turboEM: An R package to accelerate the EM algorithm

1 Introduction

In the R package "turboEM" four schemes to accelerate the EM algorithm have been implemented. The four accelerators are SQUAREM [1], Parabolic EM [2], a quasi-Newton algorithm [3], and Dynamic ECME [4]. The main function `turboem()` allows the user to fit one or more of the acceleration schemes for a given starting value and fixed point mapping function. The `turboem()` function outputs an object of class `turbo`. Methods implemented for the `turbo` class are `print`, `pars`, `error`, `plot`, `grad`, `hessian`, and `stderror`.

We have also developed methods to summarize and display results from a simulation study comparing accelerators. For an object resulting from a simulation study of class `turbosim`, the methods `summary`, `boxplot`, `dataprof`, and `pairs` have been implemented.

In this document we first provide an overview of the `turbem` function. We then demonstrate its use through an example. Last, we demonstrate how a simulation study to compare acceleration schemes may be conducted and results visualized using the class `turbosim` and its corresponding methods.

2 Implementation

First load the `turboEM` R package, which includes the `turboem` function and its associated methods.

```
> library(turboEM)
```

Usage

```
> turboem(par, fixptfn, objfn, method = c("em", "squarem", "pem",
+     "decme", "qn"), boundary, pconstr = NULL, project = NULL,
+     ..., control.method = replicate(length(method), list()),
+     control.run = list())
```

Arguments

- `par`: A vector of parameters denoting the initial guess for the fixed point
- `fixptfn`: A vector function F that denotes the fixed-point mapping. This function should accept a parameter vector as input and should return a parameter vector of the

same length. This function defines the fixed-point iteration $x_{k+1} = F(x_k)$. In the case of EM algorithm, F defines a single E- and M-step.

- `objfn`: This is a scalar function, L, that denotes a "merit" function which attains its local minimum at the fixed-point of F. This function should accept a parameter vector as input and should return a scalar value. In the EM algorithm, the merit function L is the negative log-likelihood. In some problems, a natural merit function may not exist. However, this argument is required for all of the algorithms *except* SQUAREM (which defaults to SQUAREM-2 if `objfn` not provided) and EM.
- `method`: Specifies which algorithm(s) will be applied. Must be a vector containing one or more of `c("em", "squarem", "pem", "decme", "qn")`.
- `boundary`: Argument required for Dynamic ECME (`"decme"`) only. Function to define the subspaces over which the line search is conducted.
- `pconstr`: Optional function for defining boundary constraints on parameter values. Function maps a vector of parameter values to TRUE if constraints are satisfied. Note that this argument is only used for SQUAREM (`squarem`), Parabolic EM (`pem`), and quasi-Newton (`qn`) algorithms, and it has no effect on the other algorithms.
- `project`: Optional function for defining a projection that maps an out-of-bound parameter value into the constrained parameter space. Requires the `pconstr` argument to be specified in order for the `project` to be applied.
- `control.method`: if `method = c(method1, method2, ...)`, then `conrol.method = list(list1, list2, ...)` where `list1` is the list of control parameters for `method1`, `list2` is the list of control parameters for `method2`, and so on. If `length(method) == 1`, then `control.method` is the list of control parameters for the acceleration scheme.
- `control.run`: A list of control parameters for convergence and stopping the algorithms. See *Details* under `help("turboem")` and Section 3.2 below.
- ...: Arguments passed to `fixptfn` and `objfn`

Value

The function `turboem` returns an object of class `turbo`. An object of class `turbo` is a list containing at least the following components:

- `fail`: A vector of logical values whose jth element indicates whether algorithm j failed (produced an error)
- `value.objfn`: A vector of the value of the objective function at termination for each algorithm
- `itr`: A vector of the number of iterations completed for each algorithm
- `fpeval`: A vector of the number of fixed-point evaluations completed for each algorithm
- `objfeval`: A vector of the number of objective function evaluations completed for each algorithm
- `convergence`: A vector of logical values whose jth element indicates whether algorithm j satisfied the convergence criterion before termination
- `runtime`: A matrix whose jth row contains the "user", "system", and "elapsed" time for running the jth algorithm
- `errors`: A vector whose jth element is either NA or contains the error message from

running the jth algorithm

- `pars`: A matrix whose jth row contains the fixed-point parameter values at termination for the jth algorithm
- `trace.objfval`: if `control.run[["keep.objfval"]]=TRUE`, contains a list whose jth component is a vector of objective function values across iterations for the jth algorithm

The list will also contain the components `method`, `control.method`, `control.run`, `fixptfn`, `objfn`, `pconstr`, and `project`, which were provided as arguments for `turboem`.

