SPOT Package Vignette (Version 0.1.1065)

T. Bartz-Beielstein, J. Ziegenhirt, M. Zaefferer

October 14, 2010

1 Introduction

The main goal of this vignette is to demonstrate the usage of SPOT Package functions and their interfaces. For more detailed information on how to work with spot scientificly, check the literature, as this is basically just a technical description on how to use the different functions. A more detailed manual that explains how to use these functions successfully can be seen in other SPOT vignette, SPOTmanual.

As an easy example this vignette will demonstrate how to apply SPOT to a Simulated Annealing (SANN) algorithm. Besides reading additional Literature about Simulated Annealing you can also check the documentation of the SANN implementation in R by typing:

```
> help(optim)
```

in your R console. That command opens a description of the optim function implemented in the standard R-Distribution. The SANN method is one of the methods used by optim.

SANN is a heuristic optimization algorithm, in this case used to determine the minimum of the Branin function, which is a common test function often used to test optimization methods. It is non-separable and has three minima with the same function value, which means all three optima are global. The optimum (minimum) of the test function is at $x_1 = (-\pi, 12.275), x_2 = (\pi, 2.275), x_3 = (9.425, 2.475) y_{1,2,3} = (0.397887)$. This is a good reference for us to later see how well the SANN worked with the parameters chosen by SPOT, thus having an estimation of the performance of our SPOT run.

So we have 2 levels of optimization. On level one SANN is used to find the optimum of the Branin function. On level two SPOT is used to find the optimal parameters for SANN. But before we can start the experiments we need to do some preparations.

2 Preparations for this Example

If SPOT was installed properly, it should first be loaded into the workspace with the following command.

```
> require("SPOT")
```

For our example some important files a distributed with the package. The packages will be located in a default or user defined folder on your system. This folder can be obtained by using the .find.package() function of R. We then have the folder of our spot installation, this folder in- cludes a folder with the needed files for our example, called "demo12Sann". Since we need that specific folder, we will add it to the path variable.

```
> demoPath <- .find.package("SPOT")
> demoPath <- file.path(demoPath, "demo12Sann")
```

If you want to change this example and try your own settings or use another algorithm in stead of the SANN, this is the folder you will have to look at. The files distributed in this folder are described in the following chapter. If you want to create your own project for testing with spot, you can just copy the important files from this folder to your own, rename them and change the content as needed. Of course you will then need to change the demoPath accordingly, so that it refers to your own project folder.

3 Description of project files

The files related to our example can be separated into two groups. Group one are the files that are related to the problem (the SANN algorithm and the target function). Group two are the files related to SPOT, defining options and parameters for our SPOT run.

3.1 Algorithm files

For group one there are two files.

3.1.1 Function call file

The first one is a function call to the SANN Algorithm startSannAlgorithm.R. In there is the function spot will call when trying to evaluate new parameter settings. The branin function as used in this sample is also included in that file.

3.1.2 Algorithm design file

The second of the algorithm related files, is the .apd file. That file describes some settings for the algorithm that will not be touched by SPOT.

```
f = "Branin0.0"
x0 = c(10,10)
parscale = c(20,20)
n = 2
maxit = 1000
```

—Listing: sann0001.apd—

f describes what function the SANN will optimize. That string will be used in some output files and on screen as a descriptive string. x0 is the startpoint for each run of SANN, parscale is a scaling vector, n is the problem dimension (of the target function, i.e. Branin), and maxit is the maximum number of iterations performed by the SANN algorithm, which is also the only stopping criterion for SANN.

3.2 SPOT files

The second group consists of a .conf file and a .roi file, which describe how SPOT should do the meta optimization of SANN.

3.2.1 Region of Interest file

The .roi file describes the Region Of Interest, which means the region of the parameter space that SPOT will examine.

```
name low high type
TEMP 1 100 FLOAT
TMAX 1 100 FLOAT
```

—sann0001.roi—

As you can see the two parameters that SPOT will optimize are named in here, and both given an upper (high) and lower (low) boundary. In this case both parameters will vary between values of 1 and 100. Also there is a data type set for them, in this case FLOAT. TEMP is the starting temperature for the SANN, while TMAX is the number of function evaluations for each Temperature.

