Journal of Statistical Software

February 2020, Volume 92, Issue 7. *doi: 10.18637/jss.v092.i07*

SQUAREM: An R Package for Off-the-Shelf Acceleration of EM, MM and Other EM-Like Monotone Algorithms

Yu Du
Johns Hopkins University

Ravi Varadhan
Johns Hopkins University

Abstract

We discuss the R package **SQUAREM** for accelerating iterative algorithms which exhibit slow, monotone convergence. These include the well-known expectation-maximization algorithm, majorize-minimize (MM), and other EM-like algorithms such as expectation conditional maximization, and generalized EM algorithms. We demonstrate the simplicity, generality, and power of **SQUAREM** through a wide array of applications of EM/MM problems, including binary Poisson mixture, factor analysis, interval censoring, genetics admixture, and logistic regression maximum likelihood estimation (an MM problem). We show that **SQUAREM** is easy to apply, and can accelerate any fixed-point, smooth, contraction mapping with linear convergence rate. The squared iterative scheme (SQUAREM) algorithm provides significant speed-up of EM-like algorithms. The margin of the advantage for SQUAREM is especially huge for high-dimensional problems or when the EM step is relatively time-consuming to evaluate. SQUAREM can be used off-the-shelf since there is no need for the user to tweak any control parameters to optimize performance. Given its remarkable ease of use, SQUAREM may be considered as a default accelerator for slowly converging EM-like algorithms. All the comparisons of CPU computing time in the paper are made on a quad-core 2.3 GHz Intel Core i7 Mac computer. R package **SQUAREM** is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=SQUAREM/`.

Keywords: EM algorithm, fixed-point iteration, monotone convergence, convergence acceleration, optimization, high dimensional models, extrapolation methods.

1. Introduction

The R package **SQUAREM** provides convergence acceleration techniques for speeding-up slow, monotone iterative algorithms. These include the well-known expectation-maximization (EM) algorithm (Dempster, Laird, and Rubin 1977), majorize-minimize (MM; Lange, Hunter, and

Yang 2000), and other algorithmic variants such as expectation-conditional maximization (ECM; Meng and Rubin 1993), expectation-conditional maximization or either (ECME; Liu and Rubin 1998), among others. Dempster *et al.* (1977) refer to such variants as "generalized EM (GEM)" when the M step is only partially implemented. In this paper, we term these "EM-like algorithms", because they all have a contractive fixed-point mapping with linear rate of convergence, like EM. For the definition of linear rate of convergence and contractive mapping, please refer to Ortega and Rheinboldt (1970, Chapter 5). All of these algorithms are essentially based on the idea that a relatively difficult optimization problem can be converted to a much simpler iterative algorithm with guaranteed, albeit slow, convergence. A visual imagery is apt here: Instead of embarking upon a direct and treacherously steep climb, we approach the summit through a winding, gradually ascending path. Interestingly, this idea has become very attractive now with the advent of big data revolution and high-dimensional applications, where solving the original optimization problem is either impossible or prohibitively expensive. There is a nice analogy to this in numerical linear algebra for solving large-scale linear system of equations. Indirect, iterative techniques (e.g., Gauss-Seidel) for solving linear systems were considered to be too slow and impractical, and only of pedagogical interest. Instead, the attention of the research community was focused on direct methods such as the various decomposition and factorization methods (LU, QR, SVD). But, such direct techniques are ill-suited to solve the modern day, large-scale linear systems with millions of equations. Therefore, clever adaptations of indirect iterative methods are emerging as the methods of choice (e.g., conjugate-gradient; Censor and Zenios 1997; Saad 2003). Similarly, for the estimation of statistical models in large, high-dimensional modern applications, EM-like algorithms are becoming indispensable tools in the arsenal of computational scientists (Patro, Mount, and Kingsford 2014; Shiraishi, Tremmel, Miyano, and Stephens 2015; Raj, Shim, Gilad, Pritchard, and Stephens 2015; Chiou, Xu, Yan, and Huang 2018, etc.).

EM-like algorithms are characterized by two essential features: reliable, monotone convergence, and slow, linear rate of convergence. Therefore, any strategy that can accelerate the rate of convergence of these algorithms, without compromising on their reliability and ease of use, will be of huge help. Zhou, Alexander, and Lange (2011) remarked that "In many statistical problems, maximum likelihood estimation by an EM or MM algorithm suffers from excruciatingly slow convergence. This tendency limits the application of these algorithms to modern high-dimensional problems in data mining, genomics, and imaging. Unfortunately, most existing acceleration techniques are ill-suited to complicated models involving large numbers of parameters. The squared iterative methods (SQUAREM) recently proposed by Varadhan and Roland constitute one notable exception." The goal of this paper is to demonstrate the utility of this "notable exception", SQUAREM, proposed by Varadhan and Roland (2008), which is available in the R package **SQUAREM** (Varadhan 2020).

The main aim of **SQUAREM** is to facilitate the development of computationally efficient new statistical models. In particular, our package provides acceleration schemes which can speed up the estimation of the statistical models, where the model parameters are estimated with monotone, EM-like algorithms. Acceleration of these estimation algorithms can be readily achieved using the function `squarem()`. Here we demonstrate the simplicity, generality, and power of **SQUAREM** through a wide array of applications of EM/MM problems in R (R Core Team 2019), showcasing how easy it is to use SQUAREM to derive efficient solutions. However, it should be recognized that there is no foolproof numerical algorithm; in poorly identified problems, where even the EM algorithm can fail, SQUAREM is not guaranteed to work.

2. Squared iterative method

Suppose we have observed data $y = (y_1, \ldots, y_N)^\top$ that comes from a probability density function $g(y; \theta)$ where $\theta \in \Omega \subset \mathbb{R}^p$ is the parameter of interest. We are often interested in computing the MLE (maximum likelihood estimates) of θ, denoted by $\theta^\star$. The EM algorithm is a popular technique for computing MLE, which consists of two steps, E step and M step (Dempster *et al.* 1977). The EM algorithm is natural when there is a missing data component z in the probability model, which when known greatly simplifies the estimation of $\theta^\star$. Let us use $x = \{y, z\}$ to denote the complete data. EM algorithm then becomes:

E step: A Q function is constructed such that

$$Q(\theta; \theta_n) = \int L_c(\theta; x) f(z; y, \theta_n) dz, \quad n = 0, 1, \ldots,$$

where n refers to the nth iteration of the algorithm, $L_c(\theta; x)$ is the complete data log-likelihood, and $f(z; y, \theta_n)$ is the conditional density function of missing data z given observed data y. Thus, the Q function computes the expected value of the complete data log-likelihood given the current parameter estimates and observed data.

M step: M step maximizes the Q function obtained in the E step over $\theta \in \Omega \subset \mathbb{R}^p$ to iteratively compute the next, $(n+1)$th iteration of parameter values, θ_{n+1}, such that

$$\theta_{n+1} = \operatorname{argmax} Q(\theta; \theta_n), \quad n = 0, 1, \ldots, .$$

The EM algorithm therefore defines a fixed-point mapping F such that $F : \Omega \subset \mathbb{R}^p \mapsto \Omega$ and

$$\theta_{n+1} = F(\theta_n), \quad n = 0, 1, \ldots .$$

Two convergence criteria can be applied and are both satisfied by the EM algorithm : 1) for parameter estimates θ_n, as $n \to \infty$, $\|\theta_n - \theta^\star\| \to 0$ ($\|\cdot\|$ is the Euclidean norm); 2) the convergence is defined by the sequence of the likelihood function of the parameter estimates, $L(\theta_n)$, such that $|L(\theta_n) - L(\theta^\star)| \to 0$, as $n \to \infty$. The EM algorithm guarantees to produce monotone convergence such that $L(\theta_{n+1}) \geq L(\theta_n)$. By Taylor's theorem under regularity conditions, expand $F(\theta_n)$ around $\theta^\star$:

$$\theta_{n+1} - \theta^\star = J(\theta^\star)(\theta_n - \theta^\star) + o(\|\theta_n - \theta^\star\|),$$

where $J(\theta^\star)$ is the Jacobian matrix of F evaluated at $\theta^\star$. Dempster *et al.* (1977) showed that $J(\theta^\star)$, the Jacobian matrix, measures the fraction of missing information. Under weak regularity conditions, the eigenvalues of $J(\theta^\star)$ lie on $[0, 1)$. Thus, the largest eigenvalue of $J(\theta^\star)$ governs the rate of convergence for EM. The closer this is to unity, the slower EM converges as it indicates a large fraction of missing information.

Motivated by the Cauchy-Barzilai-Borwein (CBB) method (Raydan and Svaiter 2002), Roland and Varadhan (2005) and Varadhan and Roland (2008) constructed SQUAREM by defining the following recursive error relation:

$$e_{n+1} = [I - \alpha_n (J - I)]^2 e_n,$$

where $e_n = \theta_n - \theta^\star$, I is the identity matrix and α_n is the steplength that takes into account the larger eigenvalues of $J(\theta^\star)$. The pseudocode for the SQUAREM algorithm is listed in

Algorithm 1: Pseudocode for SQUAREM.

Input: F, L, θ_0, $\eta \geq 0$
while *not converged* **do**
  $\theta_1 = F(\theta_0)$
  $\theta_2 = F(\theta_1)$
  $r = \theta_1 - \theta_0$
  $v = (\theta_2 - \theta_1) - r$
  Compute steplength α
  $\theta_{sq} = \theta_0 - 2\alpha r + \alpha^2 v$
  if $L(\theta_{sq}) > L(\theta_2) - \eta$ **then**
    Set $\theta' = \theta_{sq}$.
  else
    $\theta' = \theta_2$
  end
  $\theta_0 = F(\theta')$, stabilization step (done only if $\theta' = \theta_{sq}$)
end

Algorithm 1 (Varadhan and Roland 2008), which demonstrates the remarkable simplicity of the proposed method.

There are three choices for α, the steplength as described in Varadhan and Roland (2008). It is our experience that $\alpha = -\frac{\|r\|}{\|v\|}$ generally works the best, and hence it is the default steplength used in **SQUAREM**. Varadhan and Roland (2008) also showed global convergence of the SQUAREM algorithm, i.e., SQUAREM can converge to a stationary point from any starting value in the parameter space, or at least, in a large part of it by modifying steplength to ensure monotonicity. Note that when steplength is equal to -1, one SQUAREM evaluation is equivalent to two EM iterations. Thus, each iteration of SQUAREM involves 2 or 3 evaluations of EM. Hence, when we compare the two methods, we use the number of EM steps rather than the number of iterations. Apart from the EM steps, there is minimal cost in computing the SQUAREM parameter updates, including the computation of the value of likelihood functions. In addition to the convergence criteria provided earlier, we give a definition of convergence acceleration as follows: Suppose $\{\theta_n\}$ is the sequence of estimates produced by Algorithm 1, while $\{\theta'_n\}$ is that given by Algorithm 2, then we say that Algorithm 2 accelerates Algorithm 1 if $\frac{\|\theta'_n - \theta^\star\|}{\|\theta_n - \theta^\star\|} \to 0$ as $n \to \infty$.

3. Description of R package SQUAREM

R package **SQUAREM** is available from the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/package=SQUAREM/. **SQUAREM** works for any smooth, contraction mapping with a linear convergence rate (e.g., EM-like algorithms). We describe below the two main functions, `squarem()` and `fpiter()`. Undoubtedly, `squarem()` is the featured function in the package.

- `squarem()`, for squared iterative scheme:

  `squarem()` is a function to accelerate any smooth, contractive, fixed-point iteration

algorithm including EM/MM and other EM-like algorithms. The main arguments include `par`, `fixptfn`, `objfn` and `control`. `par` denotes the starting value of parameters. The argument `fixptfn` defines a function F constituting the fixed-point iteration: $\theta_{k+1} = F(\theta_k)$. `fixptfn` encodes a single step of any EM-like algorithm.

```
R> fixptfn <- function(par, data, ...) {
+    pnew <- F(par, data, ...)
+    return(pnew)
+  }
```

`objfn` is the objective function we want to minimize. In the case of EM-like algorithms, it would be the negative log-likelihood function of data. It is not essential to supply the objective function in order for the function to work, but its provision guarantees global convergence. `control` specifies a list of algorithm options including `maxiter`, maximum number of iterations, and `tol`, tolerance, among others. If $\|F(\theta_k) - \theta_k\| \leq$ `tol`, the algorithm declares convergence at the $(k+1)$-th iteration ($\|\cdot\|$ shows the Euclidean norm). Under regularity conditions given by Wu (1983), the satisfaction of the convergence does imply a local optimum. There are 3 other important control parameters in `squarem()`, namely, `K`, `method` and `objfn.inc`. `method` specifies the choice of steplength and `K` specifies the order of the squared iterative scheme. The default values `method = 3` and `K = 1` generally work well. `objfn.inc` guides the monotonicity of the objective function. Setting `objfn.inc = 0` ensures strict monotonicity, while `objfn.inc = Inf` results in an unguarded acceleration scheme, where the objective function is not evaluated at all. The default is `objfn.inc = 1`, resulting in a nearly-monotone acceleration scheme. Another option is to set the value at the average log-likelihood, i.e., the log-likelihood per individual sample.

To summarize, the default usage of `squarem()` is

```
squarem(par, fixptfn, objfn, ..., control = list())
```

- `fpiter()`, for fixed-point iteration scheme:

  `fpiter()` is a function to implement the fixed-point iteration algorithm including EM, MM and other EM-like algorithms as is. The main arguments include `par`, `fixptfn`, `objfn` and `control`, working the same way as in `squarem()` except that there are no SQUAREM specific control parameters in the argument `control`.

  To summarize, the default usage of `fpiter()` is

  ```
  fpiter(par, fixptfn, objfn, ..., control = list())
  ```

In the next section, we demonstrate a detailed illustration of how to implement SQUAREM in R.

4. How to apply SQUAREM acceleration

Imagine that an EM-like algorithm is used to estimate a model, with slow, linear rate of convergence. In order to speed up the algorithm using R package **SQUAREM**, there are two main steps to be prepared.

