initialization

The following call will select the C-RTS algorithm and will set the random seed to the UNIX epoch (which ensures a new set of values every second).

```
RSCO_initialize(myobjective, dim, minarg, maxarg, RSCO_CRTS, 0);
```

Searching

The search can be run for a fixed amount of time, say 5 seconds, by calling

```
RSCO_optimize(5);
```

Alternatively, a loop can be implemented where the `RSCO_step()` function is repeatedly called:

```
double v;
for ( i = 0; i < max_steps; i++ )
   RSCO_step (v);
```

Note the double variable `v`, which is needed in order to store the minimum value.

If the parameter settings corresponding to the minimum value are needed, they can be obtained at any time by a call to the `RSCO_best()` function, referenced by a constant vector reference:

```
double v;
const double * x = RSCO_best (v);
```

Platform-specific information

Unix system development

On Unix-like systems (Linux, Mac OS X, Win32/Cygwin) the functions are provided on the static library file `libRS-CO.a`, with header in `RS-CO.h`.

The usage examples in the distribution directory show how to link the library into the user program.

Windows (non-Cygwin) development

On native Windows systems, the RS-CO library is delivered in two flavors:

Static library
: Library file: `libRS-CO.lib`; Header file: `RS-CO.h`.

Dynamic library
: Library interface file: `RS-CO.lib`; dynamic file: `RS-CO.dll`; Header file: `RS-CO-import.h`.

See the Makefile in the distribution directory for an example of linking your user program to the DLL.

Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved

Common questions

How can I get different results every time I run the program?

The `RS-CO` library, like most optimization programs, has a random component, so that it is not required to perform the same choices all times. This ensures a much more robust behavior, compare for instance what happens if you present the same hard problem to different human experts.
In many cases, however, presenting the same solution every time the program executes on the same problem is important. Here are the two simple steps to make to ensure reproducibility:

- Always pass the same non-zero positive integer as random number generator seed to the `RSCO_initialize()` function.
- Use a fixed number of calls to `RSCO_step()` to find a solution.

If you wish to obtain potentially different results for each run, please do the following:

- Either pass different values to the `seed` parameter of the `RSCO_initialize()` function.
- you can choose whether to call `RSCO_step()` repeatedly, or `RSCO_optimize()` once: the latter executes for a fixed period of time, so that its performance depends on CPU speed and load.

I need to *maximize* a function — How can I do?

While the `ContinuousOptimization` library operates function minimization, a simple trick will do: change the sign of the objective function, i.e., put a minus sign before the returned expression in your implementation of `objectiveFunction()`.

Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved

List of files

The package contains the following files:

`RS-CO-manual.pdf`
This file.

`RS-CO.h`
The library's header file, to be included in the user's program.

`libRS-CO.a` or `libRS-CO.lib`
The static library file, to be linked to the user's program.

`example1.cpp`
A sample user program employing the library.

`example2.cpp`
A more complex sample user program employing the library.

`Makefile`
A sample Makefile to compile the example programs.

Windows only:

`RS-CO-import.h`, `RS-CO.lib` and `RS-CO.dll`
Dynamic library files.

`example1-dll.cpp`
A sample user program employing the library.

Reactive-Search.com, © 1999–2009 Roberto Battiti and Mauro Brunato,
all rights reserved