Introduction to `ipoptr`: an R interface to Ipopt *

Jelmer Ypma

July 16, 2010

Abstract

This document describes how to use `ipoptr`, which is an R interface to Ipopt (Interior Point Optimizer). Ipopt is an open source software package for large-scale nonlinear optimization (Wächter & Biegler, 2006). It can be used to solve general nonlinear programming problems with nonlinear constraints and lower and upper bounds for the controls. Ipopt is written in C++ and is released as open source code under the Common Public License (CPL). It is available from the COIN-OR initiative. The code has been written by Carl Laird and Andreas Wächter, who is the COIN project leader for Ipopt.

1 Introduction

All credit for implementing the C++ code for Ipopt should go to Andreas Wächter and Carl Laird. Please show your appreciation by citing their paper. This vignette describes the R interface and some of the information here is heavily based on the Ipopt Wiki.

Ipopt is designed to find (local) solutions of mathematical optimization problems of the from

$$\begin{aligned} & \min_{x \in R^n} f(x) \\ s.t. \quad & g_L <= g(x) <= g_U \\ & x_L <= x <= x_U \end{aligned}$$

where $f(x) : R^n \to R$ is the objective function, and $g(x) : R^n \to R^m$ are the constraint functions. The vectors g_L and g_U denote the lower and upper bounds on the constraints, and the vectors x_L and x_U are the bounds on the variables x. The functions $f(x)$ and $g(x)$ can be nonlinear and nonconvex, but should be twice continuously differentiable. Note that equality constraints can be formulated in the above formulation by setting the corresponding components of g_L and g_U to the same value.

This package can not yet be distributed. Downloading and installing Ipopt has to be done separately because of licensing issues (look at the Ipopt website

*This package should be considered in beta and comments about any aspect of the package are welcome. This document is an R vignette prepared with the aid of `Sweave`, Leisch(2002). Financial support of the UK Economic and Social Research Council through a grant (RES-589-28-0001) to the ESRC Centre for Microdata Methods and Practice (CeMMAP) is gratefully acknowledged.

for hints on how to do this successfully). To install (and compile) Ipopt a C/C++ compiler has to be available. On Windows I was succesful using MSYS and Rtools.

2 Installation

Installing the `ipoptr` package is not as straightforward as most other R packages, because it depends on Ipopt, which has to be installed separately. Ipopt and the R interface to Ipopt are distributed separately because of differences in licenses between Ipopt, some of the libraries it depends on, and R. Because of these differences in license, distribution of Ipopt and `ipoptr` together is not allowed. There are ways to simplify the Ipopt part of the installation by using some of the pre-compiled libraries of Ipopt with Mumps[1].

Besides a working installation of Ipopt, you need to be able to compile R packages from source. On Windows platforms Rtools can be helpful, on Linux a compiler is usually available. First, a file `Makevars` in the source directory of the interface package needs to be configured for your system from the file `Makevars.in`. The `configure_ipoptr.R` script should make this easier. Start R, go to the directory where the `ipoptr` interface is located and load the script.

```
> setwd('~/ipoptr')
> source('configure_ipoptr.R')
```

Then run the `configure_ipoptr` command with the directory of the Ipopt build and the directory where `ipoptr` is located as arguments, e.g.

```
> configure_ipoptr( ipopt.build.dir = "/cvos/shared/apps/Ipopt/3.5.5-source",
                    ipoptr.dir = "~/ipoptr" )
```

The `ipoptr.build.dir` should have a file called `Makefile` in the subdirectory `/Ipopt/examples/hs071_cpp`. If there were no errors, then a file called `Makevars` (or `Makevars.win` if you're using Windows) has now been created in the directory `ipoptr/src`. You can then install the package with the command

```
install.packages('~/ipoptr', repos=NULL, type='source')
```

where the first argument specifies the directory where the R interface to Ipopt is located. Ipopt should now be available from R, which you can test.

```
> library('ipoptr')
> ?ipoptr
```

3 Minimizing the Rosenbrock Banana function

As a first example we will solve an unconstrained minimization problem. The function we look at is the Rosenbrock Banana function

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2$$

[1] Any help to simplify the installation process is appreciated and if someone uses the pre-compiled libraries and has a `Makefile`, please let me know.