Death, i	Frequency, n_i	Death, i	Frequency, n_i
0	162	5	61
1	267	6	27
2	271	7	8
3	185	8	3
4	111	9	1

Table 1: Data on deaths of women 80 years or older during 1910 to 1912 from *The London Times*.

Step 1: Create an R function that fulfils one iteration of the EM-like algorithm. This function corresponds to the argument `fixptfn` in function `squarem()`.

Step 2: Write an associated merit function to minimize, for example, the negative log-likelihood function. This function passes to the argument `objfn` in function `squarem()`.

There are several other arguments in function `squarem()` as specified in Section 3, such as starting values, tolerance and maximum number of iterations, but the default choices often work well. Once we have these arguments ready, we can launch the function `squarem()` to put the acceleration into production, simple and easy. We next illustrate the usage of R package **SQUAREM** in detail with a simple example of a mixture problem introduced below, which was also discussed in Varadhan and Roland (2008). Here we revisit this example, mainly to illustrate how remarkably easy it is to apply the `squarem()` function from an existing EM algorithm script.

In many studies, the study sample comes from a population mixing two or more types of units, each with varying characteristics. Finite mixture models are ideally suited to account for this kind of heterogeneity. A finite mixture model estimates parameters describing each subpopulation and their mixing probabilities. The EM algorithm is a popular technique to compute the maximum likelihood estimates for mixture models, but is notorious for its slow convergence. Here, we use a two-component Poisson mixture to illustrate the usage and power of SQUAREM compared to the EM algorithm. We use the data on the number of deaths of women 80 years and older during the years 1910–1912 from *The London Times* (Hasselblad 1969).

We use p to denote the mixing probability and let μ_1, μ_2 be the mean of the Poisson distribution from population 1 and 2, respectively. Let i be the number of death, $i = 0, 1, \ldots, 9$ and n_i be the number of days when number of death i occurred.

The real data are displayed in Table 1, where the number of death varies with values $i = 0, 1, \ldots, 9$.

EM algorithm

For derivation of the EM step, see Appendix B.1.

The EM update is such that

$$p^{(k+1)} = \frac{\sum_i n_i p_i^{(k)}}{\sum_i n_i}, \qquad \mu_1^{(k+1)} = \frac{\sum_i i n_i p_i^{(k)}}{\sum_i n_i p_i^{(k)}}, \qquad \mu_2^{(k+1)} = \frac{\sum_i i n_i (1 - p_i^{(k)})}{\sum_i n_i (1 - p_i^{(k)})},$$

where $p^{(k+1)}, \mu_1^{(k+1)}, \mu_2^{(k+1)}$ are the derived estimates in the $(k+1)$th iteration and $p_i^{(k)}$ is defined in Appendix B.1. Below demonstrates how easy it is to set up SQUAREM acceleration of an EM-like algorithm. We implement the EM algorithm using function `EM.poisson.mixture()` provided below.

```
R> EM.poisson.mixture <- function(p, maxiter = 5000, tol = 1e-08, y) {
+    iter <- 1
+    conv <- FALSE
+    pnew <- rep(NA_real_, 3)
+    while (iter < maxiter) {
+      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))
+      res <- sqrt(crossprod(pnew - p))
+      p <- pnew
+      if (res < tol) {
+        conv <- TRUE
+        break
+      }
+      iter <- iter + 1
+    }
+    return(list(par = p, fpevals = iter, convergence = conv))
+  }
```

In order to implement SQUAREM using function `squarem()`, we extract the part in the above EM function that corresponds to one EM step and put it into a separate function `poissmix.em()`. This function corresponds to the argument `fixptfn` in the `squarem()` function. By cutting and pasting the relevant code chunk from the function above, we create such a function for `fixptfn` and complete step 1 in applying SQUAREM.

```
R> poissmix.em <- function(p, y) {
+    pnew <- rep(NA_real_, 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)
+  }
```

Step 2 is to write an associated merit function to minimize, in this case, namely the negative log-likelihood function. The log-likelihood of observed data i, n_i is such that:

$$\ell(p, \mu_1, \mu_2) = \sum_i n_i (\log [p e^{-\mu_1} \mu_1^i / i! + (1-p) e^{-\mu_2} \mu_2^i / i!]).$$

Therefore, the negative log-likelihood is coded into function `poissmix.loglik()`. This function corresponds to the argument `objfn` in function `squarem()`.

```
R> poissmix.loglik <- 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))
+  }
```

We are now all set to apply `squarem()` and compare to the EM algorithm. We set the starting value $(p, \mu_1, \mu_2) = (0.3, 1, 5)$ and tolerance being 10^{-8}.

```
R> library("SQUAREM")
R> poissmix.dat <- data.frame(death = 0 : 9,
+    freq = c(162, 267, 271, 185, 111, 61, 27, 8, 3, 1))
R> y <- poissmix.dat$freq
R> p0 <- c(0.3, 1, 5)
R> system.time(f0 <- EM.poisson.mixture(p = p0, y = y))

user system elapsed
0.036 0.005 0.040

R> f0

$par
[1] 0.3598864 1.2560968 2.6634056
$value.objfn
[1] 1989.946

$fpevals
[1] 2696

$convergence
[1] TRUE

R> system.time(f1 <- fpiter(par = p0, fixptfn = poissmix.em,
+    objfn = poissmix.loglik, control = list(tol = 1.e-08), y = y))

user system elapsed
0.039 0.000 0.039

R> f1

$par
[1] 0.3598864 1.2560968 2.6634056
```

```
$value.objfn
[1] 1989.946

$fpevals
[1] 2696

$objfevals
[1] 0

$convergence
[1] TRUE
```

```
R> system.time(f2 <- squarem(par = p0, fixptfn = poissmix.em,
+    objfn = poissmix.loglik, control = list(tol = 1.e-08), y = y))

user system elapsed
0.003 0.000 0.002

R> f2

$par
[1] 0.3598859 1.2560960 2.6634050

$value.objfn
[1] 1989.946

$iter
[1] 19

$fpevals
[1] 54

$objfevals
[1] 19

$convergence
[1] TRUE
```

The output shows the equivalence in parameter estimates, maximum likelihood value and the number of EM iterations between the EM algorithm function `EM.poisson.mixture()` and the function `fpiter()` provided in R package **SQUAREM**. A dramatic improvement for SQUAREM over the EM algorithm has been seen as it outperforms EM by a factor of 50 in terms of the number of EM evaluations and by a factor of 20 with regards to the CPU running time. From this point on, we use function `fpiter()` to implement EM and other EM-like algorithms for ease of illustration.

We next run two algorithms (EM and SQUAREM) for 5000 randomly generated starting values

$$(p, \mu_1, \mu_2) = (U[0.05, 0.95], U[0, 20], U[0, 20]),$$

	fevals	CPU time (s)
EM	3140 (2511, 3182)	0.039 (0.031, 0.046)
SQUAREM	101 (57, 132)	0.003 (0.002, 0.005)

Table 2: The convergence performance in terms of the number of EM evaluations, fevals and the CPU running time comparing EM to SQUAREM for Poisson mixture estimation with 5000 randomly generated starting values. The median value is reported along with a corresponding window (2.5-th percentile, 97.5-th percentile).

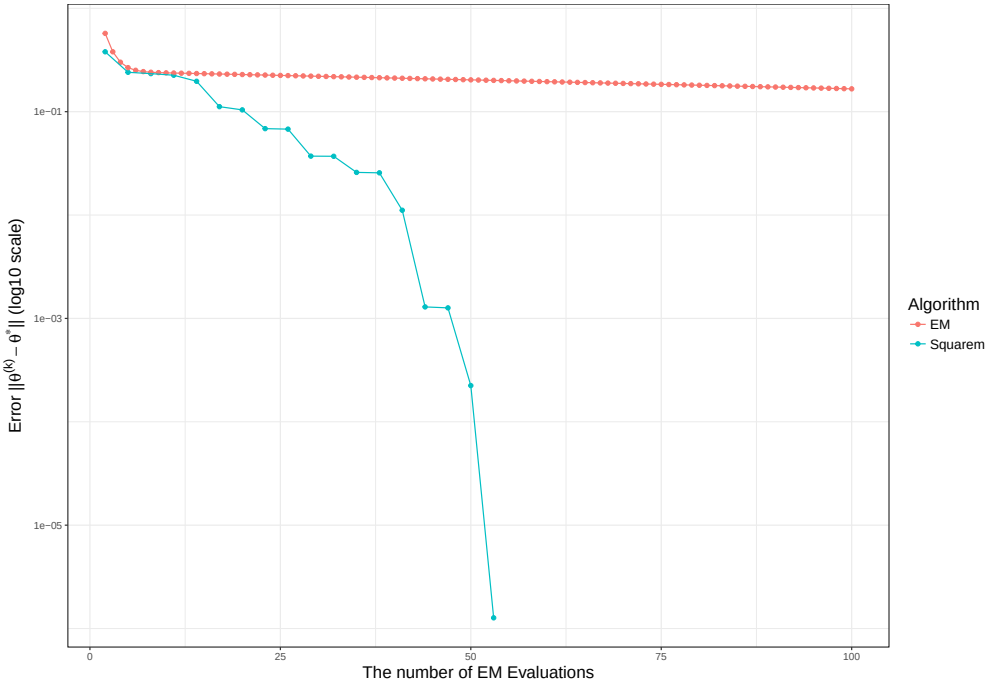


Figure 1: The comparison of convergence behavior between SQUAREM and EM.

where $U[a, b]$ is a uniform random variable on the interval $[a, b]$. Table 2 displays the results. We provide the median and a corresponding window from 2.5th to 97.5-th percentile for the number of EM evaluations, fevals and the CPU running time (in seconds). In general, SQUAREM converges 13 times faster than EM with only 3.2% of the number of EM evaluations that the EM algorithm would take.

We also plot the error curve $\|\theta^{(k)} - \theta^*\|$ as a function of the number of EM evaluations k in Figure 1 using the starting value $(p, \mu_1, \mu_2) = (0.3, 1, 5)$, where $\theta^{(k)}$ are the kth iteration estimates $(p^{(k)}, \mu_1^{(k)}, \mu_2^{(k)})$. The truth, θ^*, is derived by running `squarem()` with a very small convergence tolerance, e.g., 10^{-18}. Figure 1 shows that the error for SQUAREM drops at a much faster rate than EM. SQUAREM converges in approximately 50 EM evaluations while the error still remains considerably large for EM after 100 iterations.

In the next section, we continue to demonstrate the utility of **SQUAREM** through a wide array of applications of EM/MM problems, including interval censoring, genetics admixture, and logistic regression maximum likelihood estimation (an MM problem).

5. Examples

5.1. Interval censoring

Interval censoring is a common phenomenon in survival analysis, where we do not observe the precise time of an event for each individual, but we only know the time interval during which the individual's event occurs. Following the notations in Gentleman and Geyer (1994), we assume that survival time, X, also known as failure time, come from a distribution F. Each individual i goes through a sequence of inspection times $t_{i,1}, t_{i,2}, \ldots$ The survival time x_i for individual i is not observed, however, the last inspection time prior to x_i and the first inspection time after are recorded. An example of interval censored data is displayed in Table 3.

Therefore, data consist of time intervals $I_i = (L_i, R_i)$ for each individual $i, i = 1, 2, \ldots, n$ and the event for individual i is known to happen during that interval. Let $\{s_j\}_{j=0}^m$ be the unique ordered times of $\{0, \{L_i\}_{i=1}^n, \{R_i\}_{i=1}^n\}$, and $\alpha_{ij}, i = 1, 2, \ldots, n, j = 1, 2, \ldots, m$, the ij cell of an α matrix, be such that

$$\alpha_{ij} = \begin{cases} 1 & \text{if } (s_{j-1}, s_j) \subseteq I_i, \text{ the event for individual } i \text{ can occur in } (s_{j-1}, s_j) \\ 0 & \text{otherwise} \end{cases}$$

and $p_j = F(s_j-) - F(s_{j-1})$, $p = (p_1, p_2, \ldots, p_m)^\top$. The log-likelihood of the data is therefore

$$\ell(p) = \sum_{i=1}^{n} \log \left(\sum_{j=1}^{m} \alpha_{ij} p_j \right).$$

The negative log-likelihood is coded in function `loglik()`, corresponding to the argument `objfn` in `squarem()`. `A` in function `loglik()` refers to the alpha matrix α and `pvec` is the vector of probabilities, p.

```
R> loglik <- function(pvec, A) {
+    -sum(log(c(A %*% pvec)))
+  }
```

EM algorithm

For derivation of the EM step, see Appendix B.3.

	Last inspection time prior to x_i	First inspection time after x_i
Individual 1	1	3
Individual 2	2	6
⋮	⋮	⋮
Individual n	3	4

Table 3: The example of interval censored data (unit: year).

(45, Inf]	(6, 10]	(0, 7]	(46, Inf]	(7, 16]	(17, Inf]
(7, 14]	(37, 44]	(0, 8]	(4, 11]	(15, Inf]	(11, 15]
(22, Inf]	(46, Inf]	(46, Inf]	(25, 37]	(46, Inf]	(26, 40]
(46, Inf]	(27, 34]	(36, 44]	(46, Inf]	(36, 48]	(37, Inf]
(40, Inf]	(17, 25]	(46, Inf]	(11, 18]	(38, Inf]	(5, 12]
(37, Inf]	(0, 5]	(18, Inf]	(24, Inf]	(36, Inf]	(5, 11]
(19, 35]	(17, 25]	(24, Inf]	(32, Inf]	(33, Inf]	(19, 26]
(37, Inf]	(34, Inf]	(36, Inf]	(46, Inf]		

Table 4: The censored intervals when cosmetic deterioration occurred.

