Sparse Matrices for Stan

Daniel Simpson and Aki Vehtari

12 October, 2017

Purpose of Document

The aim of this document is to list some of the features that would be useful Stan that the MxNet crew might also be interested in. I'm going to focus on the "big picture" items first and then some of the "nuts and bolts" implementation details in the next section. Things will be ordered roughly in order of importance (ie how much we want them relative to other things).

Big picture items (Sparse matrix edition)

Adding these items would make, e.g., lmer, glmer, gamm, spline, GP with basis function presentation, Markovian time series, Markovian spatial, and Markovian spatiotemporal type of models **much faster** due to 1) proper sparse matrix operations (even if running with single core) and 2) GPU and CPU parallelization.

Speed up linear mixed models.

Linear mixed models (multilevel models) are of the form

$$y = Z(\theta)\beta + \epsilon,$$

where $\epsilon \sim N(0, \sigma^2 I)$, $\beta \sim N(0, I)$ and $Z(\theta)$ are sparse matrices that may have dense rows that depends on some parameters θ. This is one of the big use cases of Stan.

Currently, these models are *too slow*! In particular, Stan builds a Markov chain for the parameters (θ, σ, β), where the length of β might be as big as the data.

This is wasteful because we can use the conditional Gaussianity of this model to marginalise out the β paratmeters massively reducing the number of paramters. A quick calculation shows that

$$\beta \mid \sigma, \theta, y \sim N([ZZ^T + I]^{-1} Z^T y, [\sigma^{-2} ZZ^T + I]),$$

where I have supppressed the dependence on θ. Noting then that

$$p(\theta, \sigma \mid y) \propto \frac{p(\theta, \beta, \sigma, y)}{p(\beta \mid y, \sigma\, \theta)}$$

(which is true for any β) we get (at $\beta = 0$) something like

$$p(\theta, \sigma \mid y) \propto p(\theta, \sigma) |\sigma^{-2} ZZ^T + I|^{-1/2} \exp\left(-\frac{1}{2\sigma^2} y^T y\right).$$

Now, if Z is sparse and the rows are orderdered correctly, ZZ^T will also be sparse. Covariates will add dense rows and columns. Also Z depends on parameters, so parts of it will need to be re-computed at every step (although we should avoid unnecessarily re-doing data-sized computations).

Implementation note: While I wrote the above in terms of Z, you really only need to specify $Q = ZZ^T$. Depending on the situation, it may be easier to specify Q directly.

Speeding up linear mixed models needs:

- A fast way to build and re-build $Q = ZZ^T$. As Q will typcially have a block structure and only some of these blocks will change at every iteration, we need a fast way to construct block-sparse matrices (where some of the blocks may be dense). There are probably clever ways of doing this that only uses `var`s for the parameter dependent parts of Q. Bob suggested (if I understood it) that we could write $Q = Q_{\text{var}} + Q_{\text{double}}$ and only combine them when we actually need to build the double matrix to evauate things. This is certainly the most memory efficient way to do it, but at least initially it's probably not too much of a sin to make everything a var. The key feature needed to make this work in a "big data" contest is to store expensive operations (like computing $X^T X$ where the columns of X are covariates) in `transformed data` so they only need to be computed once.
- A sparse Cholesky factorisation that reorders the matrix (dense blocks to the end) and factorises it. It isn't possible to store the reordering, so we need to compute it every time. This is used to compute the determinant.
- The derivative of the log determinant.

That last part is the only slightly tricky bit. A quick look at the standard reference for matrix AD (Giles, 2008) and usuing his notation shows what we need to compute. If $C = |Q|$, then the reverse-mode automatic derivative (ie sensitivity) is

$$\bar{Q} = C\bar{C}[Q^{-1}]_{ij \in \mathcal{S}(Q)}.$$

The notation $[A]_{ij \in \mathcal{S}}$ indicates a sparse matrix with sparsity pattern $\mathcal{S}$ where the non-zero elements are populated from the appropriate elements of A.

So we need to compute the elements of $\Sigma = Q^{-1}$ that correspond to the non-zero elements of Q. Rue and Martino (2007, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.1392&rep=rep1&type=pdf) gave recursions for this.

