Sparse Matrices
and Large Data Issues

Workshop ENAR
March 15, 2009
Reinhard Furrer, CSM

Outline

What are sparse matrices?
How to work with sparse matrices?
Sparse positive definite matrices in statistics.
Sparse matrices and fields.

Sparse Matrices

What is “sparse” or a sparse matrix?

According to Wiktionary/Wikipedia:

Sparse: (Adjective)
1. Having widely spaced intervals
2. Not dense; meager

Sparse matrix:
A matrix populated primarily with zeros.
Sparse Matrices

R> n <- 15
R> A <- array( runif(n^2), c(n,n)) + diag(n)
R> A[A < 0.75] <- 0

R> AtA <- t(A) %*% A

Why should we use sparse matrices?

1. Savings in storage
   nonzeros vs total elements

2. Savings in computing time
   0.066s vs 0.003 for 1,000 × 1,000 matrix multiplication

To exploit the savings need to exploit the sparsity.

We need a clever storage format and fast algorithms.
Storage Formats

Let \( A = (a_{ij}) \in \mathbb{R}^{n \times m} \) and \( z \) the number of its nonzero elements.

1. Naive/“traditional”/classic format:
   one vector of length \( n \times m \) and a dimension attribute.

2. Triplet format:
   three vectors of length \( z \), \((i, j, a_{ij})\) and a dimension attribute.

3. Compressed sparse row (CSR) format:
   eliminate redundant row indices.

4. and about 10 more . . .

Storage Formats, Example

\[
A = \begin{pmatrix}
1 & 0 & 0.1 & 0 & 0.2 & 0.3 \\
0.4 & 2 & 0 & 0.5 & 0 & 0.6 \\
0 & 0 & 3 & 0 & 0 & 0.7 \\
0.8 & 0 & 4 & 0 & 0 & 0.9 \\
0.1 & 2 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

Naive/traditional/classic:
1, 4, 0, . . , 7, . . , 9, . . , 1, 2, 0, . . , 8, 0, 0, 3, 0, . . , 0, 2, . . , 5, 0, 4, 0, . . , 3, 0, . . , 6, 0, 5

Storage Formats, Example

\[
A = \begin{pmatrix}
1 & 0.1 & 0 & 0.2 & 0.3 \\
0.4 & 2 & 0 & 0.5 & 0 \\
0 & 0 & 3 & 0 & 0.6 \\
0.7 & 0.8 & 0 & 4 & 0 \\
0.9 & 0 & 0 & 0 & 0.5
\end{pmatrix}
\]

Triplet:
\[
i : 1 1 1 2 2 2 3 3 4 4 4 5 5 5 \\
j : 1 2 4 5 1 2 4 2 3 1 2 4 1 3 5 \\
a_{ij} : 1.1 1.2 .3 .4 .2 .5 .3 .6 .7 .8 4 .9 .0 5
\]
Storage Formats, Example

\[ A = \begin{pmatrix}
1 & 0.1 & 0 & 0.2 & 0.3 \\
0.4 & 2 & 0 & 0.5 & 0 \\
0 & 0 & 3 & 0 & 0.6 \\
0.7 & 0.8 & 0 & 4 & 0 \\
0.9 & 0 & 0.0 & 0 & 5
\end{pmatrix} \]

\[ \begin{align*}
i & : 1 \ 2 \ 3 \ 4 \ 5 \\
j & : 1 \ 2 \ 4 \ 5 \ 1 \ 2 \ 4 \ 2 \ 3 \ 1 \ 2 \ 4 \ 1 \ 3 \ 5 \\
a_{ij} & : 1 \ . \ 1 \ . \ 2 \ . \ 3 \ . \ 4 \ 2 \ . \ 5 \ 3 \ . \ 6 \ . \ 7 \ . \ 8 \ . \ 9 \ . \ 0 \ 5
\end{align*} \]