The EM update is such that

$$p_j^{(k+1)} = \frac{1}{n}\sum_{i=1}^{n}\mu_{ij}, \quad j = 1, 2, \ldots, m, \qquad p^{(k+1)} = (p_1^{(k+1)}, p_2^{(k+1)}, \ldots, p_m^{(k+1)})^\top,$$

where $p^{(k+1)}$ is the $(k+1)$th iteration of derived estimates and $\mu_{ij} = \frac{\alpha_{ij}p_j}{\sum_s \alpha_{is}p_s}$, $i = 1, 2, \ldots, n, j = 1, 2, \ldots, m$. Such one EM update is written in function `intEM()`, corresponding to the argument `fixptfn` in `squarem()`.

```
R> intEM <- function(pvec, A) {
+    tA <- t(A)
+    Ap <- pvec * tA
+    pnew <- colMeans(t(Ap)/colSums(Ap))
+    pnew * (pnew > 0)
+  }
```

EM-ICM algorithm

Wellner and Zhan (1997) developed a hybrid algorithm called EM-ICM for the MLE computation of interval censored data. This algorithm alternates steps of iterative convex minorant (ICM) and of EM. Wellner and Zhan (1997) showed that EM-ICM substantially improves the performance of the EM algorithm. In addition to comparing to EM, we also compare SQUAREM with EM-ICM, the dedicated method for this problem, using a real data example and simulations. We use function `EMICM()` in R package **interval** (Fay and Shaw 2010) to implement the EM-ICM algorithm.

Real data example. The real data come from Finkelstein and Wolfe (1985) and provide the interval when cosmetic deterioration occurred in 46 individuals with early breast cancer under radiotherapy. Table 4 shows the censored intervals for each individual.

We use function `Aintmap` in R package **interval** to produce matrix α and then generate starting values.

```
R> library("interval")
R> A <- Aintmap(dat[, 1], dat[, 2])
R> m <- ncol(A)
R> pvec <- rep(1/m, length = m)
```

We modified the function `EMICM()` in R package **interval** in order to keep the same starting values across all algorithms, a uniform starting value where $p_i = 1/m$, $i = 1, 2, \ldots, m$. We next compare the performance of the aforementioned three algorithms, EM, SQUAREM and EM-ICM. We did not include EM-ICM for comparison in the number of EM evaluations because intrinsically the EM-ICM algorithm is a hybrid algorithm where each EM-ICM step is different from an EM evaluation. The tolerance for convergence is set at 10^{-8}, the same across all algorithms.

EM algorithm:

```
R> system.time(ans1 <- fpiter(par = pvec, fixptfn = intEM,
+    objfn = loglik, A = A, control = list(tol = 1e-8)))

 user  system elapsed
0.008   0.000   0.009

R> ans1$fpevals

[1] 216
```

SQUAREM:

```
R> system.time(ans2 <- squarem(par = pvec, fixptfn = intEM,
+    objfn = loglik, A = A, control = list(tol = 1e-8)))

 user  system elapsed
0.002   0.000   0.002

R> ans2$fpevals

[1] 40
```

EM-ICM algorithm:

```
R> system.time(ans3 <- EMICM.mod(dat, EMstep = TRUE, ICMstep = TRUE,
+    keepiter = FALSE, tol = 1e-08, maxiter = 1000))

 user  system elapsed
0.025   0.001   0.027

R> max(abs(ans1$par - ans2$par))

[1] 0

R> max(abs(ans2$par - ans3$pf))

[1] 4.707805e-05
```

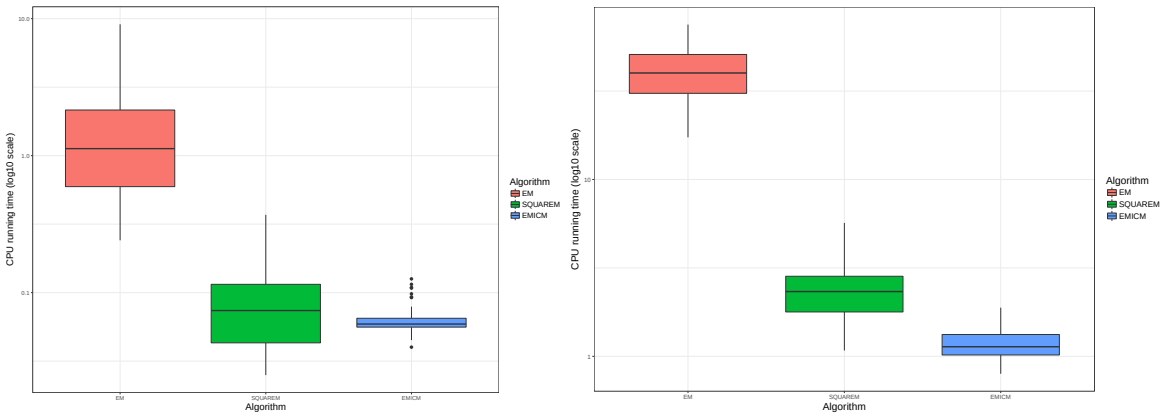


Figure 2: The comparison of CPU running time among EM, SQUAREM and EM-ICM algorithms, applied to 100 simulated datasets for varied sample size $n = 200$ (left) and $n = 2000$ (right).

All three algorithms converge to the same point as evidenced by the maximum difference in absolute value between the parameter estimates returned by these algorithms. The EM algorithm performs fairly well on this real dataset, largely due to the small sample size. Even so, SQUAREM still outperforms the EM by a factor of 5 in terms of the number of EM evaluations and a factor of 4 in CPU running time. We show in the following section that SQUAREM and EM-ICM algorithms are more advantageous than the EM algorithm as sample size increases using simulated examples.

Simulation example. For each individual, we randomly generate censored intervals by creating a survival time (event) and a stochastic sequence of inspection times. The left end of the interval is the last inspection time before the event while the right end is the first inspection time after. The function we use to generate interval censored data is coded in `gendata()`.

```
R> gendata <- function(n, mu.nexam = 5) {
+    foo <- matrix(NA_real_, nrow = n, ncol = 3)
+    for (i in 1:n) {
+      st <- rweibull(1, shape = 1, scale = 5)
+      nexam <- rpois(1, mu.nexam)
+      exam <- round(runif(nexam, 0, 10), 1)
+      exam <- c(0, exam, Inf)
+      foo[i, ] <- c(time = st, L = max(exam[st > exam]),
+        R = min(exam[st <= exam]))}
+    return(foo)
}
```

We compare the performance of the EM, SQUAREM and EM-ICM algorithms starting from sample size $n = 200$ with 100 simulations. The results are summarized in Figure 2 on the left. It can be seen from the left plot in Figure 2 that SQUAREM and EM-ICM algorithms are both, on average, approximately 10 times faster than EM, for a moderate sample size $n = 200$.

n		EM	SQUAREM
200	Mean	6746	446
	Standard deviation	4738	306
2000	Mean	13398	792
	Standard deviation	3948	263

Table 5: The comparison of mean and standard deviation of the number of EM evaluations between the EM algorithm and SQUAREM on the simulated data examples for a moderate sample size $n = 200$ and a large sample size $n = 2000$.

The performance of both algorithms are comparable, with EM-ICM having a much compact distribution of CPU running time. In order to show the improvement in the number of EM evaluations comparing SQUAREM to EM, we summarize the mean and standard deviation of the number of EM evaluations for both algorithms in Table 5 for this simulation study.

On average, EM algorithm takes 15 times more EM steps to converge than SQUAREM for this simulation study with a moderate sample size $n = 200$. Next, we increase the sample size from $n = 200$ to $n = 2000$ and evaluate again the performance of the three algorithms on another set of 100 simulated interval censored datasets.

As sample size expands to $n = 2000$, the right plot in Figure 2 shows that the advantage of SQUAREM and EM-ICM algorithms becomes greater compared to EM. EM-ICM is specifically tailored to interval censoring maximum likelihood estimation, hence it is not surprising that it is the fastest algorithm. However, it is noteworthy that SQUAREM, a general purpose and off-the-shelf EM-like algorithm accelerator, is competitive with EM-ICM in this example to which EM-ICM is a dedicated algorithm. Table 5 also compares the number of EM evaluations between the EM algorithm and SQUAREM for the sample size $n = 2000$. SQUAREM on average outperforms EM algorithm by a factor of 17 in terms of the number of EM evaluations.

5.2. Genetics global ancestry estimation problem

Here we demonstrate the use of SQUAREM to solve an important problem in quantitative genetics that is notoriously computationally challenging. Suppose our study population is an admixed population with K ancestral populations. The goal is to estimate the proportion of ancestry from each contributing population for each individual's entire genome and simultaneously estimate the allele frequencies of the K ancestral populations. Let us use $q_i = (q_{i1}, q_{i2}, \ldots, q_{iK})^\top$ to denote such admixture proportions for individual i, $i = 1, 2, \ldots, n$ where q_{ik} is the proportion of subject i genome that is attributed to the ancestral population k, $k = 1, 2, \ldots, K$ and n is the number of subjects. Let Q be the $n \times K$ admixture proportions matrix. We assume that all p genome-wide markers are bi-allelic (either allele 1 or allele 2). Let F be the $p \times K$ population allele frequency matrix with f_{jk} being the frequency of allele 1 at marker j, $j = 1, 2, \ldots, p$ in population k. Matrices F and Q consist of parameters we are interested in estimating. The data consist of genetic polymorphism data sampled from n diploid individuals. Specifically, we have recorded the genotype at p genetic polymorphisms ("markers") for each individual. Genotype at marker j for individual i is represented as allele 1 counts, $x_{ij} = 0, 1, 2$. We assume that individuals are independent and

under the admixture model, the log-likelihood of data is:

$$\ell(F,Q) = \sum_{i=1}^{n}\sum_{j=1}^{p}(x_{ij}\log\sum_{k=1}^{K} q_{ik}f_{jk} + (2-x_{ij})\log\sum_{k=1}^{K} q_{ik}(1-f_{jk})) + C,$$

where C is a constant that does not contain the parameters F and Q. See Alexander, Novembre, and Lange (2009) for a full description of the model.

The negative log-likelihood of such data is coded in function `loglike()`, corresponding to the argument `objfn` in `squarem()`.

```
R> loglike <- function(param, X, K) {
+    n <- nrow(X); p <- ncol(X)
+    F <- matrix(param[1 : (p * K)], p, K)
+    Q <- matrix(param[(p * K + 1) : (p * K + n * K)], n, K)
+    loglikelihood <- sum(X * log(Q %*% t(F)) + (2 - X) *
+      log(Q %*% (1 - t(F))))
+    return(-loglikelihood)
+  }
```

EM algorithm

For derivation of the EM step, see Appendix B.4.

The EM update of matrices F and Q is

$$f_{jk} = \frac{n_{jk}^{(1)}}{n_{jk}^{(1)} + n_{jk}^{(0)}}, \qquad\qquad q_{ik} = \frac{m_{ik}}{\sum_k m_{ik}},$$

where $n_{jk}^{(1)}, n_{jk}^{(0)}, m_{ik}$ are defined in Appendix B.4.

This one EM evaluation is written in function `admixture.em()`, corresponding to the argument `fixptfn` in `squarem()`. We adapt the code provided on Peter Carbonetto's GitHub account (Carbonetto 2016).

```
R> admixture.em <- function(param, X, K) {
+    eps <- 1e-6
+    n <- nrow(X); p <- ncol(X)
+    m <- matrix(eps, n, K)
+    n0 <- n1 <- matrix(eps, p, K)
+    F <- matrix(param[1 : (p * K)], p, K)
+    Q <- matrix(param[(p * K + 1): (p * K + n * K)], n, K)
+    r <- array(0, dim = c(p, 4, K, K))
+    for (i in 1:n) {
+      colnames(r) <- c("00", "01", "10", "11")
+      for (j in 1:K) {
+        for (k in 1:K) {
+          r[, "00", j, k] <- (X[i, ] == 0) * (1 - F[, j]) * (1 - F[, k])
+          r[, "01", j, k] <- (X[i, ] == 1) * (1 - F[, j]) * F[, k]
```

```
+          r[, "10", j, k] <- (X[i, ] == 1) * F[, j] * (1 - F[, k])
+          r[, "11", j, k] <- (X[i, ] == 2) * F[, j] * F[, k]
+          r[, , j, k] <- r[, , j, k] * Q[i, j] * Q[i, k]
+        }
+      }
+      dim(r) <- c(p, 4 * K^2)
+      r <- r / rowSums(r)
+      dim(r) <- c(p, 4, K, K)
+      colnames(r) <- c("00", "01", "10", "11")
+      m[i, ] <- m[i, ] + apply(r, 3, sum) + apply(r, 4, sum)
+      for (k in 1:K) {
+        n0[, k] <- n0[, k] + rowSums(drop(r[, "00", k, ])) +
+          rowSums(drop(r[, "01", k, ])) +
+          rowSums(drop(r[, "00", , k])) +
+          rowSums(drop(r[, "10", , k]))
+        n1[, k] <- n1[, k] + rowSums(drop(r[, "10", k, ])) +
+          rowSums(drop(r[, "11", k, ])) +
+          rowSums(drop(r[, "01", , k])) +
+          rowSums(drop(r[, "11", , k]))
+      }
+    }
+    F <- n1 / (n0 + n1)
+    Q <- m / rowSums(m)
+    return(c(as.vector(F), as.vector(Q)))
+  }
```

Simulation example. We simulate an allele 1 count matrix X where there are 150 individuals and 100 markers for each individual. We use $K = 3$. The starting value of f_{jk} is randomly drawn from a uniform distribution in the range of $(0, 1)$, while that of q_{ik} is $\frac{1}{K}$. We implement the EM algorithm and SQUAREM to compute maximum likelihood estimates of the matrices F and Q and compare their performance.