which is also used in the example given in the documentation for the standard R optimizer `optim`. The gradient of the objective function is given by

$$\nabla f(x) = \begin{pmatrix} -400 \cdot x_1 \cdot (x_2 - x_1^2) - 2 \cdot (1 - x_1) \\ 200 \cdot (x_2 - x_1^2) \end{pmatrix}.$$

Ipopt always needs gradients to be supplied by the user. After loading the library

```
> library(ipoptr)
```

we start by specifying the objective function and its gradient

```
> ## Rosenbrock Banana function
> eval_f <- function(x) {
      return( 100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2 )
  }
> ## Gradient of Rosenbrock Banana function
> eval_grad_f <- function(x) {
      return( c( -400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
                  200 * (x[2] - x[1] * x[1]) ) )
  }
```

We define initial values

```
> # initial values
> x0 <- c( -1.2, 1 )
```

and then minimize the function using the `ipoptr` command. This command runs some checks on the supplied inputs and returns an object with the exit code of the solver, the optimal value of the objective function and the solution. The checks do not always return very informative messages, but usually there is something wrong with dimensions (e.g. `eval_grad_f` returns a vector that doesn't have the same size as `x0`).

```
> # solve Rosenbrock Banana function
> res <- ipoptr( x0=x0,
                 eval_f=eval_f,
                 eval_grad_f=eval_grad_f )
```

These are the minimal arguments that have to be supplied. If, like above, no Hessian is defined, Ipopt uses an approximation. We can see the results by printing the resulting object.

```
> print( res )
Call:

ipoptr(x0 = x0, eval_f = eval_f, eval_grad_f = eval_grad_f)



Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
```

```
convergence tolerances (can be specified by options). )

Number of Iterations....: 47
Optimal value of objective function:  3.09761879321718e-19
Optimal value of controls: 1 1
```

It's advised to always check the exit code for convergence of the problem and in this case we can see that the algorithm terminated successfully. Ipopt used 47 iterations to find the solution and the optimal value of the objective function and the controls are given as well.

If you do not want to, or cannot calculate the gradient analytically, you can supply a function `eval_grad_f` that approximates the gradient. However, this is not advisable and might result in convergence problems, for instance by not finding the minimum, or by finding the wrong minimum. We can see this from the following example where we approximate `eval_grad_f` using finite differences

```
> # Approximate eval_f using finite differences
> # http://en.wikipedia.org/wiki/Numerical_differentiation
> approx_grad_f <- function( x ) {
     minAbsValue     <- 0
     stepSize        <- sqrt( .Machine$double.eps )

     # if we evaluate at 0, we need a different step size
     stepSizeVec     <- ifelse( abs(x) <= minAbsValue,
                                stepSize^3,
                                x * stepSize )

     x_prime <- x
     f       <- eval_f( x )
     grad_f  <- rep( 0, length(x) )
     for (i in 1:length(x)) {
         x_prime[i]      <- x[i] + stepSizeVec[i]
         stepSizeVec[i]  <- x_prime[i] - x[i]

         f_prime         <- eval_f( x_prime )
         grad_f[i]       <- ( f_prime - f )/stepSizeVec[i]
         x_prime[i]      <- x[i]
     }

     return( grad_f )
 }
```

and using this approximation to minimize the same Rosenbrock Banana function.

```
> # increase the maximum number of iterations
> opts <- list("tol"=1.0e-8, "max_iter"=5000)
> # solve Rosenbrock Banana function with approximated gradient
> print( ipoptr( x0=x0,
                 eval_f=eval_f,
```

```
                 eval_grad_f=approx_grad_f,
                 opts=opts) )
Call:

ipoptr(x0 = x0, eval_f = eval_f, eval_grad_f = approx_grad_f,
    opts = opts)


Ipopt solver status: 1 ( MAXITER_EXCEEDED: Maximum number
of iterations exceeded (can be specified by an option). )

Number of Iterations....: 5000
Current value of objective function:  1.9797550219875e-11
Current value of controls: 0.9999956 0.999991
```