3 Example

The example we consider is a mixture of two Poisson distributions. Data are from Hasselblad (JASA 1969).

```
> poissmix.dat <- data.frame(death = 0:9, freq = c(162, 267, 271,
+     185, 111, 61, 27, 8, 3, 1))
> y <- poissmix.dat$freq
> npar <- 3
```

The fixed point mapping of the EM algorithm may be coded as

```
> fixptfn <- function(p, y) {
+     pnew <- rep(NA, 3)
+     i <- 0:(length(y) - 1)
+     zi <- p[1] * exp(-p[2]) * p[2]^i/(p[1] * exp(-p[2]) * p[2]^i +
+         (1 - p[1]) * exp(-p[3]) * p[3]^i)
+     pnew[1] <- sum(y * zi)/sum(y)
+     pnew[2] <- sum(y * i * zi)/sum(y * zi)
+     pnew[3] <- sum(y * i * (1 - zi))/sum(y * (1 - zi))
+     p <- pnew
+     return(pnew)
+ }
```

The objective function to be minimized (negative log-likelihood) is as follows.

```
> objfn <- function(p, y) {
+     i <- 0:(length(y) - 1)
+     loglik <- y * log(p[1] * exp(-p[2]) * p[2]^i/exp(lgamma(i +
+         1)) + (1 - p[1]) * exp(-p[3]) * p[3]^i/exp(lgamma(i +
+         1)))
+     return(-sum(loglik))
+ }
```

3.1 Illustration of basic features

First, we demonstrate how to use `turboem` to fit the EM algorithm as well as the SQUAREM and Parabolic EM acceleration schemes, using the default settings for each algorithm.

```
> res <- turboem(par = c(0.5, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     method = c("em", "squarem", "pem"), y = y)
> options(digits = 13)
> res

   method    value.objfn  itr fpeval objfeval convergence elapsed.time
1      em 1989.945860141 1500   1500        1       FALSE         0.20
2 squarem 1989.945859883   23     45       24        TRUE         0.00
3     pem 1989.945859883   25     60      176        TRUE         0.03
```

We see that the EM algorithm did not achieve convergence at the default tolerance within the allotted 1500 iterations.

Using the `pars` method, we can look at the parameter values obtained across the three algorithms:

```
> pars(res)

                     p1             p2             p3
em      0.3600250921611 1.256337900747 2.663574957686
squarem 0.3598853970236 1.256095100582 2.663404357078
pem     0.3598853767328 1.256095051266 2.663404340195
```

We can also compute the gradient, hessian, and standard error estimates for the parameter values.

```
> options(digits = 7)
> grad(res)

                [,1]          [,2]         [,3]
em      2.879504e-03  3.303934e-04 1.946839e-04
squarem 5.475554e-08 -5.702013e-08 3.343641e-08
pem     7.360408e-07 -1.211904e-06 6.597645e-07

> hessian(res)

$em
          [,1]       [,2]       [,3]
[1,]  906.9336 -270.26837 -341.16556
[2,] -270.2684  113.52906   61.69322
[3,] -341.1656   61.69322  192.70596

$squarem
          [,1]       [,2]       [,3]
[1,]  907.1089 -270.22931 -341.25284
[2,] -270.2293  113.48027   61.68208
[3,] -341.2528   61.68208  192.78081
```

```
$pem
          [,1]       [,2]       [,3]
[1,]  907.1055 -270.22910 -341.25280
[2,] -270.2291  113.48013   61.68212
[3,] -341.2528   61.68212  192.78081
```

```
> stderror(res)
```

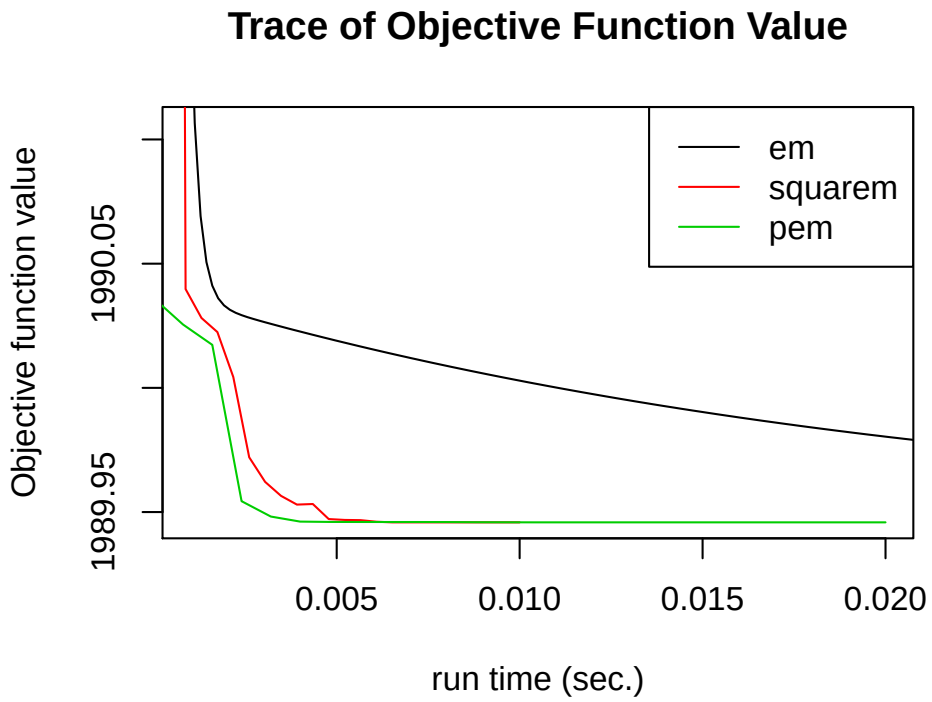
```
             [,1]      [,2]      [,3]
em      0.1947437 0.3500155 0.2505722
squarem 0.1946798 0.3500223 0.2504732
pem     0.1946904 0.3500397 0.2504853
```