EM algorithm:

```
R> load("geno.sim.RData")
R> set.seed(413)
R> p <- 100; n <- 150; K <- 3
R> F <- matrix(runif(p * K), p, K)
R> Q <- matrix(1/K, n, K)
R> param.start <- c(as.vector(F), as.vector(Q))
R> system.time(f1 <- fpiter(par = param.start, fixptfn = admixture.em,
+    objfn = loglike, control = list(tol = 1e-4), X = geno, K = 3))

   user  system elapsed
197.094   5.817 203.040

R> f1$fpevals
```

```
[1] 1115
```

SQUAREM:

```
R> system.time(f2 <- squarem(par = param.start, fixptfn = admixture.em,
+    objfn = loglike, control = list(tol = 1e-4, maxiter = 2000),
+      X = geno, K = 3))

 user   system  elapsed
47.460   1.403  48.869

R> f2$fpevals

[1] 270
```

In this example, SQUAREM outperforms the EM algorithm by a factor of 4 in terms of both CPU running time and the number of EM evaluations. For large genetic datasets, the E step is by far the most computationally intensive part of the algorithm. For a faster implementation of the E step using C (and interfaced to R using the `.Call()` function), see Carbonetto (2016). Although the admixture problem is naturally framed using EM, its convergence is very slow. Alexander *et al.* (2009) implemented a faster solution to this problem (using a block relaxation optimization method), which has permitted application of the admixture model to very large genetic datasets. Our EM-based implementation in R is much slower than this admixture specific implementation, but, nevertheless, it serves to illustrate the benefits of SQUAREM in a difficult optimization problem from genetics.

5.3. MM algorithm: Logistic regression maximum likelihood estimation

In this section, we discuss a quadratic majorization algorithm (an MM algorithm) for computing the maximum likelihood estimates of logistic regression coefficients. Minorize and maximize or equivalently, majorize and minimize (MM) algorithms typically exhibit slow linear convergence just like the EM algorithm. We show that SQUAREM can provide significant acceleration of MM algorithms. The example in this section comes from (De Leeuw 2006).

The EM algorithm may be viewed as a special case of MM algorithms (Zhou and Zhang 2012). The majorization algorithms are widely applied, for example, in the work of De Leeuw (1994), Heiser (1995), Lange *et al.* (2000), among others. Suppose we want to minimize function f over $X \subseteq \mathbb{R}^n$. We construct a majorization function g on $X \times X$ such that

$$f(x) \leq g(x, x^{(k)}) \quad \forall x, x^{(k)} \in X,$$

$$f(x^{(k)}) = g(x^{(k)}, x^{(k)}) \quad \forall x^{(k)} \in X,$$

where k denotes the kth iteration, $k = 0, 1, \ldots$ Therefore, instead of minimizing f, we minimize g such that

$$x^{(k+1)} = \underset{x \in X}{\operatorname{argmin}}\, g(x, x^{(k)}).$$

We repeat the updates of x until convergence and this completes the majorization algorithm. Note that in the EM algorithm, the $Q(\theta; \theta_k)$ function plays the role of the minorizing function.

Quadratic majorization algorithm

Taylor's theorem often leads to quadratic majorization algorithms (Böhning and Lindsay 1988) where the majorization function g is quadratic. By Taylor's theorem, expand $f(x)$ at $x^{(k)}$,

$$f(x) = f(x^{(k)}) + (x - x^{(k)})^\top \partial f(x^{(k)}) + \frac{1}{2}(x - x^{(k)})^\top \partial^2 f(\xi)(x - x^{(k)}),$$

where ξ is on the line between x and $x^{(k)}$. The majorization function g is constructed by constructing a matrix B such that $B - \partial^2 f(\xi)$ is always positive semi-definite regardless of ξ. So,

$$g(x, x^{(k)}) = f(x^{(k)}) + (x - x^{(k)})^\top \partial f(x^{(k)}) + \frac{1}{2}(x - x^{(k)})^\top B(x - x^{(k)})$$

is a majorization function for f. Let us define a clever variable $z^{(k)}$ such that $z^{(k)} = x^{(k)} - B^{-1}\partial f(x^{(k)})$ and majorization function g is equivalent to the following:

$$g(x, x^{(k)}) = f(x^{(k)}) + \frac{1}{2}(x - z^{(k)})^\top B(x - z^{(k)}) - \frac{1}{2}\partial f(x^{(k)})^\top B^{-1}\partial f(x^{(k)}).$$

At the kth iteration, to minimize $g(x, x^{(k)})$ over $x \in X$ is simply to minimize $(x - z^{(k)})^\top B(x - z^{(k)})$, thus the majorization algorithm becomes:

$$x^{(k+1)} = x^{(k)} - B^{-1}\partial f(x^{(k)}).$$

Therefore, in order to implement the quadratic majorization algorithm, we need to construct the matrix B and compute the gradient of function f. Let us consider logistic regression maximum likelihood estimation. Suppose we have an $n \times p$ design matrix X where there are n subjects and p predictors. Let y_i be the number of successes for subject i, $i = 1, 2, \ldots, n$ given the overall number of experiments, N_i. We use β to denote the regression coefficients. The goal is to derive the maximum likelihood estimates of β.

The negative log-likelihood of data is:

$$\begin{aligned} f(\beta) &= -\log\left(\prod_i \mathsf{P}(y_i)\right) \propto -\log\left(\prod_i p_i(\beta)^{y_i}(1 - p_i(\beta))^{(N_i - y_i)}\right) \\ &= -\sum_{i=1}^n y_i \log p_i(\beta) - \sum_{i=1}^n (N_i - y_i)\log\left(1 - p_i(\beta)\right) \\ &= -\sum_{i=1}^n y_i x_i^\top \beta - \sum_{i=1}^n N_i \log\left(1 - p_i(\beta)\right), \end{aligned}$$

where

$$p_i(\beta) = \frac{1}{1 + \exp\left(-x_i^\top \beta\right)}.$$

The gradient of $f(\beta)$ is such that:

$$\partial f(\beta) = \sum_{i=1}^n (N_i p_i(\beta) - y_i)x_i = X^\top u(\beta),$$

where the ith element of $u(\beta)$ is $N_i p_i(\beta) - y_i$, while the second derivative of $f(\beta)$ is:

$$\partial^2 f(\beta) = \sum_{i=1}^n (N_i p_i(\beta)(1 - p_i(\beta)))x_i x_i^\top = X^\top V(\beta) X,$$

where $V(\beta)$ is a diagonal matrix with the ith diagonal element $N_i p_i(\beta)(1-p_i(\beta))$. Based on the fact that $p_i(\beta)(1-p_i(\beta)) \leq \frac{1}{4}$, the matrix B can be constructed such that $B = \frac{1}{4}X^\top NX$, where N is the diagonal matrix consisting of elements N_i. Thus the quadratic algorithm becomes:

$$\beta^{(k+1)} = \beta^{(k)} - 4(X^\top NX)^{-1}X^\top u(\beta).$$

Let us refer to the above algorithm as uniform bound quadratic algorithm since $p_i(\beta)(1-p_i(\beta)) \leq \frac{1}{4}$ uniformly for any β and each subject i. Jaakkola and Jordan (2000) and Groenen, Giaquinto, and Kiers (2003) developed a non-uniform bound, $X^\top W(\beta)X$, where $W(\beta)$ is a diagonal matrix that consists of elements $w_i(\beta) = N_i \frac{2p_i(\beta)-1}{2x_i^\top \beta}$, $i = 1, 2, \ldots, n$. Thus the non-uniform bound quadratic algorithm becomes:

$$\beta^{(k+1)} = \beta^{(k)} - (X^\top W(\beta)X)^{-1}X^\top u(\beta).$$

Real data example. We use the cancer remission data in Lee (1974). The outcome is a binary indicator of whether cancer remission occurred for the subject. Column 1 is the intercept and variables `V2`, `V3`, ..., `V7` are results of six medical tests. The first five lines of data are as follows:

```
R> ld <- read.table("lee_data.txt")
R> head(ld, 5)

V1  V2   V3   V4  V5    V6    V7 V8
 1 0.8 0.83 0.66 1.9 1.100 0.996  1
 1 0.9 0.36 0.32 1.4 0.740 0.992  1
 1 0.8 0.88 0.70 0.8 0.176 0.982  0
 1 1.0 0.87 0.87 0.7 1.053 0.986  0
 1 0.9 0.75 0.68 1.3 0.519 0.980  1
```

The negative log-likelihood function $f(\beta)$ is coded in `binom.loglike()`, corresponding to the argument `objfn` in `squarem()`.

```
R> binom.loglike <- function(par, Z, y) {
+    zb <- c(Z %*% par)
+    pib <- 1 / (1 + exp(-zb))
+    return(as.numeric(-t(y) %*% (Z %*% par) - sum(log(1 - pib))))
+  }
```

The one iteration update for both the uniform and the non-uniform bound quadratic majorization algorithms is written in function `qmub.update()` and `qmvb.update()`, respectively, corresponding to the argument `fixptfn` in `squarem()`. Note that for high dimensional linear problems, it is worthwhile to investigate more efficient computation that can further reduce computing time. See examples in Bates (2004).

```
R> qmub.update <- function(par, Z, y) {
+    Zmat <- solve(crossprod(Z)) %*% t(Z)
+    zb <- c(Z %*% par)
```

```
+    pib <- 1 / (1 + exp(-zb))
+    ub <-  pib - y
+    par <- par - 4 * c(Zmat %*% ub)
+    par
+  }
R> qmvb.update <- function(par, Z, y) {
+    zb <- c(Z %*% par)
+    pib <- 1 / (1 + exp(-zb))
+    wmat <- diag((2 * pib - 1)/(2 * zb))
+    ub <-  pib - y
+    Zmat <- solve(t(Z) %*% wmat %*% Z) %*% t(Z)
+    par <- par - c(Zmat %*% ub)
+    par
+  }
```

We next apply these two quadratic majorization algorithms and their "squared" versions where we implement the aforementioned squared iterative scheme (SQUAREM), to compare their performance. The tolerance used is 10^{-7} and the starting value is $\beta^{(0)} = (10, 10, \ldots, 10)^\top$.

Uniform bound quadratic majorization algorithm:

```
R> library("SQUAREM")
R> Z <- as.matrix(ld[, 1:7])
R> y <- ld[, 8]p0 <- rep(10, 7)
R> system.time(ans1 <- fpiter(par = p0, fixptfn = qmub.update,
+    objfn = binom.loglike, control = list(maxiter = 20000), Z = Z,
+    y = y))

   user  system  elapsed
  0.051   0.003   0.055

R> ans1

$par
[1]  58.0384838  24.6615508  19.2935824 -19.6012695   3.8959635
[6]   0.1510923  -87.4339059

$value.objfn
[1] 10.87533

$fpevals
[1] 1127

$objfevals
[1] 0

$convergence
[1] TRUE
```

Squared uniform bound quadratic majorization algorithm:

```
R> system.time(ans2 <- squarem(par = p0, fixptfn = qmub.update,
+    objfn = binom.loglike, Z = Z, y = y))

 user  system  elapsed
0.011   0.000   0.011

R> ans2

$par
[1]  58.0384863  24.6615466  19.2935777 -19.6012645   3.8959634
[6]  0.1510923  -87.4339043

$value.objfn
[1] 10.87533

$iter
[1] 41

$fpevals
[1] 118

$objfevals
[1] 43

$convergence
[1] TRUE
```

Non-uniform bound quadratic majorization algorithm:

```
R> system.time(ans3 <- fpiter(par = p0, fixptfn = qmvb.update,
+    objfn = binom.loglike, control = list(maxiter = 20000), Z = Z,
+    y = y)

 user  system  elapsed
0.029   0.001   0.030

R> ans3

$par
[1]  58.0384866  24.6615451  19.2935760 -19.6012627   3.8959634
[6]  0.1510923  -87.4339030

$value.objfn
[1] 10.87533

$fpevals
```

```
[1] 442

$objfevals
[1] 0

$convergence
[1] TRUE
```

Squared non-uniform bound quadratic majorization algorithm:

```
R> system.time(ans4 <- squarem(par = p0, fixptfn = qmvb.update,
+    objfn = binom.loglike, Z = Z, y = y))

 user  system  elapsed
0.009   0.000   0.009

R> ans4

$par
[1]  58.0384868  24.6615443  19.2935751 -19.6012618   3.8959633
[6]  0.1510923  -87.4339024

$value.objfn
[1] 10.87533

$iter
[1] 30

$fpevals
[1] 88

$objfevals
[1] 30

$convergence
[1] TRUE
```

All four algorithms converge to the same maximum likelihood estimates but SQUAREM improves on both uniform and non-uniform bound quadratic majorization algorithms in terms of the number of quadratic majorization updates and CPU running time (in seconds). For uniform bound, SQUAREM converges around 5 times faster and saves the number of quadratic majorization updates by a factor of 10. The non-uniform bound quadratic majorization performs better than the uniform bound counterpart, but its squared version empowers further acceleration. Compared to the non-uniform bound algorithm, SQUAREM shortens the computing time by a factor of 3 and cuts the number of quadratic majorization updates by a factor of 5.