In this case 5000 iterations are not enough to solve the minimization problem to the required tolerance. This has to do with the step size we choose to approximate the gradient

```
> sqrt( .Machine$double.eps )
[1] 1.490116e-08
```

which is of the same order of magnitude. If we decrease the tolerance, the algorithm converges, but the solution is less precise than if we supply gradients and it takes more iterations to get there.

```
> # decrease the convergence criterium
> opts <- list("tol"=1.0e-7)
> # solve Rosenbrock Banana function with approximated gradient
> print( ipoptr( x0=x0,
                 eval_f=eval_f,
                 eval_grad_f=approx_grad_f,
                 opts=opts) )
Call:

ipoptr(x0 = x0, eval_f = eval_f, eval_grad_f = approx_grad_f,
    opts = opts)


Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
convergence tolerances (can be specified by options). )

Number of Iterations....: 50
Optimal value of objective function:  1.98034174754174e-11
Optimal value of controls: 0.9999956 0.999991
```

4 Sparse matrix structure

Ipopt can handle sparse matrices. The sparseness structure should be defined in advance and stay the same throughout the minimization procedure. A sparseness structure can be defined as a list of vectors, where each vector contains the indices of the non-zero elements of one row. E.g. the matrix

$$\begin{pmatrix} . & . & . & 1 \\ 1 & 1 & . & . \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

has a non-zero element in position 4 in the first row. In the second row it has non-zero elements in position 1 and 2, and the third row contains non-zero elements at every position. Its structure can be defined as

```
> sparse_structure <- list( c( 4 ), c( 1, 2 ), c( 1, 2, 3, 4 ) )
```

The function `make.sparse` can simplify this procedure

```
> make.sparse( rbind( c(0, 0, 0, 1), c( 1, 1, 0, 0 ), c( 1, 1, 1, 1 ) ) )
[[1]]
[1] 4

[[2]]
[1] 1 2

[[3]]
[1] 1 2 3 4
```

The function `print.sparseness` shows the non-zero elements

```
> print.sparseness( sparse_structure )
  1 2 3 4
1 . . . 1
2 2 3 . .
3 4 5 6 7
```

By default `print.sparseness` shows the indices of the non-zero elements in the sparse matrix. Values for the non-zero elements of a sparse matrix have to be supplied in one vector, in the same order as the the non-zero elements occur in the structure. I.e. the order of the indices matters and the values of the following two matrices should be supplied in a different order

```
> print.sparseness( list( c(1,3,6,8), c(2,5), c(3,7,9) ) )
  1 2 3 4 5 6 7 8 9
1 1 . 2 . . 3 . 4 .
2 . 5 . . 6 . . . .
3 . . 7 . . . 8 . 9
> print.sparseness( list( c(3,1,6,8), c(2,5), c(3,9,7) ) )
  1 2 3 4 5 6 7 8 9
1 2 . 1 . . 3 . 4 .
2 . 5 . . 6 . . . .
3 . . 7 . . . 9 . 8
```

Since the sparseness structure defines the indices of non-zero elements by row, the order of the rows cannot be changed in the R implementation. In principle a more general order of the non-zero elements (independent of row or column) could be specified, but this is not high on the priority list. Below are two final examples on sparseness structure (see `?print.sparseness` for more options and examples)

```
> # print lower-diagonal 5x5 matrix generated with make.sparse
> A_lower <- make.sparse( lower.tri( matrix(1, nrow=5, ncol=5), diag=TRUE ) )
> print.sparseness( A_lower )
   1  2  3  4  5
1  1  .  .  .  .
2  2  3  .  .  .
3  4  5  6  .  .
4  7  8  9 10  .
5 11 12 13 14 15
> # print a diagonal 5x5 matrix without indices counts
> A_diag  <- make.sparse( diag(5) > 0 )
> print.sparseness( A_diag, indices=FALSE )
  1 2 3 4 5
1 x . . . .
2 . x . . .
3 . . x . .
4 . . . x .
5 . . . . x
```

For larger matrices it is easier to plot them using the `plot.sparseness` command

```
> s <- do.call( "cbind", lapply( 1:5, function(i) {
                                         diag(5) %x% matrix(1, nrow=5, ncol=20)
                                      } ) )
> s <- do.call( "rbind", lapply( 1:5,  function(i) { s } ) )
> s <- cbind( matrix( 1, nrow=nrow(s), ncol=40 ), s )
> plot.sparseness( make.sparse( s ) )
```

The resulting sparse matrix structure from this code can be seen in figure 1.