We can also compare the objective function value over time across the different methods. To do this, we must specify that we would like `turboem` to store the computed value at each iteration.

```
> res1 <- turboem(par = c(0.5, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     method = c("em", "squarem", "pem"), y = y, control.run = list(keep.objfval = TRU
> res1
```

```
   method value.objfn  itr fpeval objfeval convergence elapsed.time
1      em    1989.946 1500   1500     1501       FALSE         0.22
2 squarem    1989.946   23     45       24        TRUE         0.01
3     pem    1989.946   25     60      176        TRUE         0.02
```

```
> plot(res1, xlim = c(0.001, 0.02))
```



In order to apply the dynamic ECME acceleration scheme, one must specify the subspace over which line searches will be conducted. This is done through specification of the `boundary` function. Given the current parameter value `par` and the direction of the search `dr`, the function defining the subspace at each iteration for this example is given by

```
> boundary <- function(par, dr) {
+     lower <- c(0, 0, 0)
+     upper <- c(1, 10000, 10000)
+     low1 <- max(pmin((lower - par)/dr, (upper - par)/dr))
+     upp1 <- min(pmax((lower - par)/dr, (upper - par)/dr))
+     return(c(low1, upp1))
+ }
```

We can now use `turboem` for the Dynamic ECME algorithm, as follows.

```
> res2 <- turboem(par = c(0.5, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     boundary = boundary, method = "decme", y = y)
> options(digits = 13)
> res2

  method    value.objfn itr fpeval objfeval convergence elapsed.time
1  decme 1989.945859883  32     32      156        TRUE         0.03
```

For some problems, an objective function may not be available. Only SQUAREM and EM do not require an objective function to be provided. The other algorithms, parabolic EM, quasi-Newton, and Dynamic ECME, will produce an error message if no objective function is provided.

```
> res3 <- turboem(par = c(0.5, 1, 3), fixptfn = fixptfn, boundary = boundary,
+     y = y)
> res3

   method value.objfn  itr fpeval objfeval convergence elapsed.time
1      em          NA 1500   1500        0       FALSE         0.17
2 squarem          NA   12     71        0        TRUE         0.02

Acceleration scheme 3 (pem), 4 (decme), 5 (qn) failed

> error(res3)

method 3 (pem): Error in accelerate(par = par, fixptfn = fixptfn, objfn = objfn, boundar
  argument "objfn" is missing, with no default

method 4 (decme): Error in accelerate(par = par, fixptfn = fixptfn, objfn = objfn, bound
  argument "objfn" is missing, with no default

method 5 (qn): Error in is.null(objfn) : 'objfn' is missing
```

In certain circumstances, quasi-Newton may produce invalid parameter values (e.g. values outside the parameter space). For example, if we use as a starting value a point near the boundary of the parameter space, quasi-Newton will produce an error:

```
> res4 <- turboem(par = c(0.9, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     boundary = boundary, y = y)
> res4

   method    value.objfn  itr fpeval objfeval convergence elapsed.time
1      em 1989.945868319 1500   1500        1       FALSE         0.17
2 squarem 1989.945859883   32     63       33        TRUE         0.02
3     pem 1989.945859883   24     58      185        TRUE         0.01
4   decme 1989.945859883   38     38      186        TRUE         0.05

Acceleration scheme 5 (qn) failed
```

Invalid parameter values at a particular iteration of quasi-Newton typically yields the following error message

```
> error(res4)

method 5 (qn): Error in if (l2 < lnew) { : missing value where TRUE/FALSE needed
```

One way to rectify this problem is to include the `pconstr` argument, which defines the bounds of the parameter space

```
> pconstr <- function(par) {
+     lower <- c(0, 0, 0)
+     upper <- c(1, Inf, Inf)
+     return(all(lower < par & par < upper))
+ }
> res5 <- turboem(par = c(0.9, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     boundary = boundary, y = y, pconstr = pconstr)
> res5

   method    value.objfn  itr fpeval objfeval convergence elapsed.time
1      em 1989.945868319 1500   1500        1       FALSE         0.18
2 squarem 1989.945859883   32     63       33        TRUE         0.00
3     pem 1989.945859883   24     58      185        TRUE         0.04
4   decme 1989.945859883   38     38      186        TRUE         0.04
5      qn 1989.945859883   29     33       57        TRUE         0.02
```