3.2.2 SPOT configuration file

The .conf file describes the configuration of SPOT itself. There can be quite many settings in that file, tho most do not have to be in there because SPOT is able to chose default settings if none are set by the user. First there are some settings regarding the Algorithm:

```
alg.func = "startSannAlgorithm"
alg.seed = 1235
```

—partial Listing: sann0001.conf—

The function name of the algorithm is given with alg.func. The alg.seed is used as a starting random seed for the algorithm, it will be iterated if the same setting is evaluated repeatedly.

Another seed to keep track of is the random seed for SPOT itself:

```
spot.seed = 125
```

—partial Listing: sann0001.conf—

Now there will follow some settings which regard the different steps of SPOT.

The auto step:

```
auto.loop.steps = 5;
```

—partial Listing: sann0001.conf—

There is an option to use an automated SPOT run, that in this case will sequentially create 5 meta models. In our example however we will actually go step by step, so this setting will be ignored by spot.

The initial step:

```
init.design.func = "spotCreateDesignLhd";
init.design.size = 5;
```

—partial Listing: sann0001.conf—

The initial step creates a design, which means a number of points that cover the parameter space defined in the roi file. In this case the LHD (Latin Hypercube Design) is used for sampling this initial design. There will be 5 parameter settings created.

The sequential step:

```
seq.design.maxRepeats = 4;
seq.design.size = 100
seq.predictionModel.func = "spotPredictTree"
```

—partial Listing: sann0001.conf—

In the sequential step the results of preceding steps get evaluated with a predictor to create new design points. In this case the "spotPredictTree" function is used, which uses a Tree model. Other possible predictors that are allready included in the SPOT package can be seen in comments in the .conf file. The design.size determines how many points of the predicted model will be evaluated, the best ones in this list will be evaluated by spot in the following step.

Any other additional settings for the above mentioned steps, or for other steps, will be used with default values. If you want to check the default values of all parameters you can see them in the help of the spotGetOptions() function.

```
> help(spotGetOptions)
```

Make sure you used the "require" command as shown above, so R can actually load that help file.

4 Running the Example

Since we allready have the path leading to our test files, starting a complete SPOT run would be really easy by using the "auto" command. For better understanding however we will not use an "auto" run, but rather start the different steps of SPOT manually, to explain each one in turn. Of course doing those steps manually is not just to help comprehensibility of our example. Certain problems require so much run time, that it might be more convenient to decide after each single step how to proceed. For doing so we need 2 inputs for our spot() function: The main configuration file sann0001.conf and the task or step we need to start. The configuration file is in the path of our project:

```
> testFile <- file.path(demoPath, "sann0001.conf")
```

With that information we can now start the different SPOT steps.

4.1 Initialisation

The first step of course is the initialisation, wich means we need to pass "init" to our function:

```
> myConfigAndResults = spot(testFile, "init")

spot.R::spot started
Create Inital Design
  Entering spotDesignLhd
     low high  type
TEMP   1  100 FLOAT
TMAX   1  100 FLOAT
  Leaving spotCreateDesignLhd
```

This creates some output in the console, telling us which region of interest was chosen, and what files were created. However the most important thing, that the "init" step did, was creating the initial design. The initial design is a number of points in our search space, in this case a latin hypercube design. Since we chose the init.design.size in our .conf file with "5", there should be 5 points created, covering the whole search space. If you want to see those, they are part of the myConfigAndResults list. If the config parameter spot.fileMode was set to TRUE you could also look them up in the .des File that the "init" step then creates. It is located in the same directory as the other files of our example project. Since we allways start the example with the same random seeds, the result should look like this:

TEMP TMAX CONFIG REPEATS SEED
23.7707780034747 6.39907314046286 1 1 1235
69.0059362523724 21.9058858577162 2 1 1235
59.6085319525562 70.2274352503475 3 1 1235
93.8123979566153 93.9297357015312 4 1 1235
16.5563532061875 52.133604357997 5 1 1235

—Initial Design—

The values chosen for the parameters to be optimized are between 1 and 100, as defined in our .roi file. Also ever configuration got assigned a "CONFIG" number, a number of "REPEATS" (according to init.design.repeats in .conf) and a random "SEED" to be tested with. The next step, called the "run" step, will use the data in this file .