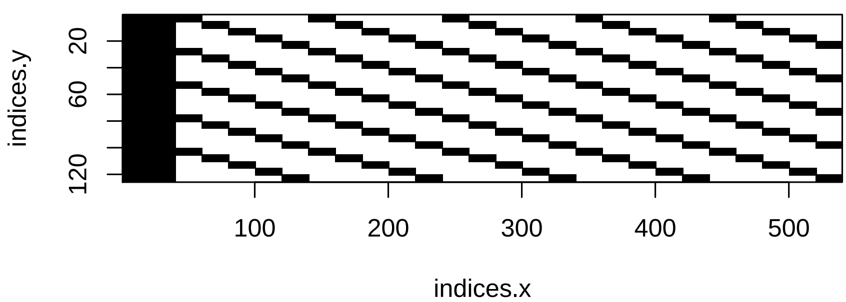

Figure 1: Plot of large sparseness structure

5 Supplying the Hessian

Now that we know how to define a sparseness structure we can supply the Hessian to the Rosenbrock Banana function from above. Its Hessian is given by

$$\nabla^2 f(x) = \begin{pmatrix} 2 - 400 \cdot (x_2 - x_1^2) + 800x_1^2 & -400x_1 \\ -400x_1 & 200 \end{pmatrix}$$

Ipopt needs the Hessian of the Lagrangian in the following form

$$\sigma_f \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x),$$

where $g_i(x)$ represents the ith of m constraints, λ_i are the multipliers of the constraints and σ_f is introduced so that Ipopt can ask for the Hessian of the objective or the constraints independently if required.

In this case we don't have any constraints. The user-defined function `eval_f` takes three arguments. The first argument contains the value of the control variables, x, the second argument contains the multiplication factor of the Hessian of the objective function, σ_f, and the third argument contains a vector with the multipliers of the constraints, λ. We can define the structure of the Hessian and the function to evaluate the Hessian as follows

```
> # The Hessian for this problem is actually dense,
> # This is a symmetric matrix, fill the lower left triangle only.
> eval_h_structure <- list( c(1), c(1,2) )
> eval_h <- function( x, obj_factor, lambda ) {
    return( obj_factor*c( 2 - 400*(x[2] - x[1]^2) + 800*x[1]^2,      # 1,1
                          -400*x[1],                                  # 2,1
                          200 ) )                                     # 2,2
 }
```

Note that we only specify the lower half of the Hessian, since it is a symmetric matrix. Also, `eval_h` returns a vector with all the non-zero elements of the Hessian. Then we minimize the function using the `ipoptr` command

```
> opts <- list("print_level"=0,
             "file_print_level"=12,
             "output_file"="banana.out",
             "tol"=1.0e-8)
> # solve Rosenbrock Banana function with analytic Hessian
> print( ipoptr( x0=x0,
                 eval_f=eval_f,
                 eval_grad_f=eval_grad_f,
                 eval_h=eval_h,
                 eval_h_structure=eval_h_structure,
                 opts=opts) )
Call:

ipoptr(x0 = x0, eval_f = eval_f, eval_grad_f = eval_grad_f, eval_h = eval_h,
    eval_h_structure = eval_h_structure, opts = opts)
```

```
Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
convergence tolerances (can be specified by options). )

Number of Iterations....: 21
Optimal value of objective function:  3.74397564313947e-21
Optimal value of controls: 1 1
```

Here we also supplied options to not print any intermediate information to the R screen (`print_level=0`). Printing output to the screen directly from Ipopt does not work in all R terminals correctly, so it might be that even though you specify a positive number here, there will still be no output visible on the screen. If you want to print things to the screen, a workaround is to do this directly in the R functions you defined, such as `eval_f`.

Also, to inspect more details about the minimization we can write all the output to a file, which will be created in the current working directory. For larger problems, having a large number for `file_print_level` can easily generate files a couple of Gigabytes large, which is probably not desirable. Many more options are available, and a full list of all the options is available at the Ipopt website, `http://www.coin-or.org/Ipopt/documentation/node59.html#app.options_ref`. Options can also be supplied from an option file, that can be specified in `option_file_name`.