3.2 Alternative stopping criteria

Stopping criteria for each algorithm may be specified through the `control.run` argument. Default values of `control.run` are:

```
   convtype="parameter",
   tol=1.0e-07,
   stoptype="maxiter",
   maxiter=1500,
   maxtime=60,
   convfn.user=NULL,
   stopfn.user=NULL,
   trace=FALSE,
   keep.objfval=FALSE.
```

There are two ways the algorithm will terminate. Either the algorithm will terminate if convergence has been achieved, or the algorithm will terminate if convergence has not been achieved within a pre-specified maximum number of iterations or maximum running time (alternative stopping criterion). At each iteration for each acceleration scheme, both the convergence criterion and the alternative stopping criterion will be checked. The arguments `convtype`, `tol`, and `convfn.user` control the convergence criterion. The arguments `stoptype`, `maxiter`, `maxtime`, and `stopfn.user` control the alternative stopping criterion.

Two types of convergence criteria have been implemented, with an option for the user to define his/her own convergence criterion. If `convtype="parameter"` (the default setting), then the default convergence criterion is to terminate if at the first iteration n satisfying

$$\left\{\sum_{k=1}^{K}(p_k^{(n)} - p_k^{(n-1)})^2\right\}^{1/2} < \texttt{tol},$$

where $p_k^{(n)}$ denotes the kth element of the fixed-point value p at the nth iteration. For example, to set the convergence criterion to be $\left\{\sum_{k=1}^{K}(p_k^{(n)} - p_k^{(n-1)})^2\right\}^{1/2} < 10^{-10}$ we specify `control.run` as

```
> res6 <- turboem(par = c(0.5, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     method = c("em", "pem", "squarem"), y = y, control.run = list(tol = 1e-10))
> res6
```

```
   method    value.objfn  itr fpeval objfeval convergence elapsed.time
1      em 1989.945860141 1500   1500        1       FALSE         0.21
2     pem 1989.945859883  208    426      582        TRUE         0.07
3 squarem 1989.945859883   25     49       26        TRUE         0.02
```

By specifying `convtype="objfn"`, the default convergence criterion is to terminate at the first iteration n such that

$$\left|L(\texttt{par}_n) - L(\texttt{par}_{n-1})\right| < \texttt{tol}.$$

For example, to set the convergence criterion to be $|L(\texttt{par}_n) - L(\texttt{par}_{n-1})| < 10^{-10}$ we specify `control.run` as follows

```
> res7 <- turboem(par = c(0.5, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     method = c("em", "pem", "squarem"), y = y, control.run = list(tol = 1e-10,
+         convtype = "objfn"))
> res7
```

```
   method    value.objfn  itr fpeval objfeval convergence elapsed.time
1      em 1989.945860141 1500   1500     1501       FALSE         0.20
2     pem 1989.945859883   19     48      136        TRUE         0.03
3 squarem 1989.945859883   22     43       23        TRUE         0.00
```

If the user desires alternate convergence criteria, `convfn.user` may be specified as a function with inputs `new` and `old` that maps to a logical taking the value TRUE if convergence is achieved and the value FALSE if convergence is not achieved. For example, to set the convergence rule to be $\max\{|\texttt{par}_n - \texttt{par}_{n-1}|\} < 10^{-10}$, `control.run` may be specified as follows

```
> convfn.user <- function(old, new) {
+     max(abs(new - old)) < tol
+ }
> res8 <- turboem(par = c(0.5, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     method = c("em", "pem", "squarem"), y = y, control.run = list(tol = 1e-10,
+         convfn.user = convfn.user))
> res8
```

```
   method    value.objfn  itr fpeval objfeval convergence elapsed.time
1      em 1989.945860141 1500   1500        1       FALSE         0.16
2     pem 1989.945859883  173    356      503        TRUE         0.07
3 squarem 1989.945859883   25     49       26        TRUE         0.00
```

Similarly, to set the convergence rule to be

$$\frac{|L(\texttt{par}_n) - L(\texttt{par}_{n-1})|}{|L(\texttt{par}_{n-1})| + 1} < 10^{-8},$$

`control.run` may be specified as follows

```
> convfn.user <- function(old, new) {
+     abs(new - old)/(abs(old) + 1) < tol
+ }
> res9 <- turboem(par = c(0.5, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     method = c("em", "pem", "squarem"), y = y, control.run = list(tol = 1e-08,
+         convtype = "objfn", convfn.user = convfn.user))
> res9
```

```
   method    value.objfn itr fpeval objfeval convergence elapsed.time
1      em 1989.948186528 447    447      448        TRUE         0.06
2     pem 1989.946009291   7     24       49        TRUE         0.00
3 squarem 1989.945859912  16     31       17        TRUE         0.02
```