CSR:

\[ \text{ptr} : 1 \ 5 \ 8 \ 10 \ 13 \ 16 \]
\[ j : 1 \ 2 \ 4 \ 5 \ 1 \ 2 \ 4 \ 2 \ 3 \ 1 \ 2 \ 4 \ 1 \ 3 \ 5 \]
\[ a_{ij} : 1 \ . \ 1 \ . \ 2 \ . \ 3 \ . \ 4 \ 2 \ . \ 5 \ 3 \ . \ 6 \ . \ 7 \ . \ 8 \ . \ 9 \ . \ 0 \ 5 \]

Compressed Sparse Row Format

1. the nonzero values row by row
2. the (ordered) column indices of nonzero values
3. the position in the previous two vectors corresponding to new rows, given as pointers
4. the column dimension of the matrix.
(Dis)Advantages

1. Naive format:
   No savings in storage and computation (for sparse matrices)
   Status quo
2. Triplet format:
   Savings in storage and computation for sparse matrices
   Loss in storage and computation for full matrices
   Intuitive
3. Compressed sparse row (CSR) format:
   Apart from intuitive, same as triplet
   Faster element access
   Many available algorithms
   Arbitrary choice for “row” vs “column” format (CSC)

Implications

With a new storage format new “algorithms” are required . . .
Is it worthwhile???

Timing

Setup:

```r
R> timing <- function(expr)
+ as.vector( system.time( for (i in 1:N) expr)[1])
R> N <- 1000 # how many operations
R> n <- 999 # matrix dimension
R> cutoff <- 0.9 # what will be set to 0
R> A <- array( runif(n^2), c(n,n))
R> A[A < cutoff] <- 0
R> S <- somecalltomagicfunctiontogetsparseformat( A)
```

Compare timing for different operations on A and S.
Timing

R> timing( A + sqrt(A) )
[1] 0.058
R> timing( S + sqrt(S) )
[1] 0.061
R> timing( AtA <- t(A) %*% A )
[1] 0.467
R> timing( StS <- t(S) %*% S )
[1] 4.222

Timing

R> timing( A[1,2] <- .5 )
[1] 0.007
R> timing( S[1,2] <- .5 )
[1] 0.018
R> timing( A[n,n-1] <- .5 )
[1] 0.001
R> timing( S[n,n-1] <- .5 )
[1] 0.012

Timing

R> timing( xA <- solve(AtA, rep(1,n)) )
[1] 1.116
R> timing( xS <- solve(StS, rep(1,n)) )
[1] 1.51
R> timing( RA <- chol(AtA) )
[1] 0.488
R> timing( RS <- chol(StS) )
[1] 1.504

Is it really worthwhile? What is going on?
Timing

Matrix S

Matrix StS

Timing

With cutoff 0.99:

R> timing(AtA <- t(A) %*% A )
[1] 0.106
R> timing( StS <- t(S) %*% S )
[1] 0.089
R> timing( RA <- chol(AtA) )
[1] 0.494
R> timing( RS <- chol(StS) )
[1] 0.451

Density of the factor is 41% with fill-in ratio 7.2.
Timing

With cutoff 0.999:

R> timing( AtA <- t(A) %*% A )
[1] 0.059
R> timing( StS <- t(S) %*% S )
[1] 0.002
R> timing( RA <- chol(AtA) )
[1] 0.466
R> timing( RS <- chol(StS) )
[1] 0.007

Matrix StS

Density of the factor is .6% with fill-in ratio 2.3.

Implications

With a new storage format new "algorithms" are required . . .

Is it worthwhile??? Yes!

Especially since

spam: R package for sparse matrix algebra.
What is spam?

