`turboEM`: A Suite of Convergence Acceleration Schemes for EM and MM algorithms

Jennifer F. Bobb

The R package `turboEM` implements four methods to accelerate EM and MM algorithms: SQUAREM [1], Parabolic EM [2], a quasi-Newton algorithm [3], and Dynamic ECME [4].

In the first part of this document, we illustrate how to use `turboEM` to apply the acceleration schemes through an extended example. We show how to (i) apply each state-of-the-art accelerator using a single function call, (ii) compare the algorithms' solutions and compute standard errors, (iii) specify different convergence criteria and stopping rules, and (iv) run the acceleration schemes in parallel in order to make computation fast and efficient.

In the second part, we illustrate how `turboEM` may be used as a tool for conducting benchmark studies to critically compare the acceleration schemes. We show how (i) a benchmark study can be run using a simple function call, (ii) the study can be made more efficient through parallelization, and (iii) how to apply simple and sophisticated metrics for summarizing and visualizing the benchmark study results.

1 Poisson mixture distribution

First, load the `turboEM` package into R.

```
> library(turboEM)
```

You can get a brief overview of the main function `turboem` and the associated methods by typing

```
> help(package="turboEM")
```

1.1 Example data

Let's consider a simple example of speeding up the EM algorithm for estimating parameters of a mixture of two Poisson distributions. Here are data from Hasselblad (1969).

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

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)
+         denom <- p[1]*exp(-p[2])*p[2]^i + (1 - p[1])*exp(-p[3])*p[3]^i
+         zi <- p[1]*exp(-p[2])*p[2]^i / denom
+         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) for the Poisson mixture is given by

```
> 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) )
+ }
```

1.2 Illustration of basic features of `turboem`

First, let's use `turboem` to fit the EM algorithm as well as the acceleration schemes SQUAREM and Parabolic EM, 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	0.17
2	squarem	1989.945859883	23	45	24	TRUE	0.02
3	pem	1989.945859883	25	60	176	TRUE	0.02

For this problem with this starting guess for the parameter values, the EM algorithm does not achieve convergence at the default tolerance within the allotted 1500 iterations. On the other hand, both Parabolic EM and SQUAREM do converge. We'll talk more about convergence issues, including how to use different convergence rules (or even specify your own) later on. Even in this simple example, the accelerator algorithms provide a substantial speed-up.

The `turboem` function outputs an object of class `turbo`. Different methods for handling the output are available, which we will now explore. Let's first look at the parameter values obtained across the three algorithms using the `pars` method.

```
> 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 might be interested in exploring the algorithms' histories by plotting the objective function values over time. Because the default settings of the algorithms do not keep the objective function values at each iteration (and because not all algorithms require an objective function to be provided), we must specify that we would like `turboem` to track these values over time using the `keep.objfval` argument of the control parameters.

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


   method value.objfn  itr fpeval objfeval convergence elapsed.time
1      em   1989.946 1500   1500     1501       FALSE         0.18
2 squarem   1989.946   23     45       24        TRUE         0.00
3     pem   1989.946   25     60      176        TRUE         0.02


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

Trace of Objective Function Value
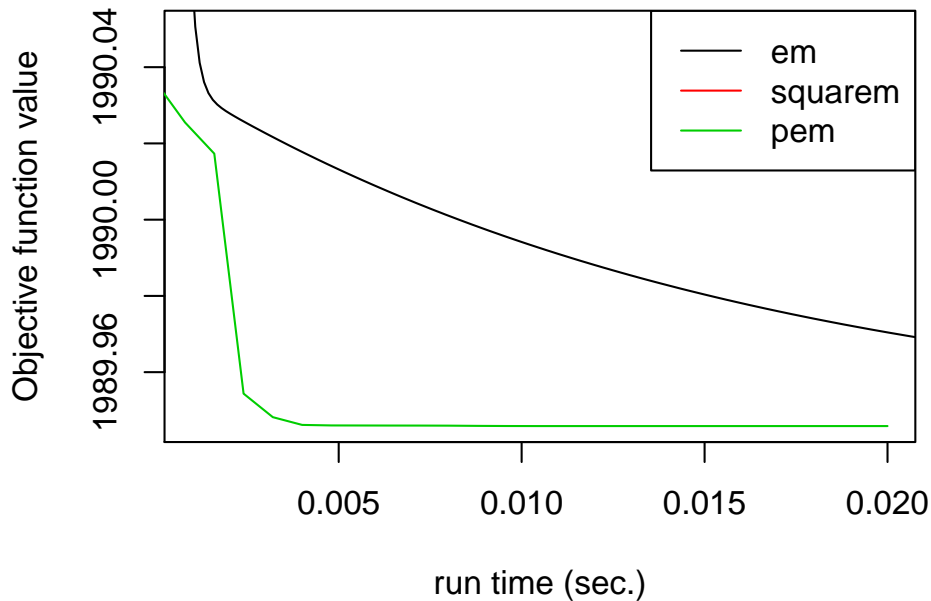
Up until this point, we have not considered the Dynamic ECME acceleration scheme. Dynamic ECME requires an additional input in order to run. For Dynamic ECME, we must specify the subspace over which line searches will be conducted, which is done through a `boundary` function. For this example, the function defining the subspace for a given parameter value `par` and a given search direction `dr` 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 may now use `turboem` for the Dynamic ECME algorithm.