Two types of alternative stopping criteria have been implemented, with the option for the user to define his/her own stopping criterion. If `stoptype="maxiter"` (the default setting), then the algorithm will terminate if convergence has not been achieved within

`maxiter` iterations of the acceleration scheme. If `stoptype="maxtime"`, then the algorithm will terminate if convergence has not been achieved within `maxtime` seconds of running the acceleration scheme. Note that the running time of the acceleration scheme is calculated once every iteration. For example, the code

```
> res10 <- turboem(par = c(0.5, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     method = c("em", "pem", "squarem"), y = y, control.run = list(tol = 1e-15,
+         stoptype = "maxtime", maxtime = 10))
> res10

   method    value.objfn  itr fpeval objfeval convergence elapsed.time
1      em 1989.945859883 6318   6318        1        TRUE         0.75
2     pem 1989.945859883 1486   2982     3361        TRUE         0.44
3 squarem 1989.945859883   69    135       70        TRUE         0.03
```

imposes a strict convergence criterion, but allows the algorithms up to 10 seconds to run.

If the user desires different alternate stopping criteria than those implemented, `stopfn.user` may be specified as a function with no inputs that maps to a logical taking the value TRUE which leads to the algorithm being terminated or the value FALSE which leads to the algorithm proceeding as usual. For example, if the user desires the function to stop if either the maximum number of iterations is above 2000 or the maximum running time is above 0.2 seconds, this may be achieved by

```
> stopfn.user <- function() {
+     iter >= maxiter | elapsed.time >= maxtime
+ }
> res11 <- turboem(par = c(0.5, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     method = c("em", "pem", "squarem"), y = y, control.run = list(tol = 1e-15,
+         stopfn.user = stopfn.user, maxtime = 0.2, maxiter = 2000))
> res11

   method    value.objfn  itr fpeval objfeval convergence elapsed.time
1      em 1989.945859944 1666   1666        1       FALSE         0.20
2     pem 1989.945859883  634   1278     1514       FALSE         0.21
3 squarem 1989.945859883   69    135       70        TRUE         0.01
```

3.3 Changing default configurations of the acceleration algorithms

Instead of using the default implementations of each acceleration scheme, we may use alternative specifications. For example, we might like to compare higher-order SQUAREM algorithms (e.g. $K = 2$ or $K = 3$), consider different values for the qn parameter in the quasi-Newton class of schemes, or use a different version of the dynamic ECME scheme. In the next code chunk, we compare the EM algorithm to the following accelerators: SQUAREM with $K = 2$ and $K = 3$, Dynamic ECME versions 2 and 2s, quasi-Newton with $qn = 1$ and $qn = 2$, and Parabolic EM versions "arithmetic" as well as the default "geometric".

```
> res12 <- turboem(par = c(0.9, 1, 3), fixptfn = fixptfn, objfn = objfn,
+     boundary = boundary, pconstr = pconstr, method = c("em",
+         "squarem", "squarem", "decme", "decme", "qn", "qn", "pem",
+         "pem"), control.method = list(list(), list(K = 2), list(K = 3),
+         list(version = 2), list(version = "2s"), list(qn = 1),
+         list(qn = 2), list(version = "arithmetic"), list(version = "geometric")),
+     y = y)
> res12
```

	method	value.objfn	itr	fpeval	objfeval	convergence	elapsed.time
1	em	1989.945868319	1500	1500	1	FALSE	0.18
2	squarem	1989.945859883	12	61	13	TRUE	0.01
3	squarem	1989.945859883	9	64	10	TRUE	0.02
4	decme	1989.945859883	27	27	699	TRUE	0.03
5	decme	1989.945859883	38	38	186	TRUE	0.05
6	qn	1989.945859883	100	103	199	TRUE	0.04
7	qn	1989.945859883	29	33	57	TRUE	0.02
8	pem	1989.945859883	27	64	491	TRUE	0.06
9	pem	1989.945859883	24	58	185	TRUE	0.02

4 Benchmark study illustration

We conduct a small benchmark study to compare EM accelerators for this Poisson mixture example in order to illustrate the benchmark visualization tools implemented in the R package.

For each of $r = 1, \ldots,$ `NREP` repetitions, we randomly simulate a starting value $\texttt{par}^{(r)}$. We then apply each of the EM accelerators, beginning at that starting value.