- an R package for sparse matrix algebra
  - publicly available from CRAN, 0.15-3
  - platform independent and documented

- storage economical and fast
  - uses “old Yale sparse format”
  - most routines are in Fortran, adapted for spam
  - balance between readability and overhead
  - flags for “expert” users

- versatile, intuitive and simple
  - wrap an `as.spam()` and go
  - S4 and S3 syntax

- situated between SparseM and Matrix

Representation of Sparse Matrices

spam defines a S4 class `spam` containing the vectors: “entries”, “colindices”, “rowpointers” and “dimension”.

```r
R> slotNames( "spam")
[1] "entries"  "colindices"  "rowpointers"  "dimension"
```

```r
R> getSlots( "spam")
entries colindices rowpointers dimension
"numeric" "integer"  "integer"  "integer"
```

```r
R> A
[1,] 1.0 0.1 0 0.2 0.3
[2,] 0.6 2.0 0 0.5 0.0
[3,] 0.0 0.0 3 0.0 0.6
[4,] 0.7 0.8 0 4.0 0.0
[5,] 0.9 0.0 1 0.0 5.0
Class 'spam'
```

```r
R> slotNames(A)
[1] "entries"  "colindices"  "rowpointers"  "dimension"
```

```r
R> A@entries
[1] 1.0 0.1 0.2 0.3 2.0 0.5 3.0 0.6 0.7 0.8 4.0 0.9 1.0 5.0
```

```r
R> A@colindices
[1] 1 2 4 5 1 2 4 3 5 1 2 4 1 3 5
```

```r
R> A@rowpointers
[1] 1 5 8 10 13 16
```

```r
R> A@dimension
[1] 5 5
```
Creating Sparse Matrices

Similar coercion techniques as with matrix:

- spam(...)
- as.spam(...)

Special functions:

- diag.spam(...)
- nearest.dist(...)

Methods for spam

- Similar behavior as with matrices
  plot; dim; determinant; %*%; +; ...

- Slightly enhanced behavior
  print; dim<-; chol;

- Specific behavior
  Math; Math2; Summary; ...

- New methods
  display; ordering;

Create Covariance Matrices

Covariance matrix:

nearest.dist and applying a covariance function:

R> C <- nearest.dist(x, diag=TRUE, upper=NULL)
R> C@entries <- Wendland(C@entries, dim=2, k=1)

Precision matrix (GMRF):

— regular grids: nearest.dist with different cutoffs
  R> diag.spam(n) +
  + (b1-b2) * nearest.dist(x, delta=1, upper=NULL) +
  + b2 * nearest.dist(x, delta=sqrt(2), upper=NULL)

— irregular grids: using incidence list and spam
  R> incidence <- list(i=..., j=..., values)
  R> C <- spam(incidence, n, n)
Solving Linear Systems

A key feature of spam is to solve efficiently linear systems.

To solve the system $Ax = b$, we

- perform a Cholesky factorisation $A = U^TU$
- solve two triangular systems $U^Tz = b$ and $Ux = z$

But we need to “ensure” that $U$ is as sparse as possible!

Permute the rows and columns of $A$: $P^TAP = U^TU$.

---

Cholesky

Some technical details about a Cholesky decomposition:

1. Determine permutation and permute the input matrix $A$ to obtain $P^TAP$
2. Symbolic factorization: the sparsity structure of $U$ is constructed
3. Numeric factorization: the elements of $U$ are computed

---

spam knows Cholesky!

- Several methods to construct permutation matrices $P$
- update to perform only ‘partial’ Cholesky factors
- Flags for avoiding sanity checks
Gain of time and memory usage with different options and arguments in the case of a second order neighbor structure of a regular 50 x 50 grid and of the US counties. The time and memory usage for the generic call chol are 6.2 seconds, 174.5 Mbytes and 15.1 seconds, 416.6 Mbytes, respectively.