```
> 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	157	TRUE	0.01

5

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 given.

```
> 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         0.14
2 squarem         NA   12     71        0        TRUE         0.01

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

If we did not know the reason certain algorithms failed, we can call the `error` method to find out.

```
> error(res3)

method 3 (pem): Error in accelerate(par = par, fixptfn = fixptfn, objfn = objfn, boundar

method 4 (decme): Error in accelerate(par = par, fixptfn = fixptfn, objfn = objfn, bound

method 5 (qn): Error in accelerate(par = par, fixptfn = fixptfn, objfn = objfn, boundary
```

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         0.13
2 squarem 1989.945859883   32     63       33        TRUE         0.01
3     pem 1989.945859883   24     58      185        TRUE         0.01
4   decme 1989.945859883   38     38      187        TRUE         0.02

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	0.14
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	187	TRUE	0.03
5	qn	1989.945859883	29	33	57	TRUE	0.00

1.3   Convergence criteria and alternative stopping rules

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 rule). At each iteration for each acceleration scheme, both the convergence criterion and the alternative stopping rule 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 rule.

1.3.1 Convergence criteria

Two types of convergence criteria have been implemented, as well as an option for a user-defined criterion. If `convtype = "parameter"` (the default setting), then the default convergence criterion is to terminate 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 use this convergence criterion with a tolerance of 10^{-10}, specify the `control.run` argument as

```
> res6 <- turboem(par=c(0.5, 1, 3), fixptfn=fixptfn, objfn=objfn,
+                 method=c("em", "pem", "squarem"), y=y,
+                 control.run=list(tol=1.0e-10))
> res6
```

	method	value.objfn	itr	fpeval	objfeval	convergence	elapsed.time
1	em	1989.945860141	1500	1500	1	FALSE	0.12
2	pem	1989.945859883	208	426	582	TRUE	0.10
3	squarem	1989.945859883	25	49	26	TRUE	0.01

To use a convergence criterion based on the objective function value at each iteration, you can specify `convtype = "objfn"`. Then the algorithm will terminate at the first iteration n such that

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

Here we use this convergence criterion with a tolerance of 10^{-10}:

```
> res7 <- turboem(par=c(0.5, 1, 3), fixptfn=fixptfn, objfn=objfn,
+                 method=c("em", "pem", "squarem"), y=y,
+                 control.run=list(tol=1.0e-10, convtype="objfn"))
> res7
```

	method	value.objfn	itr	fpeval	objfeval	convergence	elapsed.time
1	em	1989.945860141	1500	1500	1501	FALSE	0.18
2	pem	1989.945859883	19	48	136	TRUE	0.01
3	squarem	1989.945859883	22	43	23	TRUE	0.00

If you would like to use a different convergence criterion than these two options, you can define your own. To do this, define the `convfn.user` argument as a function with inputs `new` and `old` that maps to `TRUE` if convergence is achieved and maps to `FALSE` otherwise. For example, for convergence at the first iteration n where $\max\{\left|\text{par}_n - \text{par}_{n-1}\right|\} < 10^{-10}$, you may specify `control.run` as

```
> 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=1.0e-10, convfn.user = convfn.user))
> res8
```

	method	value.objfn	itr	fpeval	objfeval	convergence	elapsed.time
1	em	1989.945860141	1500	1500	1	FALSE	0.12
2	pem	1989.945859883	173	356	503	TRUE	0.05
3	squarem	1989.945859883	25	49	26	TRUE	0.02

Note that here, because we did not specify the `convtype` argument, `turboem` uses the default
option of parameter-based convergence. In other words, `turboem` assumes that the `old` and
`new` arguments of `convfn.user` refer to the parameter values par_{n-1} and par_n, respectively.

For another example, if you would like to set the convergence criterion to be

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

then the `convfn.user` argument of `control.run` may be specified as follows

```
> convfn.user.objfn <- 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=1.0e-8, convtype="objfn",
+              convfn.user = convfn.user.objfn))
> res9
```

	method	value.objfn	itr	fpeval	objfeval	convergence	elapsed.time
1	em	1989.948186528	447	447	448	TRUE	0.05
2	pem	1989.946009291	7	24	49	TRUE	0.02
3	squarem	1989.945859912	16	31	17	TRUE	0.00

1.3.2 Alternative stopping rules

Two types of alternative stopping rule have been implemented, as well as an option for a
user-defined rule. 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 you set `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=1.0e-15, stoptype="maxtime",
+               maxtime=10))
> res10
```

	method	value.objfn	itr	fpeval	objfeval	convergence	elapsed.time
1	em	1989.945859883	6318	6318	1	TRUE	0.54
2	pem	1989.945859883	1486	2982	3361	TRUE	0.38
3	squarem	1989.945859883	69	135	70	TRUE	0.05

imposes a strict tolerance for convergence, but it allows each algorithm up to 10 seconds to run.

If you would like a different stopping rule than these, you may specify the `stopfn.user` argument of `control.run`. To do this, define `stopfn.user` as a function with no inputs that maps to `TRUE` when the algorithm should be terminated and maps to `FALSE` otherwise. For example, if you would like the algorithm to stop when either the number of iterations reaches 2000 or the running time exceeds 0.2 seconds, you can specify

```
> 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=1.0e-15, stopfn.user=stopfn.user,
+               maxtime=0.2, maxiter=2000))
> res11
```

	method	value.objfn	itr	fpeval	objfeval	convergence	elapsed.time
1	em	1989.945859886	2000	2000	1	FALSE	0.19
2	pem	1989.945859883	719	1448	1702	FALSE	0.20
3	squarem	1989.945859883	69	135	70	TRUE	0.03

1.4   Changing default configurations of acceleration schemes

Each of the general acceleration schemes (SQUAREM, Parabolic EM, Dynamic ECME, and Quasi-Newton) has different variants and choices for various tuning parameters. For example, we might wish 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. It's very easy to change the algorithms' default specifications in `turboem`.

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". To do this, we will utilize the `control.method` argument.

```
> 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         0.14
2  squarem 1989.945859883   12     61       13        TRUE         0.00
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      187        TRUE         0.03
6       qn 1989.945859883  100    103      199        TRUE         0.03
7       qn 1989.945859883   29     33       57        TRUE         0.00
8      pem 1989.945859883   27     64      491        TRUE         0.03
9      pem 1989.945859883   24     58      185        TRUE         0.03
```

1.5   Parallelization of `turboem`

Up until this point, when we ran `turboem`, each of the accelerations schemes were run sequentially. If you have access to multiple cores within a computer or multiple computers, you may wish to run the accelerators in parallel. Parallelization has been implemented in `turboEM` through the foreach package.