`for` $i = n, \ldots, 1$;

  `for` $j = n, \ldots, i$;

    $\Sigma_{ij} = \delta_{ij}/L_{ii}^2 - \frac{1}{L_{ii}} \sum_{k \in \mathcal{I}(i)} L_{ki}\Sigma_{kj}$;

Here $\mathcal{I}(i) = \{k > i : L_{ki} \neq 0\}$. A *very important* implementation detail is that the condition $L_{ki} \neq 0$ should be interpreted as "L_{ki} not structurally zero". That is, we need to distinguish between elements of L that are structurally zero (unable to be anything else), and elements that just happen to have the value `0.0`. If you don't do this, the recursion may not be solveable!

This point about not dropping zeros is important to Stan more broadly. We need to ensure that the number of parameters is always the same. So if a sparse matrix is a parameter in the model, then we need to ensure that the data structure doesn't drop elements that happen to have the value `0.0` but are allowed other values.

These recursions may give more entries to Σ than we strictly need, so make sure you only output the relevant ones.

Implementing this solution will also allow for models with splines, Markovian spatial random effects, time series components etc. Section 2.4 Rue et al. (2017, https://arxiv.org/pdf/1604.00860.pdf) gives an example of how to construct Q in these cases.

If sparse matrices can be constructed with dense blocks, this will also allow for hierarchical models where one or more component is a GP. I'm uncertain how costly this will be (it might be better to do that with dense matrices rather than sparse.)

Speeding up non-linear mixed models through non-centering

If the data isn't observed with Gaussian noise, we can no longer marginalise out β exactly. But we can still speed things up.

Once again, we need to do things with Q, which enocodes the latent hierarchical structure. Rue et al. (2017, link above) suggest building Q by adding a "fictional" Gaussian random effect with small variance. This is

the equivalent of the way "jitter" is used in GP models, except now it's on the unobserved scale rather than on the data scale.

If the (possibly jittered) Q has a (sparse) Cholesky decomposition $Q = LL^T$, then we can use it to construct a non-centred parameterisation that should help speed up the mixing of the HMC algorithm. The new variable $\zeta = L^T\beta$ is *a priori* $N(0, I)$.

In order to implement this, we need efficient implementations of the derivate of

- the Cholesky of a sparse matrix
- the back-transform $\beta = L^{-T}\zeta$

Both of these can be found in Giles (2008).

Speeding up non-linear mixed models with approximate marginalisation

The above is not a solution that scales to large problems. The only way we can think of to do this is to approximately marginalise out βs. As we saw in the linear mixed model case, we can do this exactly if we know $p(\beta \mid \theta, y)$ (σ has gone into the θ vector now).

The standard way to do this to to approximate this conditional distribution with a Gaussian $p_G(\beta \mid \theta, y)$. There are many options here, but the standard is to build an approximation

$$\log(p_G(\beta \mid \theta, y)) = \text{const} + \frac{1}{2}\log(|Q(\theta) + H(\theta)|) + -\frac{1}{2}[\beta - \beta^*(\theta)]^T[Q(\theta) + H(\theta)][\beta - \beta^*(\theta)]$$

where $\beta^*(\theta)$ is the maximum of $\log(p_G(\beta \mid \theta, y))$ and and $H(\theta)$ is the Hessian of $\log(p_G(\beta \mid \theta, y))$ evaluated at β^*.

While this type of approximation can be computed for almost any model, it's most useful for commonly used exponential family distributions. The main use cases are Poisson, Binomial, Negative Binomial, and Gamma mixed models and these should be specialised.

The key computation needed for doing this is the root finding and the computation of the Hessian. Without exposing higher-order autodiff in the Stan language, I think the best we can do is hand code the analytic derivatives. If computing these derivaties is hard, it's possibly not a good idea to use a Laplace approximation at all! In theory, we could use the existing algebraic solver to do these computations. Some experiments with a simple Poisson random effects model (so Q and H are both diagonal) is throwing up some unexpected problems. (See this thread: http://discourse.mc-stan.org/t/algebraic-sovler-problems-laplace-approximation-in-stan/2172/14)