6 Adding constraints

To look at how we can add constraints to a problem, we take example problem number 71 from the Hock-Schittkowsky test suite, which is also used in the Ipopt C++ tutorial. The problem is

$$
\begin{aligned}
& \min_x x_1 * x_4 * (x_1 + x_2 + x_3) + x_3 \\
s.t. & \\
& x_1 * x_2 * x_3 * x_4 >= 25 \\
& x_1^2 + x_2^2 + x_3^2 + x_4^2 = 40 \\
& 1 <= x_1, x_2, x_3, x_4 <= 5,
\end{aligned}
$$

and we use $x = (1, 5, 5, 1)$ as initial values. In this problem we have one inequality constraint, one equality constraint and upper and lower bounds for all the variables. The optimal solution is $(1.00000000, 4.74299963, 3.82114998, 1.37940829)$. First we define the objective function and its gradient

```
> eval_f <- function( x ) {
      return( x[1]*x[4]*(x[1] + x[2] + x[3]) + x[3] )
 }
> eval_grad_f <- function( x ) {
      return( c( x[1] * x[4] + x[4] * (x[1] + x[2] + x[3]),
                 x[1] * x[4],
                 x[1] * x[4] + 1.0,
                 x[1] * (x[1] + x[2] + x[3]) ) )
 }
```

Then we define a function that returns the value of the two constraints. We define the bounds of the constraints (in this case the g_L and g_U are 25 and 40) later.

```
> # constraint functions
> eval_g <- function( x ) {
    return( c( x[1] * x[2] * x[3] * x[4],
               x[1]^2 + x[2]^2 + x[3]^2 + x[4]^2 ) )
 }
```

Then we define the structure of the Jacobian, which is a dense matrix in this case, and function to evaluate it

```
> eval_jac_g_structure <- list( c(1,2,3,4), c(1,2,3,4) )
> eval_jac_g <- function( x ) {
    return( c ( x[2]*x[3]*x[4],
                x[1]*x[3]*x[4],
                x[1]*x[2]*x[4],
                x[1]*x[2]*x[3],
                2.0*x[1],
                2.0*x[2],
                2.0*x[3],
                2.0*x[4] ) )
 }
```

The Hessian is also dense, but it looks slightly more complicated because we have to take into account the Hessian of the objective function and of the constraints at the same time, although you could write a function to calculate them both separately and then return the combined result in `eval_h`.

```
> # The Hessian for this problem is actually dense,
> # This is a symmetric matrix, fill the lower left triangle only.
> eval_h_structure <- list( c(1), c(1,2), c(1,2,3), c(1,2,3,4) )
> eval_h <- function( x, obj_factor, lambda ) {

    values <- numeric(10)
    values[1] = obj_factor * (2*x[4]) # 1,1

    values[2] = obj_factor * (x[4])   # 2,1
    values[3] = 0                     # 2,2

    values[4] = obj_factor * (x[4])   # 3,1
    values[5] = 0                     # 4,2
    values[6] = 0                     # 3,3

    values[7] = obj_factor * (2*x[1] + x[2] + x[3]) # 4,1
    values[8] = obj_factor * (x[1])                 # 4,2
    values[9] = obj_factor * (x[1])                 # 4,3
    values[10] = 0                                  # 4,4


    # add the portion for the first constraint
```

```
        values[2] = values[2] + lambda[1] * (x[3] * x[4]) # 2,1

        values[4] = values[4] + lambda[1] * (x[2] * x[4]) # 3,1
        values[5] = values[5] + lambda[1] * (x[1] * x[4]) # 3,2

        values[7] = values[7] + lambda[1] * (x[2] * x[3]) # 4,1
        values[8] = values[8] + lambda[1] * (x[1] * x[3]) # 4,2
        values[9] = values[9] + lambda[1] * (x[1] * x[2]) # 4,3

        # add the portion for the second constraint
        values[1] = values[1] + lambda[2] * 2 # 1,1
        values[3] = values[3] + lambda[2] * 2 # 2,2
        values[6] = values[6] + lambda[2] * 2 # 3,3
        values[10] = values[10] + lambda[2] * 2 # 4,4

        return ( values )
    }
```