There are two steps to running the algorithms in parallel

1. Register a *parallel backend.*
2. Set the argument `parallel = TRUE` in `turboem`.

The parallel backend is the method of parallelization, and which parallel backend you use will depend on your computing environment. Some of the parallel backends available include

- **doMC**: Based on the `multicore` package, this backend uses multiple cores on a single machine. It is currently supported by Mac or Unix/Linux operating systems.
- **doSNOW**: Based on the `snow` (Simple Network of Workstations) package, this backend allows multiple computers and multiple cores within a computer. It is supported on Mac, Unix/Linux, and Windows machines.
- **doMPI**: Based on the `Rmpi` package, this method works on clusters of computers with Message Passing Interface (MPI) installed.

In addition to looking at the vignettes for each of the backends, another useful overview for parallel computing with `foreach` can be found here.

As an example, here we run `turboem` using the `doSNOW` backend. First we register the backend:

```
> library(doSNOW)
> cl <- makeSOCKcluster(2)
> registerDoSNOW(cl)
```

Now we run `turboem`.

```
> time.parallel <- system.time(res.parallel <-
+         turboem(par=c(0.9, 1, 6), fixptfn=fixptfn, objfn=objfn,
+                 method=c("em", "pem", "squarem"), y=y, parallel=TRUE,
+                 control.run=list(tol=1.0e-14, stoptype="maxtime",
+                 maxtime=10)))
> res.parallel
```

```
   method     value.objfn  itr fpeval objfeval convergence elapsed.time
1      em 1989.945859883 6262   6262        1        TRUE         0.60
2     pem 1989.945859883 1445   2900     3284        TRUE         0.40
3 squarem 1989.945859883   76    151       77        TRUE         0.05
```

```
> time.parallel
```

```
  user  system elapsed
  0.00    0.00    1.17
```

Then for `doSNOW` we must stop the cluster:

```
> stopCluster(cl)
```

This computer has 2 cores available for use (although here only 3 of the cores were used–one for each acceleration scheme).

We can compare the computation time to running the algorithms sequentially.

```
> time.sequential <- system.time(res.sequential <-
+          turboem(par=c(0.9, 1, 6), fixptfn=fixptfn, objfn=objfn,
+                  method=c("em", "pem", "squarem"), y=y, parallel=FALSE,
+                  control.run=list(tol=1.0e-14, stoptype="maxtime",
+                  maxtime=10)))
> res.sequential


   method      value.objfn  itr fpeval objfeval convergence elapsed.time
1      em 1989.945859883 6262   6262        1        TRUE          0.63
2     pem 1989.945859883 1445   2900     3284        TRUE          0.36
3 squarem 1989.945859883   76    151       77        TRUE          0.04


> time.sequential


   user  system elapsed
   1.00    0.00    1.04
```

While each of the individual algorithms took longer to run in parallel due to the overhead of communicating across multiple cores, running the algorithms in parallel led to an overall speed-up of 0.89.

For the small example we have considered here, the potential gain in running the schemes in parallel is not that great, since the algorithms do not take very long to run in the first place. More complex examples will yield much greater gains. See, for example, the results of a benchmark study in Section (2.3) that demonstrates the value of parallel execution.

Note that if your goal is to compare computation times of various acceleration schemes, you probably should not use the option `parallel = TRUE` in `turboem`. If some of the computers/processors over which the work is split are less powerful than others, any difference in computation times could be due to computing power rather than to differences among algorithms. If you'd like to compare the algorithms' performance, `turboEM` provides several useful tools for conducting benchmark studies. We'll show how they work in the next section.

2 Conducting benchmark studies

Let's use `turboEM` to conduct a small benchmark study to compare EM accelerators for our Poisson mixture example.