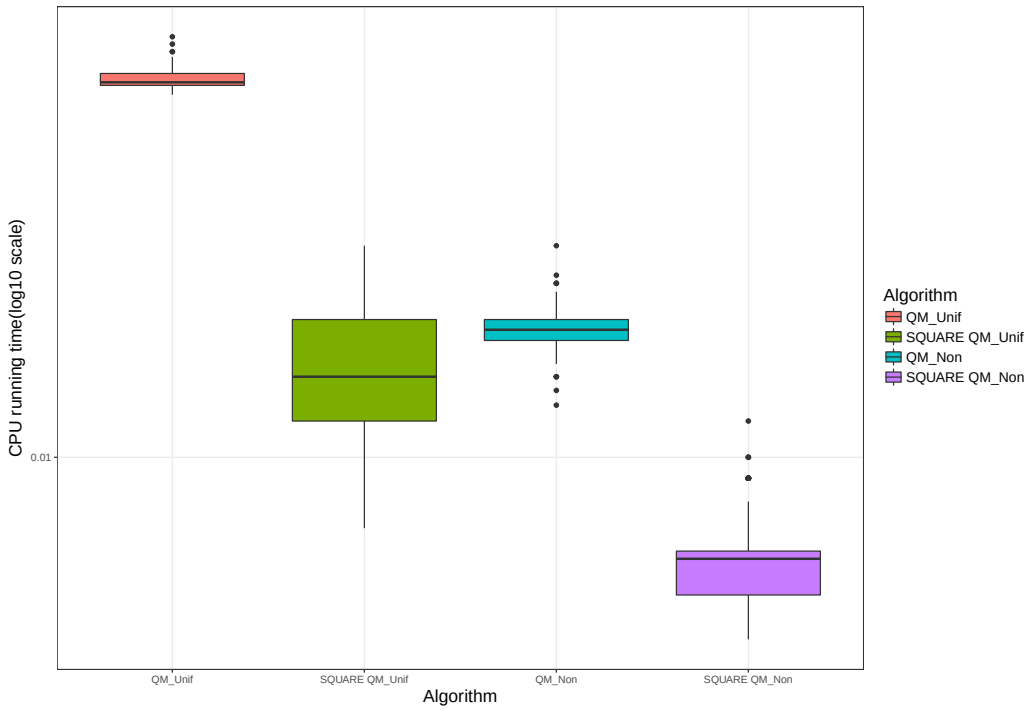


Figure 3: The comparison among the original and squared uniform/non-uniform bound quadratic majorization algorithms in terms of CPU running time (in seconds).

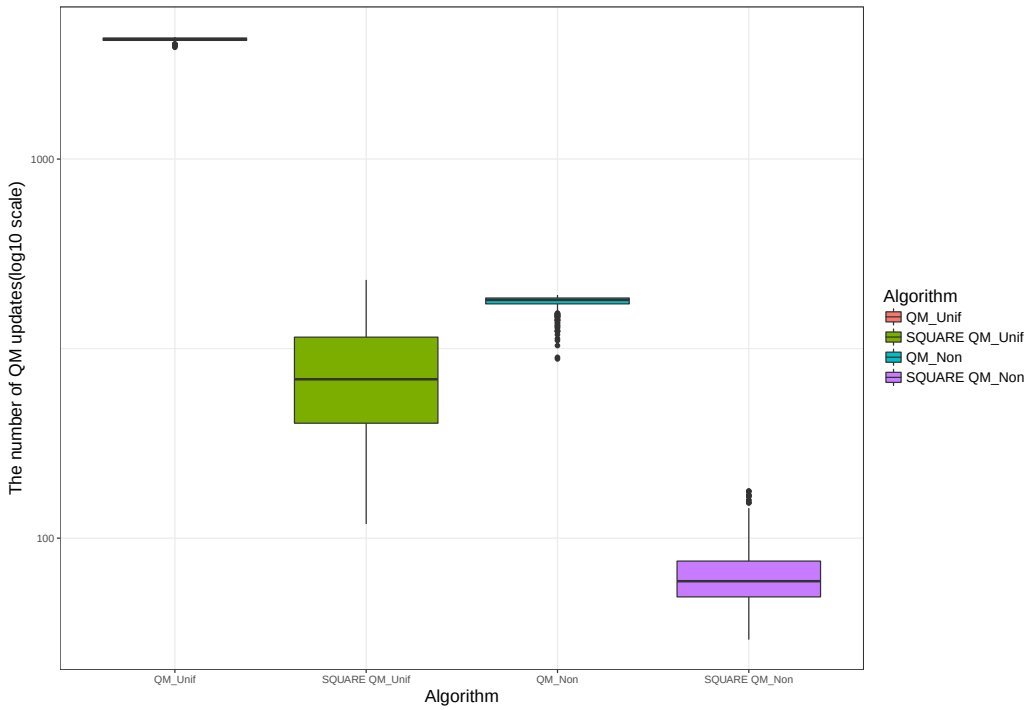


Figure 4: The comparison among the original and squared uniform/non-uniform bound quadratic majorization algorithms in terms of the number of quadratic majorization(QM) updates.

	Ub QM	Squared Ub QM	Non-Ub QM	Squared Non-Ub QM
CPU time (mean)	0.068	0.016	0.019	0.006
CPU time (sd)	0.002	0.005	0.001	0.001
QM updates (mean)	2068	273	419	80
QM updates (sd)	22	81	19	15

Table 6: The comparison among the original and squared uniform/non-uniform bound quadratic majorization algorithms in terms of the mean and standard deviation of CPU running time and the number of quadratic majorization (QM) updates for cancer remission data with 500 randomly selected starting values.

We randomly generate 500 starting values $\beta^{(0)} = (U(0,10), U(0,10), \ldots, U(0,10))^\top$ where $U(0,10)$ is a uniform random variable in the range of $(0,10)$. We then summarize the performance for these four algorithms in terms of CPU running time and the number of QM (quadratic majorization) updates in Figures 3 and 4.

Figures 3 and 4 clearly show that SQUAREM consistently provides substantial acceleration for both uniform bound and non-uniform bound quadratic majorization algorithms. Table 6 displays that for different starting values, compared to the uniform bound QM algorithm, its corresponding squared version on average saves the number of QM updates by a factor of 7 and runs 4 times faster. Non-uniform bound QM algorithm already improves the performance of the uniform bound counterpart, however, its squared version makes further improvement by a factor of 3 in CPU running time and a factor of 5 in the number of QM updates.

6. Discussion

Since the seminal work of Dempster *et al.* (1977), EM and its variants have become the workhorse of computational statistics. More broadly, there are iterative algorithms which do not fit into the missing data framework, but which are EM-like in the sense that they exhibit slow, monotone, global convergence like the EM algorithm. These include the minorize and maximize (MM) algorithm. Even more broadly, we can include all fixed-point iterations which are contractive (Ortega and Rheinboldt 1970) and linearly convergent. They are broader in the sense that there need not be an objective function (e.g., log-likelihood) associated with the contraction mapping. The remarkable fact is that it is extremely easy to use SQUAREM to try and accelerate these iterative algorithms. All that the user has to do is to create a function, `fixptfn()`, that implements a single step of the fixed-point iteration, whether it is EM/ECM/ECME/GEM/MM or any other contractive mapping. The objective function, `objfn()`, is optional, although we recommend that it be provided, if it is easy to code. The convergence criterion used in function `squarem()` is stringent for high-dimensional problems, and in future versions, we will incorporate other parameter-based criteria and criteria based on the objective function. In addition, there are other features of SQUAREM not illustrated in this paper, including the options of higher-order SQUAREM schemes and the tracking of algorithm's progress. For full features of SQUAREM, see https://CRAN.R-project.org/package=SQUAREM/SQUAREM.pdf.

The main theme of the paper is that existing modeling problems based on EM-like algorithms can potentially be made computationally more efficient by using the convergence acceleration

provided by SQUAREM. This is particularly true in high-dimensional problems where EM-like algorithms can be excruciatingly slow. There are several examples in the literature where SQUAREM has been effectively used. To name a few, Matthew Stephens' lab at the University of Chicago has been using **SQUAREM** in many of their models pertaining to genetic studies where a very large number of parameters are estimated (Shiraishi *et al.* 2015; Raj *et al.* 2015, among others). Patro *et al.* (2014) incorporated SQUAREM in the development of their new computational method, "Sailfish", to substantially increase the efficiency of processing sequencing reads. More recently, Chiou *et al.* (2018) used **SQUAREM** to speed up the estimation in a semi-parametric model for panel recurrent event count data. There are more such examples.

SQUAREM is computationally efficient. It requires little effort beyond the basic fixed-point iteration. The computation of SQUAREM parameter update is trivially easy since it only requires a couple of vector products. The only place where some inefficiency could occur is in the evaluation of the objective function to assess whether the SQUAREM step can be accepted. When the SQUAREM step results in non-monotonicity, it is rejected and instead the most recent EM update is retained. This results in some wasted effort. This is typical for algorithms with fast local convergence, for example, the Newton's method in unconstrained optimization which needs to be safe-guarded with line-search or a trust-region approach to ensure global convergence. However, SQUAREM is efficient when the objective function evaluation is relatively cheaper than that of the fixed-point iteration, which is often the case.

SQUAREM can be used off-the-shelf since there is no need for the user to tweak any control parameters to optimize its performance. Given its ease of application, SQUAREM may be considered as a default accelerator for EM-like algorithms. We invite the reader to try SQUAREM acceleration on their own EM-like algorithm or any slowly convergent, contractive fixed-point iteration.

References

Alexander DH, Novembre J, Lange K (2009). "Fast Model-Based Estimation of Ancestry in Unrelated Individuals." *Genome Research*, **19**(9), 1655–1664. doi:10.1101/gr.094052.109.

Bates D (2004). "Least Squares Calculations in R." *R News*, **4**(1), 17–20. URL https://CRAN.R-project.org/doc/Rnews/.

Böhning D, Lindsay BG (1988). "Monotonicity of Quadratic-Approximation Algorithms." *The Annals of the Institute of Statistical Mathematics*, **40**(4), 641–663. doi:10.1007/bf00049423.

Carbonetto P (2016). "**admixture**." GitHub repository, URL https://github.com/pcarbo/admixture.

Censor Y, Zenios SA (1997). *Parallel Optimization: Theory, Algorithms, and Applications*. Oxford University Press.

Chiou SH, Xu G, Yan J, Huang CY (2018). "Semiparametric Estimation of the Accelerated Mean Model with Panel Count Data under Informative Examination Times." *Biometrics*, **74**(3), 944–953. doi:10.1111/biom.12840.

De Leeuw J (1994). "Block-Relaxation Algorithms in Statistics." In *Information Systems and Data Analysis*, pp. 308–324. Springer-Verlag.

De Leeuw J (2006). "Quadratic and Cubic Majorization." *eScholarship 46r0p0vz*, UCLA: Department of Statistics, UCLA. URL https://escholarship.org/uc/item/46r0p0vz.

Dempster AP, Laird NM, Rubin DB (1977). "Maximum Likelihood from Incomplete Data via the EM Algorithm." *Journal of the Royal Statistical Society B*, **39**(1), 1–38. doi:10.1111/j.2517-6161.1977.tb01600.x.

Fay MP, Shaw PA (2010). "Exact and Asymptotic Weighted Logrank Tests for Interval Censored Data: The **interval** R Package." *Journal of Statistical Software*, **36**(2), 1–34. doi:10.18637/jss.v036.i02.

Finkelstein DM, Wolfe RA (1985). "A Semiparametric Model for Regression Analysis of Interval-Censored Failure Time Data." *Biometrics*, **41**(4), 933–945. doi:10.2307/2530965.

Gentleman R, Geyer CJ (1994). "Maximum Likelihood for Interval Censored Data: Consistency and Computation." *Biometrika*, **81**(3), 618–623. doi:10.1093/biomet/81.3.618.

Groenen PJF, Giaquinto P, Kiers HAL (2003). "Weighted Majorization Algorithms for Weighted Least Squares Decomposition Models." *Technical report*, Econometric Institute Research Papers.

Hasselblad V (1969). "Estimation of Finite Mixtures of Distributions from the Exponential Family." *Journal of the American Statistical Association*, **64**(328), 1459–1471. doi:10.1080/01621459.1969.10501071.

Heiser WJ (1995). "Convergent Computation by Iterative Majorization: Theory and Applications in Multidimensional Data Analysis." In WJ Krzanowski (ed.), *Recent Advances in Descriptive Multivariate Analysis*, pp. 157–189. Clarendon Press.

Jaakkola TS, Jordan MI (2000). "Bayesian Parameter Estimation via Variational Methods." *Statistics and Computing*, **10**(1), 25–37. doi:10.1023/a:1008932416310.

Jöreskog KG (1967). "A General Approach to Confirmatory Maximum Likelihood Factor Analysis." *ETS Research Bulletin Series*, **1967**(2), 183–202. doi:10.1002/j.2333-8504.1967.tb00991.x.

Lange K, Hunter DR, Yang I (2000). "Optimization Transfer Using Surrogate Objective Functions." *Journal of Computational and Graphical Statistics*, **9**(1), 1–20. doi:10.1080/10618600.2000.10474858.

Lee ET (1974). "A Computer Program for Linear Logistic Regression Analysis." *Computer Programs in Biomedicine*, **4**(2), 80–92. doi:10.1016/0010-468x(74)90011-7.

Liu C, Rubin DB (1998). "Maximum Likelihood Estimation of Factor Analysis Using the ECME Algorithm with Complete and Incomplete Data." *Statistica Sinica*, **8**(3), 729–747.

Meng XL, Rubin DB (1993). "Maximum Likelihood Estimation via the ECM Algorithm: A General Framework." *Biometrika*, **80**(2), 267–278. doi:10.1093/biomet/80.2.267.

Ortega JM, Rheinboldt WC (1970). *Iterative Solution of Nonlinear Equations in Several Variables*, volume 30. SIAM.

Patro R, Mount SM, Kingsford C (2014). "Sailfish Enables Alignment-Free Isoform Quantification from RNA-Seq Reads Using Lightweight Algorithms." *Nature Biotechnology*, **32**(5), 462–464. doi:10.1038/nbt.2862.

Raj A, Shim H, Gilad Y, Pritchard JK, Stephens M (2015). "**msCentipede**: Modeling Heterogeneity across Genomic Sites and Replicates Improves Accuracy in the Inference of Transcription Factor Binding." *PlOS ONE*, **10**(9), e0138030. doi:10.1371/journal.pone.0138030.

Raydan M, Svaiter BF (2002). "Relaxed Steepest Descent and Cauchy-Barzilai-Borwein Method." *Computational Optimization and Applications*, **21**(2), 155–167. doi:10.1023/a:1013708715892.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Roland C, Varadhan R (2005). "New Iterative Schemes for Nonlinear Fixed Point Problems, with Applications to Problems with Bifurcations and Incomplete-Data Problems." *Applied Numerical Mathematics*, **55**(2), 215–226. doi:10.1016/j.apnum.2005.02.006.

Rubin DB, Thayer DT (1982). "EM Algorithms for ML Factor Analysis." *Psychometrika*, **47**(1), 69–76. doi:10.1007/bf02293851.