After the hard part is done, we only have to define the initial values, the lower and upper bounds of the control variables, and the lower and upper bounds of the constraints. If a variable or a constraint does not have lower or upper bounds, the values `-Inf` or `Inf` can be used. If the upper and lower bounds of a constraint are equal, Ipopt recognizes this as an equality constraint and acts accordingly.

```
> # initial values
> x0 <- c( 1, 5, 5, 1 )
> # lower and upper bounds of control
> lb <- c( 1, 1, 1, 1 )
> ub <- c( 5, 5, 5, 5 )
> # lower and upper bounds of constraints
> constraint_lb <- c(  25, 40 )
> constraint_ub <- c( Inf, 40 )
> opts <- list("print_level"=0,
              "file_print_level"=12,
              "output_file"="hs071_nlp.out")
> print( ipoptr( x0=x0,
                eval_f=eval_f,
                eval_grad_f=eval_grad_f,
                lb=lb,
                ub=ub,
                eval_g=eval_g,
                eval_jac_g=eval_jac_g,
                constraint_lb=constraint_lb,
                constraint_ub=constraint_ub,
                eval_jac_g_structure=eval_jac_g_structure,
                eval_h=eval_h,
                eval_h_structure=eval_h_structure,
                opts=opts) )
Call:
```

```
ipoptr(x0 = x0, eval_f = eval_f, eval_grad_f = eval_grad_f, lb = lb,
    ub = ub, eval_g = eval_g, eval_jac_g = eval_jac_g, eval_jac_g_structure = eval_jac_
    constraint_lb = constraint_lb, constraint_ub = constraint_ub,
    eval_h = eval_h, eval_h_structure = eval_h_structure, opts = opts)
```

```
Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
convergence tolerances (can be specified by options). )

Number of Iterations....: 8
Optimal value of objective function:  17.0140171451792
Optimal value of controls: 1 4.743 3.82115 1.379408
```

7 Using data

The final subject we have to cover, is how to pass data to an objective function or the constraints. This is achieved by writing a wrapper function around the objective function with the parameters as an argument, or by defining an environment that holds the data, and passing this environment to `ipoptr`. Both methods are shown in `tests/parameters.R`, here I will only show the second.

As a very simple example[2] suppose we want to find the minimum of

$$f(x) = a_1x^2 + a_2x + a_3$$

for different values of the parameters a_1, a_2 and a_3. First we define the objective function and its gradient using, assuming that there is some variable `params` that contains the values of the parameters.

```
> eval_f <- function(x) {
     return( params[1]*x^2 + params[2]*x + params[3] )
 }
> eval_grad_f <- function(x) {
     return( 2*params[1]*x + params[2] )
 }
```

Then we define an environment that contains specific values of `params`

```
> # define a new environment that contains params
> auxdata        <- new.env()
> auxdata$params <- c(1,2,3)
```

To solve this we supply `auxdata` as an argument to `ipoptr`, that will take care of evaluating the functions in the correct environment, so that auxiliary data is available.

[2]A more interesting example is given in `tests/lasso.R`

```
> # pass the environment that should be used to evaluate functions to ipoptr
> ipoptr( x0                 = 0,
         eval_f             = eval_f,
         eval_grad_f        = eval_grad_f,
         ipoptr_environment = auxdata )
Call:

ipoptr(x0 = 0, eval_f = eval_f, eval_grad_f = eval_grad_f, ipoptr_environment = auxdata


Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
convergence tolerances (can be specified by options). )

Number of Iterations....: 1
Optimal value of objective function:  2
Optimal value of controls: -1
```

8 Options

There are many options available, all of which are described on the Ipopt website. One of the options can test whether your derivatives are correct. This option is activated by setting `derivative_test` to `first-order` or `second-order` if you want to test second derivatives as well. This process can take quite some time. To see all the output from this process you can set `derivative_test_print_all` to `yes`, preferably when writing to a file, because of the problems with displaying on some terminals mentioned above. Without this options the derivative checker only shows those lines where an error occurs.

9 Remarks

If you run many large optimization problems in a row on Windows, at some point you'll get errors that mumps is running out of memory and you won't get any solutions. On Linux this same problem hasn't occurred yet.

The R terminal in Windows doesn't show any output. The Linux one does.

References

Leisch, F. (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In W. Härdle & B. Rönz (Eds.), *Compstat 2002 — proceedings in computational statistics* (pp. 575–580). Physica Verlag, Heidelberg. Available from `http://www.stat.uni-muenchen.de/ leisch/Sweave` (ISBN 3-7908-1517-9)

Wächter, A., & Biegler, L. T. (2006). On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, *106*(1), 25–57.