For each of $r = 1, \ldots,$ `NREP` repetitions, we will randomly simulate a starting value $\text{par}^{(r)}$. Then we'll apply each of the EM accelerators, beginning at that starting value, and we'll compare results across repetitions using the summary and visualization tools implemented in `turboEM`.

13

2.1 Control parameters

Because each of the acceleration schemes has different variants and control parameters, we'll first create a list containing the control parameters for each of the schemes we'll be considering.

```
> 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="2s")
> control.method[["quasiNewton"]] <- list(qn=2)
```

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

```
> control.run <- list(tol=1e-7, stoptype="maxtime", maxtime=2,
+                     convtype="parameter")
```

2.2 Starting values

Now, let's generate the starting values $\mathbf{par}^{(r)}, r = 1, \ldots, \texttt{NREP}$. If we set the seed prior to generating the starting values, then our benchmark study results can be reproduced.

```
> 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
```

14

2.3   Execute benchmark study using simple call

The `turboSim` function may be used to run the benchmark study.

```
> simtime <- system.time(
+       results <- turboSim(parmat=starting.values, fixptfn=fixptfn,
+                       objfn=objfn, method=method, boundary=boundary,
+                       pconstr=pconstr, method.names=method.names,
+                       y=y, control.method=control.method,
+                       control.run=control.run)
+                           )
> simtime

   user  system elapsed
  28.45    0.16   29.02
```

Note that all of the inputs to `turboSim()` are identical to those in the `turboem` function, except `parmat` is a matrix of starting parameter values, where each row corresponds to a single simulation iteration, and `method.names` is a new argument containing the unique names that can identify the methods being compared.

There is also the ability to run the benchmark study in parallel over multiple cores or computers, with parallelization implemented using the `foreach` package that we talked about earlier (Section 1.5). It is important that all of the algorithms are run on the same processor for a given repetition, in case some of the processors/computers are less powerful than others. Therefore, in `turboSim()` we parallelize across simulation repetitions, rather than across acceleration schemes as in `turboem()`. As above, let's use the `doSNOW` parallel backend in order to compare the total computation time of our benchmark study when multiple cores are used. Let's see how much faster we can get.

```
> cl <- makeSOCKcluster(2)
> simtime.par <- system.time(
+       results.par <- turboSim(parmat=starting.values, fixptfn=fixptfn,
+                       objfn=objfn, method=method, boundary=boundary,
+                       pconstr=pconstr, method.names=method.names,
+                       y=y, control.method=control.method,
+                       control.run=control.run, parallel=TRUE)
+                           )
> simtime.par

   user  system elapsed
   0.22    0.00   15.49

> stopCluster(cl)
```

Here, running the algorithm in parallel over 2 cores yielded a speed-up of 1.87.

2.4   Results

The `turboSim` function produces an object of class `turbosim`. Let's now explore the different methods that will help us summarize and visualize the results of our benchmark study.

```
> class(results)


[1] "turbosim"


> results


Benchmark study over 100 repetitions.

Methods:
1. EM
2. squaremK1
3. squaremK2
4. parabolicEM
5. dynamicECME
6. quasiNewton

Functions to summarize and visualize results:
    summary(), boxplot(), dataprof(), pairs()
```

The method `summary` prints a table of the number of failures across acceleration schemes. Three types of failures are considered.

1. An error message is produced by the algorithm.

2. The algorithm does not converge prior to the alternative stopping rule (maximum number of iterations or running time) being reached.

3. The convergence criterion has been satisfied but the value of the objective function is "far" from the best achievable value.

To assess the third type of failure, we determine whether the objective function value achieved by the algorithm is close (within a pre-specified value, `eps`) to the smallest value achieved across all algorithms at that iteration. Let's look at the types of failures encountered by the algorithms for our study.

```
> 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. Then we could define the third type of failure as occurring when the objective function value achieved 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. To exclude results from the iterations where there were failures, you can use the `whichfail` argument. For example, we can exclude the 21 iterations for which `squaremK2` did not achieve an objective function close to the best possible value at that iteration.