Saad Y (2003). *Iterative Methods For Sparse Linear Systems*. SIAM.

Shiraishi Y, Tremmel G, Miyano S, Stephens M (2015). "A Simple Model-Based Approach to Inferring and Visualizing Cancer Mutation Signatures." *PLOS Genetics*, **11**(12), e1005657. doi:10.1371/journal.pgen.1005657.

Varadhan R (2020). ***SQUAREM**: Squared Extrapolation Methods for Accelerating EM-Like Monotone Algorithms*. R package version 2020.1, URL https://CRAN.R-project.org/package=SQUAREM.

Varadhan R, Roland C (2008). "Simple and Globally Convergent Methods for Accelerating the Convergence of Any EM Algorithm." *Scandinavian Journal of Statistics*, **35**(2), 335–353. doi:10.1111/j.1467-9469.2007.00585.x.

Wellner JA, Zhan Y (1997). "A Hybrid Algorithm for Computation of the Nonparametric Maximum Likelihood Estimator from Censored Data." *Journal of the American Statistical Association*, **92**(439), 945–959. doi:10.1080/01621459.1997.10474049.

Wu CFJ (1983). "On the Convergence Properties of the EM Algorithm." *The Annals of Statistics*, **11**(1), 95–103. doi:10.1214/aos/1176346060.

Zhou H, Alexander D, Lange K (2011). "A Quasi-Newton Acceleration for High-Dimensional Optimization Algorithms." *Statistics and Computing*, **21**(2), 261–273. doi:10.1007/s11222-009-9166-3.

Zhou H, Zhang Y (2012). "EM vs. MM: A Case Study." *Computational Statistics & Data Analysis*, **56**(12), 3909–3920. doi:10.1016/j.csda.2012.05.018.

A. Additional example

A.1. Factor analysis

Factor analysis is a statistical modeling approach that aims to explain the variability among observed variables in terms of a smaller set of unobserved factors. Factor analysis is widely applied in areas where observed variables may be conceptualized as manifesting from some unobserved latent factors, such as psychometrics, behavioral sciences, social sciences, and marketing. The latent factors can be regarded as missing data in a multivariate normal model and the EM algorithm (Dempster *et al.* 1977), therefore, becomes a natural way to compute the maximum likelihood estimates. We will illustrate using two examples the dramatic accelerations of EM by Squarem and also compare with ECME (Liu and Rubin 1998), an acceleration of EM. One example comes from real data, as used by Liu and Rubin (1998) and Rubin and Thayer (1982), while the other is a simulation example.

Notations

Following the notation in Rubin and Thayer (1982), let Y be the $n \times p$ observed data matrix and Z be the $n \times q$ unobserved factor matrix where $q \leq p$. $(Y_i, Z_i), i = 1, 2, \ldots, n$ are independently and identically distributed vectors following multivariate normal distribution. The marginal distribution of $Z_i, i = 1, 2, \ldots, n$ is such that

$$Z_{i_{q\times 1}} \sim \text{ multivariate normal} \left(\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}_{q\times 1}, R_{q\times q} \right).$$

Let the variance of each component of $Z_i, i = 1, 2, \ldots, n$ be 1, so R is also the correlation matrix for Z_i. The factor analysis model assumes that given the factors Z_i, the components of vector Y_i become independent and $Y_{i_{p\times 1}} | Z_{i_{q\times 1}} \sim$ multivariate normal$\left(\alpha_{p\times 1} + \beta_{q\times p}^\top Z_{i_{q\times 1}}, \tau^2_{p\times p}\right)$ where $\tau^2 = \text{diag}\{\tau_1^2, \tau_2^2, \ldots, \tau_p^2\}$. $\beta_{q\times p}$ is called factor loading matrix while the diagonal variances in τ^2 are called uniquenesses in factor analysis. In general, maximum likelihood factor analysis involves estimating α, β, τ^2 and R. But R is often considered to be the identity matrix (orthogonal factor model) and the maximum likelihood estimator of α is always $\bar{Y}$, the column means of the observed data matrix Y. Suppose we center matrix Y by its column means, α is always the zero vector. Therefore, we are left with only β, τ^2 to estimate. Given β, τ^2, the marginal distribution of $Y_i, i = 1, 2, \ldots, n$ is multivariate normal with mean zero vector and covariance matrix $\tau^2 + \beta^\top \beta$. Thus we can write the log-likelihood of the observed data matrix Y,

$$\ell(\tau^2, \beta) = -\frac{n}{2} \log |\tau^2 + \beta^\top \beta| - \frac{n}{2} \text{tr}[C_{yy}(\tau^2 + \beta^\top \beta)^{-1}],$$

where C_{yy} is the sample covariance of Y. The negative log-likelihood to be minimized is coded in function `factor.loglik()`, to check monotonicity in the Squarem algorithm.

EM algorithm

For derivation of the EM step, see Appendix B.2.

If the loading matrix β is unrestricted, the EM update is such that

$$\begin{aligned}\beta^{(k+1)} &= (\delta^\top C_{yy}\delta + \Delta)^{-1}(C_{yy}\delta)^\top,\\ \tau^{2(k+1)} &= \text{diag}\{C_{yy} - C_{yy}\delta(\delta^\top C_{yy}\delta + \Delta)^{-1}(C_{yy}\delta)^\top\},\end{aligned}$$

where $\beta^{(k+1)}, \tau^{2(k+1)}$ are the estimates at the $(k+1)$th iteration and δ, Δ are defined in Appendix B.2.

Similarly, if the loading matrix β has a priori zeroes, the EM update is such that

$$\begin{aligned}\beta_{1j}^{(k+1)} &= (\delta^\top C_{yy}\delta + \Delta)_{1j}{}^{-1}(C_{yy}\delta)_{1j}^\top,\\ \tau_j^{2(k+1)} &= C_{yyj} - (C_{yy}\delta)_{1j}(\delta^\top C_{yy}\delta + \Delta)_{1j}{}^{-1}(C_{yy}\delta)_{1j}^\top,\end{aligned}$$

where j refers to the jth variable in vector $Y_i, i = 1, 2, \ldots, n$, subscript $1j$ corresponds to the factors with nonzero loadings for the jth variable and C_{yyj} is the jth diagonal element of C_{yy}. Next we consider two data examples to illustrate the simplicity and stability of Squarem to accelerate the EM algorithm.

Real data example

We use the real data from Jöreskog (1967) as in Rubin and Thayer (1982) and Liu and Rubin (1998). The data set consists of 9 variables, 4 factors, and 2 patterns of a priori zeroes for the loadings with one a priori zero loadings on factor 4 for variables 1 through 4, and a different a priori zero loadings on factor 3 for variables 5–9. There are otherwise no restrictions. The sample covariance matrix C_{yy} is given below:

$$C_{yy} = \begin{pmatrix} 1.0 & 0.554 & 0.227 & 0.189 & 0.461 & 0.506 & 0.408 & 0.280 & 0.241\\ & 1.0 & 0.296 & 0.219 & 0.479 & 0.530 & 0.425 & 0.311 & 0.311\\ & & 1.0 & 0.769 & 0.237 & 0.243 & 0.304 & 0.718 & 0.730\\ & & & 1.0 & 0.212 & 0.226 & 0.291 & 0.681 & 0.661\\ & & & & 1.0 & 0.520 & 0.514 & 0.313 & 0.245\\ & & & & & 1.0 & 0.473 & 0.348 & 0.290\\ & & & & & & 1.0 & 0.374 & 0.306\\ & & & & & & & 1.0 & 0.672\\ & & & & & & & & 1.0 \end{pmatrix}.$$

We use the starting values of β and τ^2 as in Liu and Rubin (1998), where

$$\beta^{\text{start}\,\top} = \begin{pmatrix} 0.5954912 & -0.4893347 & -0.3848925 & 0.0000000\\ 0.6449102 & -0.4408213 & -0.3555598 & 0.0000000\\ 0.7630006 & 0.5053083 & -0.0535340 & 0.0000000\\ 0.7163828 & 0.5258722 & 0.0219100 & 0.0000000\\ 0.6175647 & -0.4714808 & 0.0000000 & 0.1931459\\ 0.6464100 & -0.4628659 & 0.0000000 & 0.4606456\\ 0.6452737 & -0.3260013 & 0.0000000 & -0.3622682\\ 0.7868222 & 0.3690580 & 0.0000000 & 0.0630371\\ 0.7482302 & 0.4326963 & 0.0000000 & 0.0431256 \end{pmatrix},$$

and $\tau_j^{2\text{start}} = 10^{-8}$ for $j = 1, 2, \ldots, 9$.

The negative log-likelihood is given by the function `factor.loglik()` below, which corresponds to the argument `objfn` in `squarem()`.

```
R> factor.loglik <- function(param, cyy) {
+    beta.vec <- param[1:36]
+    beta.mat <- matrix(beta.vec, 4, 9)
+    tau2 <- param[37:45]
+    tau2.mat <- diag(tau2)
+    Sig <- tau2.mat + t(beta.mat) %*% beta.mat
+    loglik <- -145/2 * log(det(Sig)) - 145/2 * sum(diag(solve(Sig, cyy)))
+    return(-loglik)
+  }
```

One EM update is given by the function `factor.em()` below, which corresponds to the argument `fixptfn` in `squarem()`.

```
R> factor.em <- function(param, cyy) {
+    beta.vec <- param[1:36]
+    beta.mat <- matrix(beta.vec, 4, 9)
+    tau2 <- param[37:45]
+    tau2.mat <- diag(tau2)
+    inv.quantity <- solve(tau2.mat + t(beta.mat) %*% beta.mat)
+    small.delta <- inv.quantity %*% t(beta.mat)
+    big.delta <- diag(4) - beta.mat %*% inv.quantity %*% t(beta.mat)
+    cyy.inverse <- t(small.delta) %*% cyy %*% small.delta + big.delta
+    cyy.mat <- t(small.delta) %*% cyy
+    beta.new <- matrix(0, 4, 9)
+    beta.p1 <- solve(cyy.inverse[1:3, 1:3]) %*% cyy.mat[1:3, 1:4]
+    beta.p2 <- solve(cyy.inverse[c(1, 2, 4), c(1, 2, 4)]) %*%
+      cyy.mat[c(1, 2, 4), 5:9]
+    beta.new[1:3, 1:4] <- beta.p1
+    beta.new[c(1, 2, 4), 5:9] <- beta.p2
+    tau.p1 <- diag(cyy)[1:4] - diag(t(cyy.mat[1:3, 1:4]) %*%
+      solve(cyy.inverse[1:3, 1:3]) %*% cyy.mat[1:3, 1:4])
+    tau.p2 <- diag(cyy)[5:9] - diag(t(cyy.mat[c(1, 2, 4), 5:9]) %*%
+      solve(cyy.inverse[c(1, 2, 4), c(1, 2, 4)]) %*%
+      cyy.mat[c(1, 2, 4), 5:9])
+    tau.new <- c(tau.p1, tau.p2)
+    param.new <- c(as.numeric(beta.new), tau.new)
+    param <- param.new
+    return(param.new)
+  }
```

In order to compare with the ECME algorithm as implemented by Liu and Rubin (1998), we also write the function `factor.ecme()` below to do one ECME iteration. The only difference from the EM algorithm is that for M step, after we update the loading matrix β, we find τ^2 that maximizes the actual constrained likelihood of the observed data matrix Y given the updated β using Newton-Raphson.

```
R> factor.ecme <- function(param, cyy) {
+    n <- 145
+    beta.vec <- param[1:36]
+    beta.mat <- matrix(beta.vec, 4, 9)
+    tau2 <- param[37:45]
+    tau2.mat <- diag(tau2)
+    inv.quantity <- solve(tau2.mat + t(beta.mat) %*% beta.mat)
+    small.delta <- inv.quantity %*% t(beta.mat)
+    big.delta <- diag(4) - beta.mat %*% inv.quantity %*% t(beta.mat)
+    cyy.inverse <- t(small.delta) %*% cyy %*% small.delta + big.delta
+    cyy.mat <- t(small.delta) %*% cyy
+    beta.new <- matrix(0, 4, 9)
+    beta.p1 <- solve(cyy.inverse[1:3, 1:3]) %*% cyy.mat[1:3, 1:4]
+    beta.p2 <- solve(cyy.inverse[c(1, 2, 4), c(1, 2, 4)]) %*%
+      cyy.mat[c(1, 2, 4), 5:9]
+    beta.new[1:3, 1:4] <- beta.p1
+    beta.new[c(1, 2, 4), 5:9] <- beta.p2
+    A <- solve(tau2.mat + t(beta.new) %*% beta.new)
+    sum.B <- A %*% (n * cyy) %*% A
+    gradient <- - tau2/2 * (diag(n*A) - diag(sum.B))
+    hessian <- (0.5 * (tau2 %*% t(tau2))) * (A * (n * A - 2 * sum.B))
+    diag(hessian) <- diag(hessian) + gradient
+    U <- log(tau2)
+    U <- U - solve(hessian, gradient)
+    tau.new <- exp(U)
+    param.new <- c(as.numeric(beta.new), tau.new)
+    param <- param.new
+    return(param.new)
+  }
```

Next we use R package **SQUAREM** to compute the MLE by EM, Squarem, ECME, and squared ECME algorithms. Tolerance is set to be 10^{-8} across all algorithms.

EM: In order to perform the EM algorithm, we use function `fpiter()` in the **SQUAREM** package. The arguments consist of a starting value, the function `factor.em()` that encodes one EM update, the negative log-likelihood function `factor.loglik()`, other variables as needed by these functions, and a control list to specify changes to default values. The starting values for β and τ^2 are taken from Liu and Rubin (1998).

```
R> library("SQUAREM")
R> system.time(f1 <- fpiter(par = param.start, cyy = cyy,
+    fixptfn = factor.em, objfn = factor.loglik,
+    control = list(tol = 10^(-8), maxiter = 20000)))

 user  system elapsed
2.805   0.028   2.834

R> f1$fpevals
```

```
[1] 14659
```

It takes 14659 iterations to converge for the EM algorithm, which spends 2.834 seconds.