Control parameters

Because each of the acceleration schemes has different versions and control parameters, we create a list containing the control parameters for each scheme we will consider

```
> method.names <- c("EM", "squaremK1", "squaremK2", "parabolicEM",
+     "dynamicECME", "quasiNewton")
> nmethods <- length(method.names)
> method <- c("em", "squarem", "squarem", "pem", "decme", "qn")
> control.method <- vector("list", nmethods)
> names(control.method) <- method.names
> control.method[["EM"]] <- list()
> control.method[["squaremK1"]] <- list(K = 1)
> control.method[["squaremK2"]] <- list(K = 2)
> control.method[["parabolicEM"]] <- list(version = "geometric")
> control.method[["dynamicECME"]] <- list(version = "v2")
> control.method[["quasiNewton"]] <- list(qn = 2)
```

We also set the control parameters for stopping the algorithm, including the convergence rule and an alternative stopping rule (setting the maximum runtime or maximum number of iterations).

```
> control.run <- list(tol = 1e-07, stoptype = "maxtime", maxtime = 2,
+     convtype = "parameter")
```

Starting values

We generate the starting values $\texttt{par}^{(r)}, r = 1, \ldots, \texttt{NREP}$. To ensure reproducibility, we set the seed.

```
> NREP <- 100
> library(setRNG)
> test.rng <- list(kind = "Mersenne-Twister", normal.kind = "Inversion",
+     seed = 1)
> setRNG(test.rng)
> starting.values <- cbind(runif(NREP), runif(NREP, 0, 4), runif(NREP,
+     0, 4))
> head(starting.values, 3)

                [,1]           [,2]            [,3]
[1,] 0.2655086631421 2.618895712309 1.0700328294188
[2,] 0.3721238996368 1.412789087743 0.8745811395347
[3,] 0.5728533633519 1.081040583551 2.0671873455867
```

Execute simulation

Now we run the simulation. For each acceleration scheme, we will keep track of whether the algorithm produced an error, whether the algorithm converged, the total number of iterations until convergence, the execution time, and the final value of the negative log-likelihood function.

```
> runtime <- niter <- negloglik <- conv <- fail <- errors <- matrix(NA,
+     NREP, nmethods)
> options(digits = 13)
> tot.time1 <- Sys.time()
> for (i in 1:NREP) {
+     p.start <- starting.values[i, ]
+     res <- try(turboem(par = p.start, fixptfn = fixptfn, objfn = objfn,
+         y = y, method = method, boundary = boundary, pconstr = pconstr,
+         control.method = control.method, control.run = control.run))
+     if (class(res) == "try-error") {
+         fail[i, ] <- TRUE
+     }
+     else {
```

```
+         runtime[i, ] <- res$runtime[, "elapsed"]
+         negloglik[i, ] <- res$value.objfn
+         conv[i, ] <- res$convergence
+         niter[i, ] <- res$itr
+         fail[i, ] <- res$fail
+         errors[i, ] <- res$errors
+     }
+ }
> tot.time2 <- Sys.time()
```

The total simulation running time for the $NREP = 100$ iterations is

```
> difftime(tot.time2, tot.time1)

Time difference of 36.39000010490 secs
```

Results

We create an object of class `turbosim` so that we can summarize and visualize the results of our simulation.

```
> results <- list(method.names = method.names, fail = fail, conv = conv,
+     value.objfn = negloglik, runtime = runtime, method = method,
+     control.method = control.method, control.run = control.run)
> class(results) <- "turbosim"
```

An object of class `turbosim` is a list containing at least the following components:

- `method.names`: A vector of unique identifiers for the algorithms being compared
- `fail`: A matrix whose $(i, j)-$element is a logical (TRUE/FALSE) for whether the jth algorithm at the ith simulation iteration failed (produced an error)
- `conv`: A matrix whose $(i, j)-$element is a logical (TRUE/FALSE) for whether the jth algorithm at the ith simulation iteration satisfied the convergence criterion before termination
- `value.objfn`: A matrix whose $(i, j)-$element is the value of the objective function of the jth algorithm at the ith simulation iteration
- `runtime`: A matrix whose $(i, j)-$element is the running time of the jth algorithm at the ith simulation iteration

This list will also contain the components `method`, `control.method`, and `control.run` which were provided as arguments for `turboem` in the main function call of the simulation.

There are four methods associated with the class `turbosim`, namely `summary`, `boxplot`, `dataprof`, and `pairs`. We next illustrate the use of each method for our simulation.

The method `summary` prints a table of the number of failures across acceleration schemes. We consider three types of failures. The first occurs when the algorithm produces an error message. The second is if the algorithm does not converge before the alternative stopping rule is satisfied (e.g. the maximum number of iterations or maximum prespecified runtime is

achieved). The third type of failure occurs if the algorithm claims convergence but the value of the objective function is "far" from the best achievable value. To assess this third type of failure, we determine whether the objective function value acheived by the algorithm is close (within a pre-specified value, `eps`) to the smallest value achieved across all algorithms at that iteration. A summary of the failures across iterations by algorithm is given by the method `summary`.

```
> summary(results, eps = 0.01)
```

```
           Algorithm failed  Exceeded 0.03 min.  objfn > min(objfn) + 0.01
EM                         0                   0                         0
squaremK1                  0                   0                         0
squaremK2                  0                   0                        21
parabolicEM                0                   0                         0
dynamicECME                0                   0                         0
quasiNewton                0                   0                         0
```