```
> fails <- with(results, fail | !convergence |
+               value.objfn > apply(value.objfn, 1, min) + 0.01)
> boxplot(results, whichfail=fails)
```

Runtimes
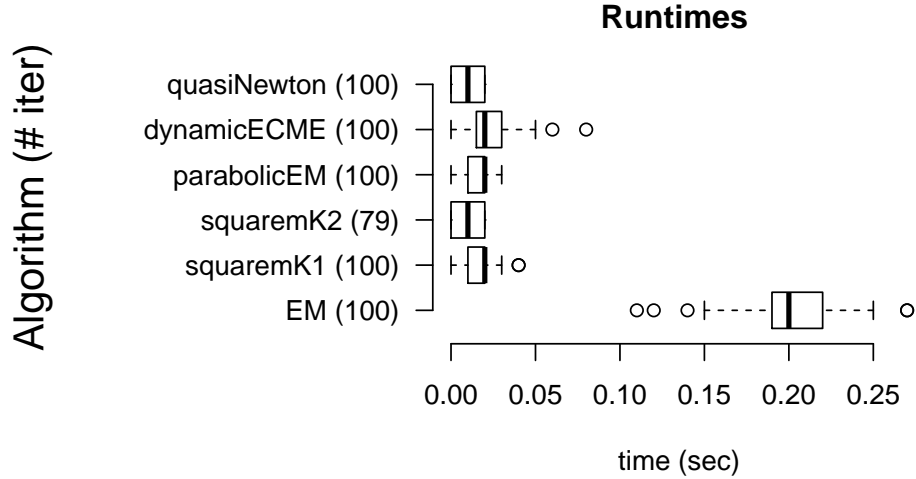
The default setting for `whichfail` in `boxplot`, as in the other methods for the `turbosim` class, excludes those simulation iterations for which either the algorithm produced an error or convergence was not achieved (failure types 1 and 2).

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)
```
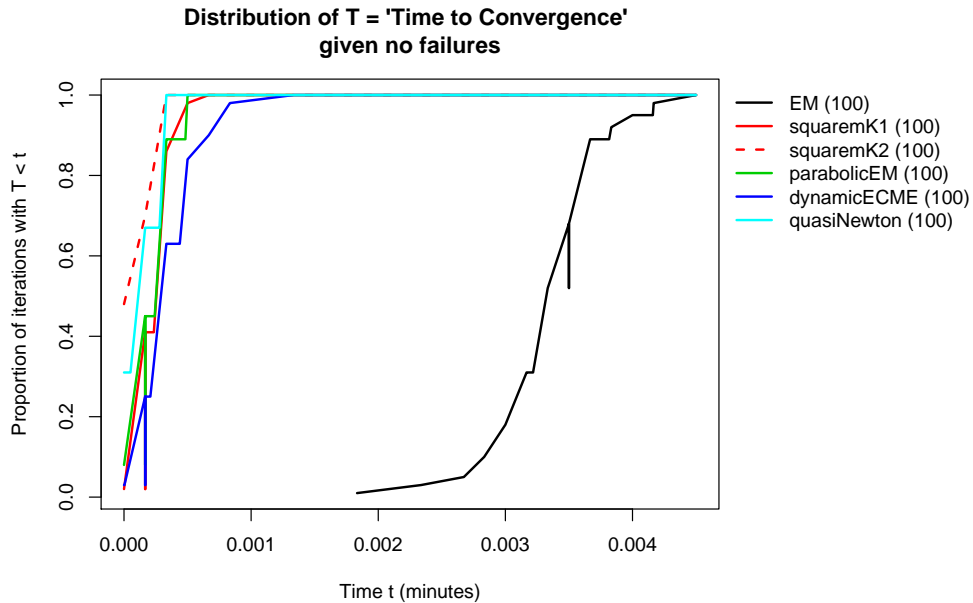

Finally, to visualize pairwise comparisons of the running time across algorithms at each iteration, we implement the `pairs` method which displays a scatterplot matrix of the run