ECME: We replace function `factor.em()` by `factor.ecme()` that implements one ECME update and thus implement the ECME algorithm.

```
R> system.time(f2 <- fpiter(par = param.start, cyy = cyy,
+    fixptfn = factor.ecme, objfn = factor.loglik,
+    control = list(tol = 10^(-8), maxiter = 20000)))

 user  system elapsed
1.378   0.029   1.409

R> f2$fpevals

[1] 6408
```

It takes 6408 iterations of ECME updates to converge, less than half of what the EM needs. Also it spends 1.409 seconds, approximately half of the time it takes for the EM to converge.

Squarem: Next, we use function `squarem()` in the **SQUAREM** package to apply the Squarem algorithm to accelerate the EM. The arguments are the same as in function `fpiter()` except a few control parameters particularly set for the Squarem algorithm.

```
R> system.time(f3 <- squarem(par = param.start, cyy = cyy,
+    fixptfn = factor.em, objfn = factor.loglik,
+    control = list(tol = 10^(-8))))

 user  system elapsed
0.226   0.006   0.233

R> f3$fpevals

[1] 876
```

It only takes 876 iterations of EM updates to converge, which is faster by a factor of 17 and 7 in terms of the number of fixed point evaluations when compared to the EM and ECME, respectively. Moreover, Squarem only uses 0.233 seconds to converge, 12 times faster than EM and 6 times faster than ECME.

Squared ECME: The Squarem algorithm can even be used to accelerate ECME, which is already a faster version of the EM algorithm. Let us call this squared ECME.

```
R> system.time(f4 <- squarem(par = param.start, cyy = cyy,
+    fixptfn = factor.ecme, objfn = factor.loglik,
+    control = list(tol = 10^(-8))))

 user  system elapsed
0.111   0.005   0.117
```

	EM	ECME	Squarem	Squared ECME
CPU time	2.799 (0.116)	1.382 (0.046)	0.221 (0.012)	0.107 (0.005)
Number of EM iterations	14659	6408	876	400

Table 7: The comparison of EM, ECME, Squarem and squared ECME algorithms on real data from Jöreskog (1967). The CPU time is in seconds.

```
R> f4$fpevals

[1] 400
```

The squared ECME converges in only 400 iterations compared to 6408 iterations for ECME, and it takes only 0.117 seconds.

In order to accommodate the randomness of CPU time, we run the above 4 schemes 100 times and summarize the mean and standard deviation of CPU running time in Table 7, along with the number of fixed point evaluations needed. Table 7 demonstrates that the Squarem algorithm can greatly and easily improve on both EM and ECME algorithms for the factor analysis problem.

Simulation example

In the simulation example, we generate 200 observations of 32 subject scores and we assume that there are 4 latent factors. In this case, we do not impose any a priori zero loadings for convenience of comparison. The function used to generate data and compare the performance of EM to Squarem is coded into `simulate.FAEM()`, accessible from the replication script.

The results of comparison of CPU running time and the number of EM evaluations between EM algorithm and Squarem are summarized in Figure 5. It can be seen that Squarem performs consistently better than the EM algorithm for both criteria, by a factor of at least 10 in most cases.

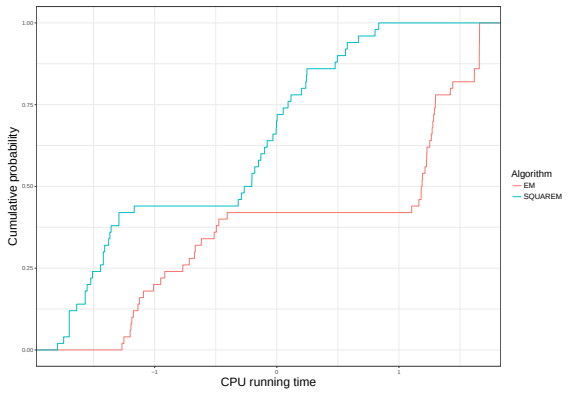
B. Derivation of EM

B.1. Poisson mixtures

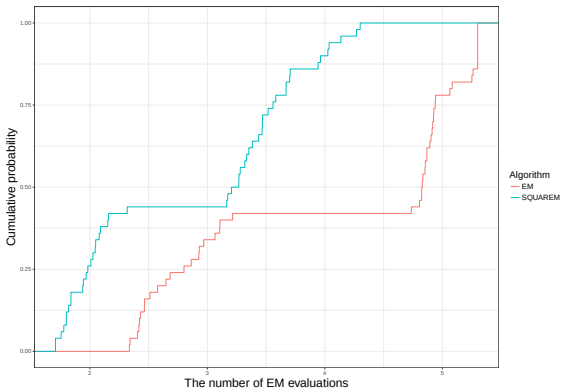
Let us define the missing variable Z_i, $i = 0, 1, \ldots, 9$ such that

$$Z_i = \begin{cases} 1 & \text{if death number } i \text{ comes from population 1,} \\ 0 & \text{if death number } i \text{ comes from population 2.} \end{cases}$$

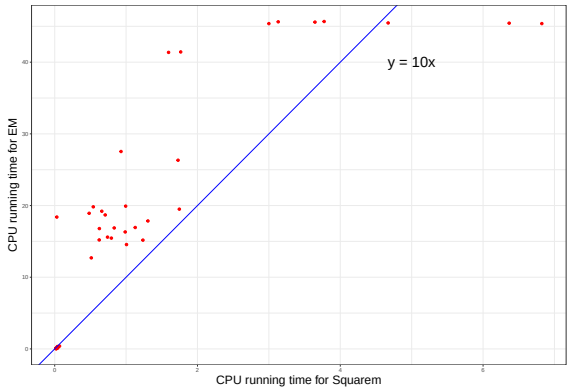
Let $\mathsf{P}(Z_i = 1) = p$, $i = 0, 1, \ldots, 9$. Given $Z_i = 1$, the death number i comes from population 1, following a Poisson distribution with mean μ_1 while otherwise, the death number i comes from population 2, following a Poisson distribution with mean μ_2.



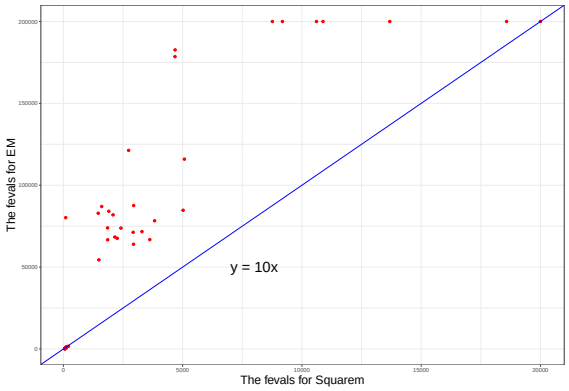
(a) The cumulative distribution function of CPU running time for the EM algorithm versus Squarem.



(b) The cumulative distribution function of the number of EM evaluations (log10 scale) for the EM algorithm versus Squarem.



(c) The scatter plot of CPU running time for the EM algorithm versus Squarem.



(d) The scatter plot of the number of EM evaluations for the EM algorithm versus Squarem.

Figure 5: The comparison of CPU running time and the number of EM evaluations between the EM algorithm and Squarem.

E step:

$$
\begin{aligned}
&Q(p, \mu_1, \mu_2 | p^{(k)}, \mu_1^{(k)}, \mu_2^{(k)}) \\
&\quad = \mathsf{E}[\log [\prod_i \mathsf{P}(i, n_i, Z_i)] | p^{(k)}, \mu_1^{(k)}, \mu_2^{(k)}, i, n_i] \\
&\quad = \mathsf{E}[\log [\prod_i ((p e^{-\mu_1} \mu_1^i / i!)^{Z_i} ((1-p) e^{-\mu_2} \mu_2^i / i!)^{1-Z_i})^{n_i}] | p^{(k)}, \mu_1^{(k)}, \mu_2^{(k)}, i, n_i] \\
&\quad = \sum_i n_i [p_i^{(k)} (\log p + \log (e^{-\mu_1} \mu_1^i / i!)) + (1 - p_i^{(k)}) (\log (1-p) + \log (e^{-\mu_2} \mu_2^i / i!))],
\end{aligned}
$$

where

$$
\begin{aligned}
p_i^{(k)} &= \mathsf{E}[Z_i|p^{(k)},\mu_1^{(k)},\mu_2^{(k)},i,n_i] \\
&= \mathsf{P}(Z_i=1|p^{(k)},\mu_1^{(k)},\mu_2^{(k)},i,n_i) \\
&= \frac{\mathsf{P}(i|Z_i=1,\mu_1^{(k)})\mathsf{P}(Z_i=1|p^{(k)})}{\mathsf{P}(i|Z_i=1,\mu_1^{(k)})\mathsf{P}(Z_i=1|p^{(k)})+\mathsf{P}(i|Z_i=0,\mu_2^{(k)})\mathsf{P}(Z_i=0|p^{(k)})} \\
&= \frac{p^{(k)}e^{-\mu_1^{(k)}}(\mu_1^{(k)})^i/i!}{p^{(k)}e^{-\mu_1^{(k)}}(\mu_1^{(k)})^i/i!+(1-p^{(k)})e^{-\mu_2^{(k)}}(\mu_2^{(k)})^i/i!}
\end{aligned}
$$

M step: We take the derivative of the Q function with respect to p,μ_1,μ_2 and set it to zero in order to derive the estimates of the $(k+1)$th iteration.

Let

$$\frac{dQ}{dp}=\frac{\sum_i n_i p_i^{(k)}}{p}-\frac{\sum_i n_i(1-p_i^{(k)})}{1-p}=0$$

So,

$$
\begin{aligned}
\frac{\sum_i n_i p_i^{(k)}}{\sum_i n_i-\sum_i n_i p_i^{(k)}} &= \frac{p}{1-p} \\
p^{(k+1)} &= \frac{\sum_i n_i p_i^{(k)}}{\sum_i n_i}.
\end{aligned}
$$

Let

$$\frac{dQ}{d\mu_1}=\sum_i n_i p_i^{(k)}(-1+\frac{i}{\mu_1})=0$$

So,

$$
\begin{aligned}
\sum_i n_i p_i^{(k)} &= \frac{\sum_i i n_i p_i^{(k)}}{\mu_1} \\
\mu_1^{(k+1)} &= \frac{\sum_i i n_i p_i^{(k)}}{\sum_i n_i p_i^{(k)}}.
\end{aligned}
$$

Similarly we can derive that

$$\mu_2^{(k+1)}=\frac{\sum_i i n_i(1-p_i^{(k)})}{\sum_i n_i(1-p_i^{(k)})}.$$

B.2. Factor analysis

E step:

$$
\begin{aligned}
&Q(\beta, \tau^2 | \beta^{(k)}, \tau^{(k)}) \\
&= \mathsf{E}[\log \mathsf{P}(Y, Z) | Y, \beta^{(k)}, \tau^{(k)}] = \sum_{i=1}^{n} \mathsf{E}[\log \mathsf{P}(Y_i, Z_i) | Y_i, \beta^{(k)}, \tau^{(k)}] \\
&= \sum_{i=1}^{n} \mathsf{E}[\log [\mathsf{P}(Y_i | Z_i) \mathsf{P}(Z_i)] | Y_i, \beta^{(k)}, \tau^{(k)}] \\
&= \sum_{i=1}^{n} \mathsf{E}[\log \{ [(2\pi)^{p/2} |\tau^2|^{-1/2} \\
&\qquad \exp \{ -\frac{1}{2} (Y_i - \beta^\top Z_i)^\top \tau^{2-1} (Y_i - \beta^\top Z_i) \} \mathsf{P}(Z_i)] \} | Y_i, \beta^{(k)}, \tau^{(k)}] \\
&= C - \frac{n}{2} \log |\tau^2| - \sum_{i=1}^{n} \mathsf{E}[\frac{1}{2} Y_i^\top \tau^{2-1} Y_i - Y_i^\top \tau^{2-1} \beta^\top Z_i + \frac{1}{2} Z_i^\top \beta \tau^{2-1} \beta^\top Z_i | Y_i, \beta^{(k)}, \tau^{(k)}] \\
&= C - \frac{n}{2} \log |\tau^2| - \sum_{i=1}^{n} (\frac{1}{2} Y_i^\top \tau^{2-1} Y_i - Y_i^\top \tau^{2-1} \beta^\top \mathsf{E}[Z_i | Y_i, \beta^{(k)}, \tau^{(k)}] \\
&\qquad + \frac{1}{2} \mathrm{tr} \{ \beta \tau^{2-1} \beta^\top \mathsf{E}[Z_i Z_i^\top | Y_i, \beta^{(k)}, \tau^{(k)}] \}),
\end{aligned}
$$

where C is a constant that does not depend on parameters and k denotes the current estimates of parameters.

Let us define $C_{yz} = \sum_1^n \frac{Y_i Z_i^\top}{n}$, $C_{zz} = \sum_1^n \frac{Z_i Z_i^\top}{n}$. So the expected value in the E step depends on conditional expectations of the statistics C_{yy}, C_{yz} and C_{zz} given the observed data matrix Y and current estimates of $\tau^{2(k)}, \beta^{(k)}$. Let $\delta = (\tau^{2(k)} + \beta^{(k)\top} \beta^{(k)})^{-1} (\beta^{(k)\top})$, so by the multivariate normal conditional distribution, $\mathsf{E}[Z_i | Y_i, \beta^{(k)}, \tau^{2(k)}] = \delta^\top Y_i$. Let $\Delta = \mathsf{VAR}[Z_i | Y_i, \beta^{(k)}, \tau^{(k)}] = I - \beta^{(k)} (\tau^{2(k)} + \beta^{(k)\top} \beta^{(k)})^{-1} \beta^{(k)\top}$.

