Vectorised objective functions

Enrico Schumann
es@enricoschumann.net

1 Introduction

Heuristics often manipulate and evolve solutions through functions: new solutions are created as functions of existing solutions; solutions are evaluated through the objective function; whether new solutions are accepted is a function of (typically) the quality of the new solutions; and so on. This gives us much flexibility in how solutions are represented; in essence, any data structure (eg, a graph) could be directly handled, provided we define appropriate functions to work with it.

Yet a number of (quite successful) heuristics, such as Differential Evolution (DE) or Particle Swarm (PS), prescribe precisely how solutions are represented and manipulated. In fact, these specific prescriptions essentially define those heuristics. For DE and PS, for instance, a solution is a numeric vector; new solutions are created as (noisy) linear combinations of existing solutions. While this reduces the algorithms' flexibility, it allows for a simpler (and more efficient) generic implementation.

Let us be more concrete here. Since both DE and PS represent solutions as numerical vectors, a natural way to store the solutions is a matrix P. In this matrix, each column is one solution; each row represents a specific decision variable. When we compute the objective function values for these solutions, a straightforward strategy is to loop over the columns of P and call the objective function for each solution. In this case, the objective function should take as arguments a single numeric vector (and possibly other data passed through `...`); the function should return a single number.

In somes cases, however, it may be preferable to actually write the objective function such that it expects the whole population as an argument, and then returns a vector of objective function values. To accommodate this behaviour, the functions `DEopt`, `GAopt` and `PSopt` have settings `algo$loopFun`, in which '`Fun`' can be '`OF`' for objective function, but also, for instance, '`repair`'. These settings default to `TRUE`, so the functions will loop over the solutions. When such a loop-setting is `FALSE`, the respective function receives the whole population as an argument.

In the next section we give three examples when this 'evaluation in one step' can be advantegeous. The functions `DEopt`, `GAopt` and `PSopt` allow to implement the objective function (and also repair and penalty functions) like this. For more details and examples, see Gilli et al. [2011].

2 Examples for vectorised computations

We attach the package.

```
> require("NMOF")
```

2.1 A test function

As an example, we use the Rosenbrock function, given by

$$\sum_{i=1}^{n-1} \left(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right) .$$

This test function is available in the package as the function `tfRosenbrock` (see `?testFunctions`). The Rosenbrock function has a minimum of zero when all elements of x are one. (In higher dimensions, this minimum may not be unique.)

```
> tfRosenbrock
```

```
function (x)
{
    n <- length(x)
    xi <- x[seq_len(n - 1L)]
    sum(100 * (x[2L:n] - xi * xi)^2 + (1 - xi)^2)
}
<environment: namespace:NMOF>
```

So we define the objective function `OF` and test it with the known solution.

```
> OF <- tfRosenbrock     ## see ?testFunctions
> size <- 5L             ## set dimension
> x <- rep.int(1, size)  ## the known solution ...
> OF(x)                  ## ... should give zero
```

```
[1] 0
```

We set the parameters for `DEopt`. Note that in this example we are only concerned with the speed of the computation, so the actual settings do not matter so much.

```
> algo <- list(printBar = FALSE,
                      nP = 50L,
                      nG = 500L,
                       F = 0.6,
                      CR = 0.9,
                     min = rep(-100, size),
                     max = rep( 100, size))
```

Suppose we have several solutions, put into a matrix such that every column is one solution. Then we could rewrite the function like so:

```
> ## a vectorised OF: works only with *matrix* x
> OF2 <- function(x) {
      n <- dim(x)[1L]
      xi <- x[1L:(n - 1L), ]
      colSums(100 * (x[2L:n, ] - xi * xi)^2 + (1 - xi)^2)
  }
```

We can test it by creating a number of random solutions.

```
> x <- matrix(rnorm(size * algo$nP), size, algo$nP)
> c(OF(x[ ,1L]), OF(x[ ,2L]), OF(x[ ,3L]))
```

```
[1]   25.925 1233.278  615.801
```

```
> OF2(x)[1L:3L]  ## should give the same result
```

```
[1]   25.925 1233.278  615.801
```

```
> all.equal(OF2(x)[1L:3L], c(OF(x[ ,1L]), OF(x[ ,2L]), OF(x[ ,3L])))
```

```
[1] TRUE
```

As pointed out above, `DEopt` either can loop over the solutions, or it can evaluate the whole population in one step. The first behaviour is triggered when `algo$loopOF` is set to `TRUE`, which is the default setting.