Also note that running "init" will delete files created by subsequent steps. So if you want to repeat an experiment with different settings, you should rename the conf file, so that SPOT will create new output instead of overwriting old output.

4.2 Running the algorithm with the initial design

This step basically just runs the algorithm to be optimized (SANN) on the created design points.

```
> myConfigAndResults = spot(testFile, "run", spotConfig = myConfigAndResults)

spot.R::spot started
spotStepRunAlg started with /var/folders/NH/NHuXaHt2HEW3bDr3cc0dnE+++TI/-Tmp-//Rinst34397738
```

The results of that algorithm run will be written into a .res file or in the result varialbe myConfigAndResults:

Y TEMP TMAX FUNCTION DIM SEED CONFIG STEP
0.398463820722149 23.7707780034747 6 Branin0.0 2 1235 1 0
0.41532359335735 69.0059362523724 22 Branin0.0 2 1235 2 0
0.410713521008908 59.6085319525562 70 Branin0.0 2 1235 3 0
0.408397362895338 93.8123979566153 94 Branin0.0 2 1235 4 0
0.398808497955464 16.5563532061875 52 Branin0.0 2 1235 5 0

—Initial Results—

For each of the design points the "run" step evaluated a Y-value of the target function. That Y-value is what SANN finds with the Parameters chosen by SPOT.

4.3 Adding additional/sequential design points

Now that the initial design points are evaluated we need to come to the most important part, the prediction. Untill now all we did was create some

sample points and evaluate them. Now we need to make a prediction about the underlying function, and chose one or more new design point in our search space to see if we can get some improvement. This is called the sequential step, since it will usually be repeated sequentially.

```
> myConfigAndResults = spot(testFile, "seq", spotConfig = myConfigAndResults)

spot.R::spot started
Create Sequential Design
Entering generateSequentialDesign
  Entering spotPrepareData
  Leaving spotPrepareData
  Entering spotDesignLhd
     low high  type
TEMP   1  100 FLOAT
TMAX   1  100 FLOAT
  Leaving spotCreateDesignLhd
spotPredictTree started
      TEMP     TMAX
1 21.96110 32.05517
2 33.07064 51.62648
spotPredictTree finished
  Leaving generateSequentialDesign
```

While doing so, the best design point of the initial design will be saved into a .bst file, before SPOT creates the new design point. Also, the best design will be evaluated again for the next step, as we can see in the updated design file:

```
TEMP TMAX CONFIG REPEATS repeatsLastConfig STEP SEED
23.7707780034747 6 1 1 1 1 1236
36.6815126021695 77.464058787974 6 2 2 1 1235
59.3121275147679 32.5098091041157 7 2 2 1 1235
```

—Sequential Design—

As we can see, the second design listed in the res file above is repeated, because it had the best Y-value. Also the predictor chose two new configurations which can now be evaluated by the ”run” step.

```
> myConfigAndResults = spot(testFile, "run", spotConfig = myConfigAndResults)

spot.R::spot started
spotStepRunAlg started with /var/folders/NH/NHuXaHt2HEW3bDr3cc0dnE+++TI/-Tmp-//Rinst34397738
```

```
Y TEMP TMAX FUNCTION DIM SEED CONFIG STEP
0.398463820722149 23.7707780034747 6 Branin0.0 2 1235 1 0
0.41532359335735 69.0059362523724 22 Branin0.0 2 1235 2 0
0.410713521008908 59.6085319525562 70 Branin0.0 2 1235 3 0
0.408397362895338 93.8123979566153 94 Branin0.0 2 1235 4 0
0.398808497955464 16.5563532061875 52 Branin0.0 2 1235 5 0
0.401623784100991 23.7707780034747 6 Branin0.0 2 1236 1 1
0.405034984877384 36.6815126021695 77 Branin0.0 2 1235 6 1
0.398556687821417 36.6815126021695 77 Branin0.0 2 1236 6 1
0.409063730358596 59.3121275147679 33 Branin0.0 2 1235 7 1
2.7324685600925 59.3121275147679 33 Branin0.0 2 1236 7 1
```

—Sequential Results—

Each new design got evaluated twice, because spot uses the default increase function to increase the number of repeats for each sequential step up to 4, since seq.design.maxRepeats is set to 4 in the .conf file. The best of the previous step is only repeated once, because it allready was evaluated once. Since it was started with a new random seed, it led to a different and in this case much worse value. For the following steps SPOT will use the default merge function (mean value) to merge the different results into one value.