Summary: - This should be possible to do in Stan 2.17 with hand-coded derivatives and the algebraic solver. But we're having problems making it work in practice.
- Sparse matrices would allow us to consider more complex random effects, but right now we could probably do everything with dense matrices as long as there aren't very many Gaussian parameters to be marginalised out. - It would be good to have a specialised density for certain likelihoods (eg Poisson)

Things that we need implemented in Stan Math (For sparse matrices)

I've listed these by sub-topic. These are ordered so if something is implemented above, it might be used in the subsequent subtopics. This is mainly because these are partially ordered. Everything needs "Sparse Matrix Support". The two "speeding up non-linear models" sections are independent but both depend on "Speeding up linear mixed models". (I'd draw a graph but I'm lazy)

Sparse matrix support

- Sparse constructors including block-sparse constructors and kronecker products. For the latter two, the dream would probably be some sort of "lazy evaluation" version that doesn't build intermediate sparse matrices. But this might be to difficult. Maybe something like MxNet has that capability.
- A full algebra for sparse matrices of doubles. A lot of this can be cribbed from the current matrix operations. This means:
  - addition (returns sparse matrix, possibly with different sparsity structure)
  - sparse matrix transpose (returns sparse matrix, different sparsity structure)
  - sparse matrix-vector multiplication (returns dense vector)
  - sparse matrix-constant multiplication (returns sparse matrix, same sparsity)
  - sparse matrix-matrix multiplication (returns a sparse matrix that likely has a different sparsity structure)
  - sparse matrix-dense matrix multiplication (should return a dense matrix)
  - sparse matrix-diagonal matrix multiplication on the left and right (returns sparse matrix, same sparsity)
  - sparse inner product and quadratic form (returns scalars)
  - I can't think of a use case for sparse matrix plus dense matrix, but it should return a dense matrix.
  - `to_dense()` and `to_sparse` casts.

Speeding up linear mixed models

- Fill-reducing reorderings
- A sparse Cholesky for a matrix of doubles.
- The computation of the log-determinant as the product of Cholesky diagonals
- Sparse linear solves and sparse triangular solves (for sampling from the marginalised out parameters in the `generated quantities` block)
- An implementation of the algorithm to compute the required elements of the inverse.
- An implementation of the reverse mode derivative of $\log(|Q|)$ for sparse Q.

Speeding up non-linear mixed models through non-centering

- Reverse-mode derivative of a Cholesky decomposition of a sparse matrix. This is the same algorithm that was initially used in the dense case before Rob implemented Iain Murray's blocked version. But it will need to be implemented in a "data structure" aware way.
- Reverse mode derivative of $L^{-T}\beta$ for sparse, lower-triangular L. This can be adapted from Giles (2008) with the understanding that if L is sparse $\bar{L}$ is also sparse!

Speeding up non-linear mixed models with approximate marginalisation

- Specialisation for Poisson, binomial and maybe gamma likelihoods.
- It would be useful to also specialise dense versions of these for things like GPs

What is needed in the Stan language

The following is a summary of some discussions I've had with Bob Carpenter, Michael Betancourt, and Aki Vehtari that hopefully give some indication of what needs to be done

A sparse matrix type

There are two things that **must be true** for any implementation of a sparse matrix type in the Stan language: - The number of parameters in the model must stay constant - The ordering of the parameters used internally cannot change.

This can be ensured by requiring that: - A `sparse_matrix` declaration includes the sparsity pattern as well as the number of rows and columns - The order of the rows and columns cannot change. (This doesn't have to be true inside functions as long as the output is consistent with the input.)

From Bob:

```
What needs to happen specifically is that we need a way to map
from a vector of unconstrained values to to a sparse matrix if
we want to use a sparse matrix as a parameter.  That shouldn't
change from iteration to iteration.
```

To help users out, it would be nice to do some shape inference. For example, a sparse matrix declaration could look as follows:

- `sparse_matrix<int rows, int columns, int[] i_vector, int[] j_vector> A;` where the length of `i_vector` and `j_vector` is the number of non zero elements, and `A[m,n] != 0` can only be true if there is an integer `k` such that `i_vector[k]=m` AND `j_vector[k]=n`.
- `sparse_matrix<A + B> C;` where `A` and `B` are previously declared sparse matrices. This declares `C` to be a sparse matrix with the same dimensions and sparsity pattern as `A+B`. If `A+B` is not a legal operation, then this returns an error.
- `sparse_matrix<A * B> C;` where `A` and `B` are previously declared sparse matrices.This declares `C` to be a sparse matrix with the same dimensions and sparsity pattern as `A * B`. If `A * B` is not a legal operation, then this returns an error.
- I am not sure if more complex algebraic expressions should be allowed.