When we want to use `OF2`, we need to set `algo$loopOF` to `FALSE`.

```
> set.seed(1223445)
> (t1 <- system.time(sol <- DEopt(OF = OF, algo = algo)))
```

```
Differential Evolution.
Best solution has objective function value 5.0103e-16 ;
standard deviation of OF in final population is 2.9557e-15 .

   user  system elapsed
  0.089   0.003   0.091
```

```
> algo$loopOF <- FALSE
> set.seed(1223445)
> (t2 <- system.time(sol2 <- DEopt(OF = OF2, algo = algo)))
```

```
Differential Evolution.
Best solution has objective function value 5.0103e-16 ;
standard deviation of OF in final population is 2.9557e-15 .

   user  system elapsed
  0.015   0.000   0.015
```

We can compare the solutions, and compute the speedup.

```
> sol$OFvalue    ## both should be zero (with luck)
```

```
[1] 5.0103e-16
```

```
> sol2$OFvalue
```

```
[1] 5.0103e-16
```

```
> t1[[3L]]/t2[[3L]]  ## speedup
```

```
[1] 6.0667
```

2.2 Portfolio optimisation

A portfolio can be described by a weight vector w. Given a variance–covariance matrix Σ, we can calculate the variance of such a portfolio like so:

$$w'\Sigma w\,.$$

Suppose now that we have a number of solutions, and we collect them in a matrix W, such that every column is one solution w. One approach would be now to loop over the columns, and for every column compute the variance. But we can use a one-line computation as well: the variances of the solutions are given by

$$\mathsf{diag}(W'\Sigma W)\,.$$

This can be written consisely, but we are unnessarily computing the off-diagonal elements of the resulting matrix. One solution, then, is to recognise that $\mathsf{diag}(W'\Sigma W)$ is equivalent to

$$\iota' \overbrace{\underbrace{\Sigma W}_{\substack{\text{matrix}\\\text{multiplication}}} W}^{\substack{\text{elementwise}\\\text{multiplication}}}$$

which is consise and more efficient. The following example illustrates this. We start by setting up a variance–covariance matrix `Sigma` and a population `W`. (We would not need to include the budget constraint here since we are only interested in computing time.)

```
> na <- 100L  ## number of assets
> np <- 100L  ## size of population
> trials <- seq_len(100L)  ## for speed test
> ## a covariance matrix
> Sigma <- array(0.7, dim = c(na, na)); diag(Sigma) <- 1
> ## set up population
> W <- array(runif(na * np), dim = c(na, np))
> ## budget constraint
> scaleFun <- function(x) x/sum(x); W <- apply(W, 2L, scaleFun)
```

Now we can test the three variants described above.

```
> ## variant 1
> t1 <- system.time({
  for (i in trials) {
      res1 <- numeric(np)
      for (j in seq_len(np)) {
          w <- W[ ,j]
          res1[j] <- w %*% Sigma %*% w
      }
  }
  })
> ## variant 2
> t2 <- system.time({
      for (i in trials) res2 <- diag(t(W) %*% Sigma %*% W)
  })
> ## variant 3
> t3 <- system.time({
      for (i in trials) res3 <- colSums(Sigma %*% W * W)
  })
```

All three computations should give the same result.

```
> all.equal(res1,res2)
```

```
[1] TRUE
```

```
> all.equal(res2,res3)
```

```
[1] TRUE
```

But the first variant requires more code than the others, and it is slower.

```
> ## time required
> #  ... variant 1
> t1
```

```
   user  system elapsed
  0.715   0.988   0.144
```

```
> ## ... variant 2
> t2
```

```
   user  system elapsed
  0.123   0.181   0.025
```

```
> ## ... variant 3
> t3
```

```
   user  system elapsed
  0.061   0.085   0.012
```

2.3 Residuals in a linear model

We wish to compute the residuals r of a linear model, $y = X\theta + r$. Suppose we have a population Θ of solution vectors; each column in Θ is one particular solution θ. Now, as before we could compute

$$\mathrm{r} = y - X\theta_i$$

for every $i \in \{1, \ldots, \text{population size}\}$. Alternatively, we may replace the loop over those solutions with the computation

$$\mathrm{R} = y\iota' - X\Theta,$$

in which R is the matrix of residuals.

Again, an example. As before, we set up random data and a random population of solutions.

```
> n  <- 100L  # number of observation
> p  <- 5L    # number of regressors
> np <- 100L  # population size
> trials <- seq_len(1000L)
> ## random data
> X <- array(rnorm(n * p), dim = c(n, p))
> y <- rnorm(n)
> ## random population
> Theta <- array(rnorm(p * np), dim = c(p, np))
> ## empty residuals matrix
> R1 <- array(NA, dim = c(n, np))
```

Now we can compare both variants.

```
> system.time({
      for (i in trials)
          for (j in seq_len(np))
              R1[ ,j] <- y - X %*% Theta[ ,j]
  })
```

```
   user  system elapsed
  0.537   0.607   0.254
```

```
> system.time({
      for (i in trials)
          R2 <- y - X %*% Theta
  })
```

```
   user  system elapsed
  0.024   0.012   0.036
```

Note that we have not explicitly computed $y\iota'$ but have used R's recycling rule.

We check whether we actually obtain the same result.

```
> all.equal(R1, R2)  ## ... should be TRUE
```

```
[1] TRUE
```

See Chapter 14 in Gilli et al. [2011].

References

Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Elsevier, 2011.