We will now quietly repeat those two steps (”seq” and”run”) 5 times, to get some reasonable data for our tree model. If you want to see the results, just check the .res file.

```
> myConfigAndResults = spot(testFile, "seq", spotConfig = myConfigAndResults)
> myConfigAndResults = spot(testFile, "run", spotConfig = myConfigAndResults)
> myConfigAndResults = spot(testFile, "seq", spotConfig = myConfigAndResults)
> myConfigAndResults = spot(testFile, "run", spotConfig = myConfigAndResults)
> myConfigAndResults = spot(testFile, "seq", spotConfig = myConfigAndResults)
> myConfigAndResults = spot(testFile, "run", spotConfig = myConfigAndResults)
> myConfigAndResults = spot(testFile, "seq", spotConfig = myConfigAndResults)
> myConfigAndResults = spot(testFile, "run", spotConfig = myConfigAndResults)
> myConfigAndResults = spot(testFile, "seq", spotConfig = myConfigAndResults)
> myConfigAndResults = spot(testFile, "run", spotConfig = myConfigAndResults)
```

Now of course we could just go on doing that, to create another additional design and evaluate them as shown above over and over again. Which would be alot easier if we used the ”auto” step.

However we could also modify our SPOT settings between each of those steps, for instance to change them to focus on areas of the parameter space that seems interesting according to the results. This wouldnt be possible if the ”auto” step was used. How this kind of interaction between user and SPOT could be done is also explained in the manual that was mentioned in the introduction. At this point we will just show how to ask SPOT for a report about the results that were produced by the above described steps.

4.4 Getting a report

Like the other steps, the Report-Step is based on the settings in the .conf file. So we will just check what results we get with our current settings.

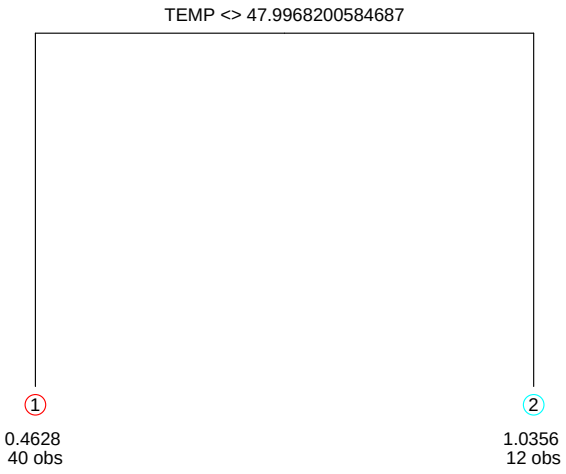
```
> myConfigAndResults = spot(testFile, "rep", spotConfig = myConfigAndResults)

spot.R::spot started
  Entering spotReportDefault
       y               TEMP            TMAX
 Min.   :0.3979   Min.   :16.45   Min.   : 6.00
 1st Qu.:0.3985   1st Qu.:22.25   1st Qu.:52.00
 Median :0.3999   Median :24.55   Median :75.00
 Mean   :0.5703   Mean   :38.56   Mean   :65.85
 3rd Qu.:0.4206   3rd Qu.:56.24   3rd Qu.:91.00
 Max.   :4.3438   Max.   :93.81   Max.   :95.00
  Entering spotPrepareData
  Leaving spotPrepareData

 Best solution found with 53 evaluations:
           Y    TEMP TMAX COUNT CONFIG
46 0.4004373 93.8124   94     4      4
```

As we see the "rep" Step tells us what solution SPOT found for our problem and some additional information about our experiment.

The report also creates some graphics, both on screen and as a pdf. The pdf is located in the folder where all the other files are stored, too.



What we see here, is a tree for our SPOT run. This is created by the default report function, as we didnt tell spot to use anything else. The default report only gives text output and tries to draw a tree with the available data. This tree will be drawn regardless of the model used for the sequential steps, as the report step uses the availabe data to create a new tree for data created by any kind of prediction model. Of course the tree is rather small, since the we didnt get much data from so few steps. For real improvement and a bigger tree we would need to invest more time.