This shape inference can be done easily using the algorithms in, for example, Eigen. Just make `A` and `B` sparse matrices where the non-zero elements are ones and perform the operation.

Helping the sparse Cholesky

The first rule of working with sparse matrices is that you almost never use them with the row/column ordering that they had when you first got them. This is because clever reorderings can give order-of-magintude or more speedup over naive orderings.

Because of the way Stan is designed, we can't "tie" a reordering to a matrix and have it persist through sampling. (So we can't, for example, have a class that stores this ordering for the future). So we have a few options:

- Leave it to the user. This would require the implementation and exposure of sparse matrix reordering algorithms and the user would need to know to use them. One way to do this would be to have a `reordering` argument for every function that uses a sparse Cholesky decomposition internally.
- Implement a reordering internally in the Cholesky factorisation. This requires some care to ensure that it is undone consistently. It also makes it basically impossible to compute the Cholesky and use it later in the Stan code (the way you do with dense matrices) because the reordering wouldn't be outputted.
- Have `sparse_spd_matrix` and `sparse_cholesky` types that make automatically named variables in the C++ code that keep the reordering information. For example have the following code chunk

  ```
  sparse_spde_matrix<rows,cols, i_vector, j_vector> Q;
  sparse_cholesky<Q> L;
  L = chol(Q);
  x = lower_triangle_solve(L,w);
  ```

evaluate to something like (this code is indicative - i'm not working out the details right now)

```
// Compute inner and outer indices to build sparse matrix
Eigen::Map< Eigen::SparseMatrix> Q(Q_inner_index_, Q_outer_index_, 0.0);

std::vector<int> Q_reorder_;
// compute the reordering

Eigen::SparseMatrix L;
std::vector<int> L_reorder_;
// copy  Q_reorder_ to L_reorder_

L = stan::math::cholesky(Q, Q_reorder_);

Eigen::MatrixX1d x = stan::math::lower_triangle_solve(L,w,L_reorder_);
```

Big picture items (Dense matrix edition)

Adding these items would make many user defined models using dense matrices (because it's much simpler to code models with them) and Gaussian process models **faster** due to 1) GPU and CPU parallelization.

Things that we need implemented

Speeding up models with dense matrices

- Parallel (GPU and/or MPI) dense linear algebra
- General matrix algebra (sum, product, kronecker, matrix_divide, determinant) with reverse-mode derivatives
- Cholesky and derivative on GPU and general parallel
- See Stan manual Section 42.2 (Matrix arithmetic operations), 42.13 (Linear Algebra Functions and Solvers)
- Mayne also maybe 42.4 (Elementwise Functions), 42.5(Dot Products and Specialized Products)

Functor for GP specification This is not something needed from MxNet, but if we can do this in Stan in the future, we may not need a wrapper to the MxNet's GP covariance function library.

- Example would be `f ~ GP(function cov_function, tuple params_tuple, vector mean, matrix locations )` for the centred parameterisation.
- For the non-centred parameterisation, you would need the back-transform `f = GP_non_centered_transform(vector non_centered_variable, function cov_function, tuple params_tuple, vector mean, matrix locations )`.
- A `GP_predict()` function that takes the appropriate arguments and produces predictions.
- Implemention would populate the matrix, do the Cholesky and compute the appropriate quantities

For advanced users, it would also be useful to have access to methods that construct matrices - There would also need to be a function for computing the covariance matrix and its cholesky in the event it's fixed.

Can we automate certain GP approximations

It would be nice to be able to do the following things in a way that's less of a burden on users.

- Induced point approximations?

- Automatic marginalising out GP for Gaussian data.
- For Gaussian data, can we have functions for the posterior mean and the posterior covariance (and its diagonal)?