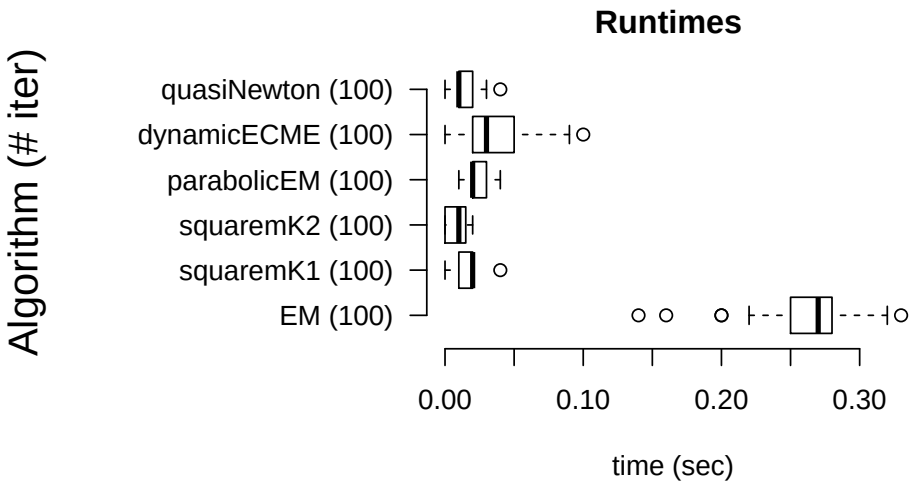
Alternatively, say we knew somehow that the global minimum of the objective function for this problem were `sol` = 1989.945859883. In this case the third type of failure would occur of the objective function value acheived by the algorithm is more than `eps` units greater than `sol`, and we could summarize the failures using

```
> summary(results, eps = 0.01, sol = 1989.945859883)
```

```
           Algorithm failed  Exceeded 0.03 min.  objfn > min(objfn) + 0.01
EM                         0                   0                         0
squaremK1                  0                   0                         0
squaremK2                  0                   0                        21
parabolicEM                0                   0                         0
dynamicECME                0                   0                         0
quasiNewton                0                   0                         0
```

The `boxplot` method shows boxplots of the running time across simulation iterations for each acceleration scheme. We can specify the argument `whichfail` which identifies which of the simulation iterations were failures. These iterations will be excluded in the boxplots. For example, if we wish to plot the iterations for which the algorithms did not produce an error, we would use the command

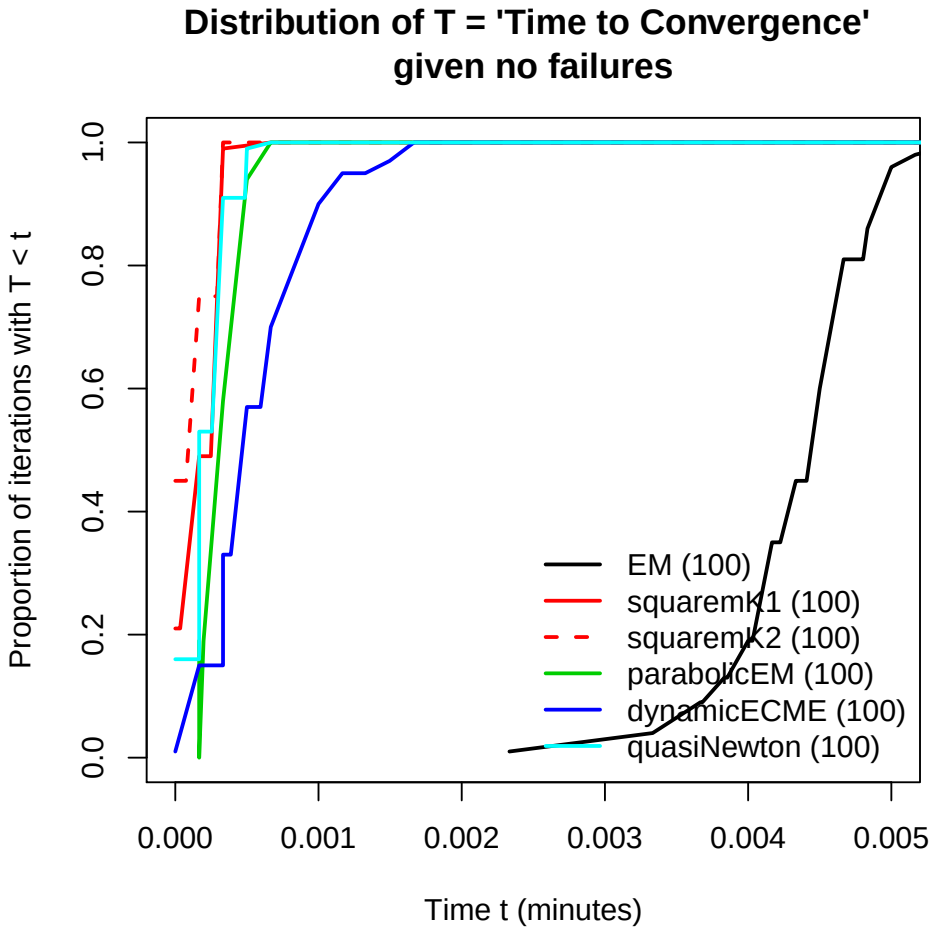
```
> boxplot(results, whichfail = results$fail)
```



The default setting for `whichfail` in `boxplot`, as in the other methods for the `turbosim` class, excludes those simulation iterations for which either the algorithm produced an error or convergence was not achieved.