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=fails)
```
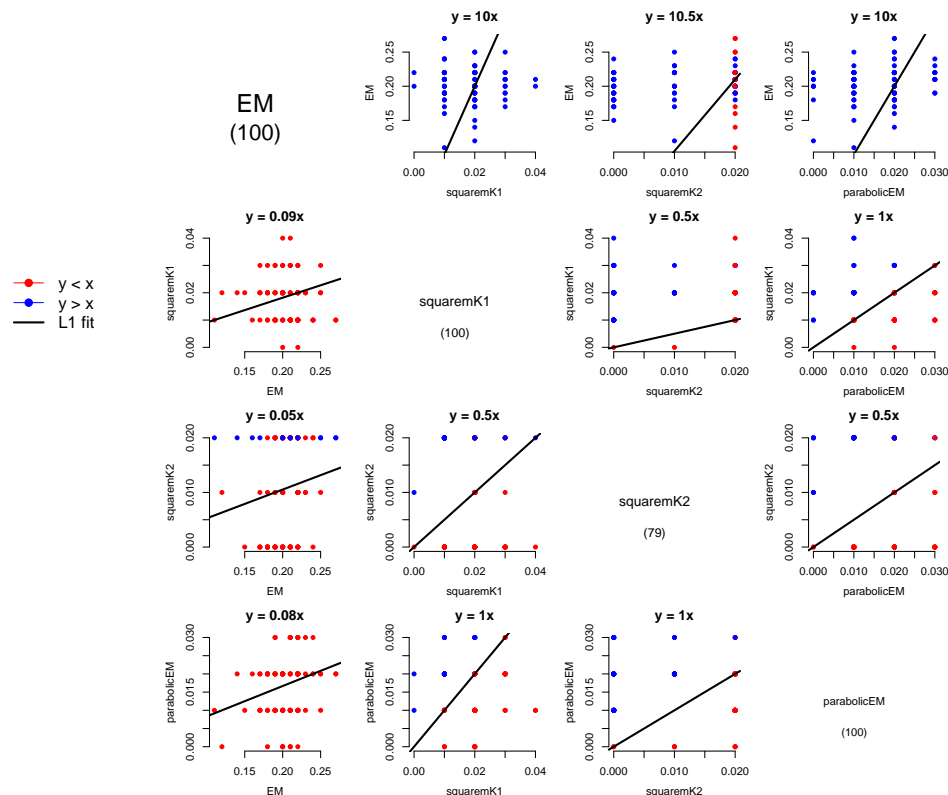

Rather than ignore points where one of the pair of algorithms failed, we plot those points along the far right or topmost part of the plot. For example, for those iterations where `squaremK2` failed, we set the running time for those iterations to the maximum running time of `squaremK2` across iterations, 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.

3 Conclusion

The `turboEM` package provides a unified implementation of acceleration schemes, which can be used *off-the-shelf* for any EM or MM problem. Here we have explored a small example to give you an overview of the different features of `turboEM`. You can specify one of the implemented convergence criteria and alternative stopping rules or you can define your own. You can run the algorithms in parallel to speed up computation time, and the parallel implementation works over a wide range of computing environments with little modification. Several methods are provided to allow output to be examined, displayed and summarized.

In addition, you can systematically compare the performance of the acceleration schemes by conducting a benchmark study, which can also be run in parallel. Finally, results from benchmark studies can be explored and presented through a suite of visualization methods. We hope that this package will enable researchers and applied scientists to easily use state-of-the-art EM accelerators and to critically evaluate the relative performances of the approaches across a wide range of optimization problems.

References

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

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

[3] Zhou H, Alexander D, and Lange K (2011). A quasi-Newton acceleration for high-dimensional optimization algorithms. *Stat Comput.* 21 (2) 261-273

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

[5] Hasselblad V (1969). Estimation of finite mixutres of distributions from the exponential family. *Journal of the American Statistical Association.* 64, 1459–1471.