Therefore,

$$
\begin{aligned}
\mathsf{E}[C_{yy} | Y, \tau^{2(k)}, \beta^{(k)}] &= C_{yy} \\
\mathsf{E}[C_{yz} | Y, \tau^{2(k)}, \beta^{(k)}] &= \sum_{i=1}^{n} \frac{\mathsf{E}[Y_i Z_i^\top | Y_i, \beta^{(k)}, \tau^{2(k)}]}{n} = \sum_{i=1}^{n} \frac{Y_i Y_i^\top}{n} \delta = C_{yy} \delta \\
\mathsf{E}[C_{zz} | Y, \tau^{2(k)}, \beta^{(k)}] &= \sum_{i=1}^{n} \frac{\mathsf{E}[Z_i Z_i^\top | Y_i, \beta^{(k)}, \tau^{2(k)}]}{n} \\
&= \sum_{i=1}^{n} \frac{\mathsf{VAR}[Z_i | Y_i, \beta^{(k)}, \tau^{(k)}] + \mathsf{E}[Z_i | Y_i, \beta^{(k)}, \tau^{2(k)}] \mathsf{E}[Z_i^\top | Y_i, \beta^{(k)}, \tau^{2(k)}]}{n} \\
&= \sum_{i=1}^{n} \frac{\Delta + \delta^\top Y_i Y_i^\top \delta}{n} = \delta^\top \frac{\sum_{i=1}^n Y_i Y_i^\top}{n} \delta + \Delta = \delta^\top C_{yy} \delta + \Delta.
\end{aligned}
$$

M step: If the loading matrix β is unrestricted:

In order to obtain the maximizer of the Q function in the E step, we take the derivative of the Q function with respect to $\beta, \tau^{2^{-1}}$ (for convenience) and set it to zero.

$$\begin{aligned}\frac{dQ(\beta,\tau^2|\beta^{(k)},\tau^{(k)})}{d\beta} &= \sum_{i=1}^{n}\tau^{2^{-1}}Y_i\mathsf{E}[Z_i^\top|Y_i,\beta^{(k)},\tau^{2^{(k)}}] \\ &\quad -\sum_{i=1}^{n}\tau^{2^{-1}}\beta^\top\mathsf{E}[Z_iZ_i^\top|Y_i,\beta^{(k)},\tau^{2^{(k)}}]=0.\end{aligned}$$

So,

$$\begin{aligned}\sum_{i=1}^{n}\tau^{2^{-1}}\beta^\top\mathsf{E}[Z_iZ_i^\top|Y_i,\beta^{(k)},\tau^{2^{(k)}}] &= \sum_{i=1}^{n}\tau^{2^{-1}}Y_i\mathsf{E}[Z_i^\top|Y_i,\beta^{(k)},\tau^{2^{(k)}}] \\ \beta^\top(\sum_{i=1}^{n}\mathsf{E}[Z_iZ_i^\top|Y_i,\beta^{(k)},\tau^{2^{(k)}}]) &= \sum_{i=1}^{n}Y_i\mathsf{E}[Z_i^\top|Y_i,\beta^{(k)},\tau^{2^{(k)}}] \\ \beta^\top\mathsf{E}[C_{zz}|Y_i,\beta^{(k)},\tau^{2^{(k)}}] &= \mathsf{E}[C_{yz}|Y_i,\beta^{(k)},\tau^{2^{(k)}}] \\ \beta^\top(\delta^\top C_{yy}\delta+\Delta) &= C_{yy}\delta \\ \beta^{(k+1)} &= (\delta^\top C_{yy}\delta+\Delta)^{-1}(C_{yy}\delta)^\top.\end{aligned}$$

$$\begin{aligned}\frac{dQ(\beta,\tau^2|\beta^{(k)},\tau^{(k)})}{d\tau^{2^{-1}}} &= \frac{n}{2}\tau^2-\sum_{i=1}^{n}(\frac{1}{2}Y_iY_i^\top-{\beta^{(k+1)}}^\top\mathsf{E}[Z_i|Y_i,\beta^{(k)},\tau^{2^{(k)}}]Y_i^\top \\ &\quad +\frac{1}{2}{\beta^{(k+1)}}^\top\mathsf{E}[Z_iZ_i^\top|Y_i,\beta^{(k)},\tau^{2^{(k)}}]\beta^{(k+1)})=0\end{aligned}$$

So,

$$\begin{aligned}\frac{n}{2}\tau^2 &= \sum_{i=1}^{n}(\frac{1}{2}Y_iY_i^\top-{\beta^{(k+1)}}^\top\mathsf{E}[Z_i|Y_i,\beta^{(k)},\tau^{2^{(k)}}]Y_i^\top \\ &\quad +\frac{1}{2}{\beta^{(k+1)}}^\top\mathsf{E}[Z_iZ_i^\top|Y_i,\beta^{(k)},\tau^{2^{(k)}}]\beta^{(k+1)}) \\ \tau^2 &= \sum_{i=1}^{n}(\frac{Y_iY_i^\top}{n}-2{\beta^{(k+1)}}^\top\mathsf{E}[\frac{Z_iY_i^\top}{n}|Y_i,\beta^{(k)},\tau^{2^{(k)}}] \\ &\quad +{\beta^{(k+1)}}^\top\mathsf{E}[\frac{Z_iZ_i^\top}{n}|Y_i,\beta^{(k)},\tau^{2^{(k)}}]\beta^{(k+1)}) \\ \tau^2 &= C_{yy}-2{\beta^{(k+1)}}^\top\mathsf{E}[C_{yz}^\top|Y_i,\beta^{(k)},\tau^{2^{(k)}}] \\ &\quad +{\beta^{(k+1)}}^\top\mathsf{E}[C_{zz}|Y_i,\beta^{(k)},\tau^{2^{(k)}}]\beta^{(k+1)} \\ \tau^2 &= C_{yy}-2{\beta^{(k+1)}}^\top(C_{yy}\delta)^\top \\ &\quad +{\beta^{(k+1)}}^\top(\delta^\top C_{yy}\delta+\Delta)(\delta^\top C_{yy}\delta+\Delta)^{-1}(C_{yy}\delta)^\top \\ \tau^2 &= C_{yy}-2{\beta^{(k+1)}}^\top(C_{yy}\delta)^\top+{\beta^{(k+1)}}^\top(C_{yy}\delta)^\top \\ \tau^2 &= C_{yy}-{\beta^{(k+1)}}^\top(C_{yy}\delta)^\top \\ \tau^{2^{(k+1)}} &= \mathrm{diag}\{C_{yy}-C_{yy}\delta(\delta^\top C_{yy}\delta+\Delta)^{-1}(C_{yy}\delta)^\top\}.\end{aligned}$$

B.3. Interval censoring

E step: α_{ij} tells us the possibility that the event for individual i can occur in interval (s_{j-1}, s_j), but we do not observe whether it actually occurs. Let us encode this missing information in a new defined variable Z_{ij} such that:

$$Z_{ij} = \begin{cases} 1 & \text{if the event for individual } i \text{ occurs in } (s_{j-1}, s_j), \\ 0 & \text{otherwise.} \end{cases}$$

With this missing variable defined, we now write the Q function in the E step.

$$Q(p|p^{(k)}) = \mathsf{E}[\log(\prod_i \prod_j p_j^{Z_{ij}})|p^{(k)}, \alpha] = \sum_{i=1}^{n} \sum_j \log p_j \mathsf{E}[Z_{ij}|p^{(k)}, \alpha]$$

Thus the Q function depends on the conditional expectation of the missing variable Z_{ij} given the α matrix and current estimates $p^{(k)}$. By the Bayesian formula,

$$\mu_{ij} = \mathsf{E}[Z_{ij}|p^{(k)}, \alpha] = \mathsf{P}[Z_{ij} = 1|p^{(k)}, \alpha] = \frac{\alpha_{ij} p_j}{\sum_s \alpha_{is} p_s}, \quad i = 1, 2, \ldots, n, j = 1, 2, \ldots, m.$$

So,

$$Q(p|p^{(k)}) = \sum_{i=1}^{n} \sum_j \log p_j \mu_{ij}.$$

M step: The probability vector that maximizes the Q function serves as the new estimates $p^{(k+1)}$. We introduce a Lagrange multiplier λ to incorporate the constraint that $\sum_j p_j = 1$. Taking the derivative with respect to p_j,

$$\frac{dQ(p|p^{(k)})}{dp_j} = \frac{d(\sum_{i=1}^{n} \sum_j \log p_j \mu_{ij} + \lambda(1 - \sum_j p_j))}{dp_j} = \sum_{i=1}^{n} \frac{\mu_{ij}}{p_j} - \lambda = 0.$$

So,

$$\begin{aligned} p_j &= \frac{\sum_{i=1}^{n} \mu_{ij}}{\lambda} \\ \sum_j p_j = 1 &= \frac{\sum_{i=1}^{n} \sum_j \mu_{ij}}{\lambda} \\ \lambda &= n \end{aligned}$$

Therefore,

$$\begin{aligned} p_j &= \frac{\sum_{i=1}^{n} \mu_{ij}}{\lambda} \\ p_j^{(k+1)} &= \frac{1}{n} \sum_{i=1}^{n} \mu_{ij}, \quad j = 1, 2, \ldots, m \\ p^{(k+1)} &= (p_1^{(k+1)}, p_2^{(k+1)}, \ldots, p_m^{(k+1)})^\top \end{aligned}$$

B.4. Genetics global ancestry estimation problem

The EM algorithm can be used to compute the maximum likelihood estimates of the matrices F and Q. Let us introduce four missing variables for each individual i and marker j, $u_{ij}^{(\text{pat})}, u_{ij}^{(\text{mat})}, z_{ij}^{(\text{pat})}$ and $z_{ij}^{(\text{mat})}$. $u_{ij}^{(\text{pat})}$ and $u_{ij}^{(\text{mat})}$ represent the unobserved allele 1 count from the paternal chromosome and the maternal chromosome respectively with possible values 0 or 1, while $z_{ij}^{(\text{pat})}$ and $z_{ij}^{(\text{mat})}$ refer to the ancestral populations for the paternal and maternal alleles with possible values $1, \ldots, K$.

E step:

$$\begin{aligned}
&Q(F, Q|F^{(0)}, Q^{(0)}) \\
&\quad = \mathsf{E}[\log\{\prod_i \prod_j \mathsf{P}(x_{ij}, u_{ij}^{(\text{pat})}, u_{ij}^{(\text{mat})}, z_{ij}^{(\text{pat})}, z_{ij}^{(\text{mat})}|X, F^{(0)}, Q^{(0)})\}] \\
&\quad = \mathsf{E}[\log\{\prod_i \prod_j (\mathsf{P}(x_{ij}|u_{ij}^{(\text{pat})}, u_{ij}^{(\text{mat})}) \prod_k \prod_a (q_{ik} f_{jk}^{I(u_{ij}^a=1)} \\
&\qquad (1-f_{jk})^{I(u_{ij}^a=0)})^{I(z_{ij}^a=k)})\}|X, F^{(0)}, Q^{(0)}] \\
&\quad = \sum_j \sum_k n_{jk}^{(1)} \log f_{jk} + n_{jk}^{(0)} \log(1-f_{jk}) + \sum_{i=1}^n \sum_k m_{ik} \log q_i k + C',
\end{aligned}$$

where C' is a constant that does not contain the parameters of interest,

$$n_{jk}^{(u)} = \sum_{i=1}^n \sum_a \mathsf{P}(z_{ij}^{(a)} = k, u_{ij}^{(a)} = u|X, F^{(0)}, Q^{(0)}), \quad u = 0, 1,$$

and

$$m_{ik} = \sum_j \sum_a \mathsf{P}(z_{ij}^{(a)} = k|X, F^{(0)}, Q^{(0)}).$$

To compute $n_{jk}^{(u)}$, $u = 0, 1$ and m_{ik}, we need to compute the joint posterior probabilities:

$$\begin{aligned}
&\mathsf{P}(u_{ij}^{(\text{pat})}, u_{ij}^{(\text{mat})}, z_{ij}^{(\text{pat})}, z_{ij}^{(\text{mat})}|f_{ij}, q_{ik}, x_{ij}) \\
&\quad \propto \mathsf{P}(x_{ij}|u_{ij}^{(\text{pat})}, u_{ij}^{(\text{mat})}) \prod_k \prod_a (q_{ik} f_{jk}^{I(u_{ij}^a=1)} (1-f_{jk})^{I(u_{ij}^a=0)})^{I(z_{ij}^a=k)} \\
&\quad = I(x_{ij} = u_{ij}^{(\text{pat})} + u_{ij}^{(\text{mat})}) \prod_k \prod_a (q_{ik} f_{jk}^{I(u_{ij}^a=1)} (1-f_{jk})^{I(u_{ij}^a=0)})^{I(z_{ij}^a=k)}.
\end{aligned}$$

Thus, $n_{jk}^{(u)}$, $u = 0, 1$ and m_{ik} for the tth, $t = 0, 1, \ldots$, iteration are computed by summing the joint posterior probabilities using $F^{(t)}$ and $Q^{(t)}$.

M step: Taking the derivative of the Q function with respect to f_{jk} and q_{ik} gives us the matrices F and Q of the next iteration,

$$f_{jk} = \frac{n_{jk}^{(1)}}{n_{jk}^{(1)} + n_{jk}^{(0)}}, \qquad q_{ik} = \frac{m_{ik}}{\sum_k m_{ik}}.$$

Affiliation:

Yu Du
Department of Biostatistics
Bloomberg School of Public Health
Johns Hopkins University
Baltimore, United States of America
E-mail: ydu10@jhu.edu
URL: http://www.biostat.jhsph.edu/~ydu/personal/

Ravi Varadhan
Division of Biostatistics and Bioinformatics
Department of Oncology
Johns Hopkins University
Baltimore, United States of America
E-mail: rvaradhan@jhmi.edu
URL: http://www.hopkinsmedicine.org/profiles/results/directory/profile/0452302/ravi-varadhan

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
February 2020, Volume 92, Issue 7 *Submitted:* 2016-05-23
doi:10.18637/jss.v092.i07 *Accepted:* 2018-10-30