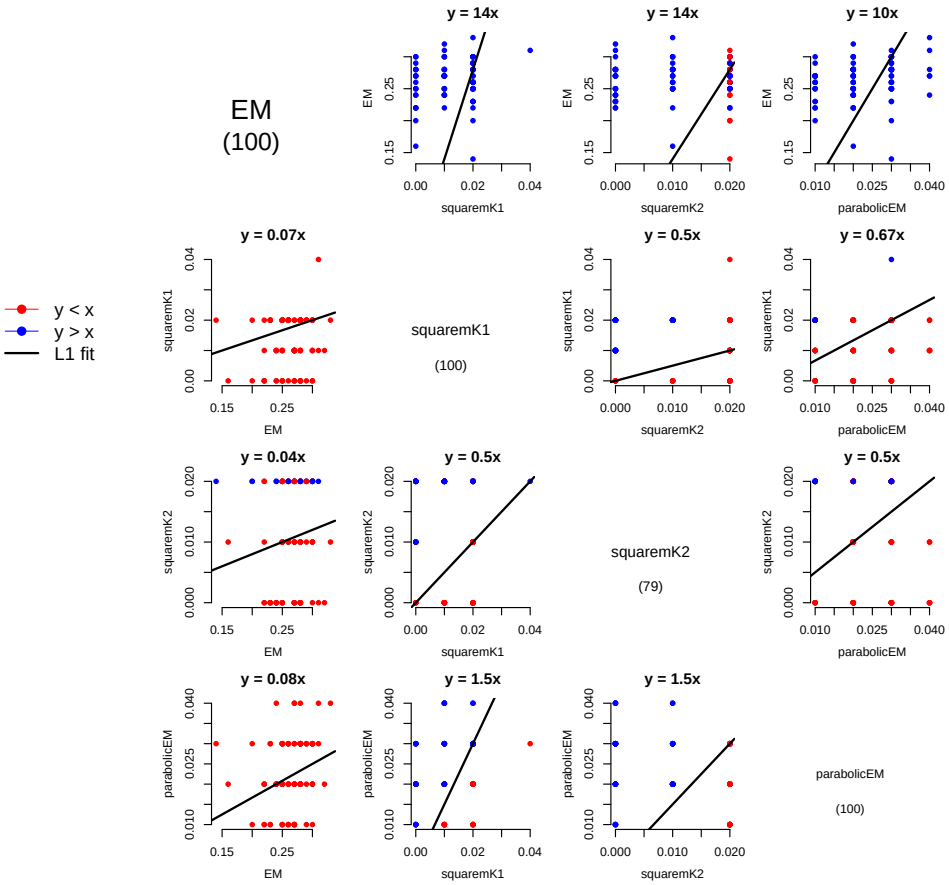
The `dataprof` method shows the estimated distribution function of the time until convergence (T) for each acceleration scheme. We set $T_{i,j} = \infty$ for those iterations i where algorithm j failed, where failures are specified using the `whichfail` argument of `dataprof()`.

```
> dataprof(results, xlim = c(0, 0.005))
```



Finally, to visualize pairwise comparisons of the running time across algorithms at each iteration, we implement the `pairs` method which displays a scatterplot matrix of the run times. For this method, as with the other methods, we can specify which of the algorithms will be shown in the results by specifying `which.methods`.

```
> pairs(results, which.methods = 1:4, cex = 0.8, whichfail = with(results,
+     fail | !conv | value.objfn > apply(value.objfn, 1, min) +
+         0.01))
```



Rather than ignore points where one of the pair of algorithms failed, we plot those points along the far right or topmost part of the plot. For example, for those iterations where `squaremK2` failed, we set the running time for those iterations to the maximum running time of `squaremK2` across iterations, and we color-coded the point as having a greater running time as compared to the algorithm that did not fail. The scatterplots also include the robust linear regression fit (using the L1 norm) constrained so that the intercept is 0.

References

[1] Varadhan and Roland (2008). Simple and Globally Convergent Methods for Accelerating the Convergence of Any EM Algorithm. Scand J Stat. 35 (2) 335-3531

[2] Berlinet and Roland (2009). Parabolic acceleration of the EM algorithm. Stat Comput. 19 (1) 35-47

[3] Zhou et al (2011). A quasi-Newton acceleration for high-dimensional optimization algorithms. Stat Comput. 21 (2) 261-273

[4] He and Liu (2010) The Dynamic ECME Algorithm. Technical Report. arXiv:1004.0524v1