<table>
<thead>
<tr>
<th>Options or arguments</th>
<th>Regular grid</th>
<th>US counties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>memory</td>
</tr>
<tr>
<td>Using the specific call chol.sparse</td>
<td>1.001</td>
<td>0.992</td>
</tr>
<tr>
<td>Option safemode=c(FALSE,FALSE,FALSE)</td>
<td>0.961</td>
<td>1.002</td>
</tr>
<tr>
<td>Option cholesymmetrycheck=FALSE</td>
<td>0.568</td>
<td>0.524</td>
</tr>
<tr>
<td>Passing memory=list(nnz=..., nnzcolindices=...)</td>
<td>0.969</td>
<td>0.979</td>
</tr>
<tr>
<td>All of the above</td>
<td>0.561</td>
<td>0.508</td>
</tr>
<tr>
<td>All of the above and passing pivot=... to chol.sparse</td>
<td>0.542</td>
<td>0.528</td>
</tr>
<tr>
<td>All of the above and option cholpivotcheck=FALSE</td>
<td>0.510</td>
<td>0.511</td>
</tr>
<tr>
<td>Numeric update only using update</td>
<td>0.132</td>
<td>0.070</td>
</tr>
</tbody>
</table>
Cholesky

BTW:
efficient Cholesky factorization $\iff$ efficient determinant calculation:

$$
\det(C) = \det(U^T)\det(U) = \prod_{i=1}^{n} U_{ii}^2
$$

Options in spam

For “experts”, flags to speed up the code:

```
R> powerboost() # in spam_0.15-4
```

```
R> noquote( format( spam.options()) )
```

R> noquote( format( spam.options() ) )

<table>
<thead>
<tr>
<th>eps</th>
<th>drop</th>
<th>printsize</th>
<th>imagesize</th>
<th>trivales</th>
<th>cex</th>
<th>10000</th>
<th>FALSE</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>safe mode</td>
<td>dopivoting</td>
<td>cholsymmetrycheck</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUE, TRUE, TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cholpivotecheck</td>
<td>cholupdateesingular</td>
<td>cholincreasefactor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUE</td>
<td>warning</td>
<td>1.25, 1.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nearestdistincreasefactor</td>
<td>nearestdistnnz</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.25</td>
<td>160000, 400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Limits of spam

What can spam not do (yet)?

- LU decompositions
- SVD/eigendecompositions
- Non-double elements
- ...

But, please, comments to rfurrer@mines.edu!
Sparse Matrices in Statistics

Where do large matrices occur?

- Location matrices
- Design matrices
- Covariance matrices
- Precision matrices

Sparse Matrices in Statistics

- Covariance matrices:
  - Compactly supported covariance functions
  - Tapering
- Precision matrices:
  - (Gaussian) Markov random fields
  - (Tapering???)

We have symmetric positive definite (spd) matrices.

Motivation

Precipitation anomaly along 40° lat.
**Motivation**

Precipitation anomaly along 40° lat.

For an isotropic and stationary process with Matérn covariance \( C_0(h) \), find a taper \( C_\theta(h) \), such that kriging with the product \( C_0(h)C_\theta(h) \) is asymptotically optimal.

\[
\frac{\text{MSE}(x^*, C_\theta C_0)}{\text{MSE}(x^*, C_0)} \to 1 \quad \frac{\varrho(x^*, C_\theta C_0)}{\text{MSE}(x^*, C_0)} \to \gamma
\]

\[
\varrho(x^*, C) = C(0) - c^T C^{-1} c^*
\]
Misspecified Covariances

In a series of (Annals) papers, Stein gives asymptotic results for misspecified covariances.

Under appropriate conditions, tapering is a form of misspecification.

The taper has to be

• as differentiable at the origin as the original covariance
• more differentiable throughout the domain than at the origin

Tapering

Tapering is an (asymptotically and computationally) efficient technique to create sparse covariance matrices.

Taper range can be justified by computing resources. However, 20–30 locations within the taper range is often sufficient.

"Classical" tapers are:

• spherical: \( C_\theta(h) = \left(1 - \frac{|h|}{\theta}\right)^2 \left(1 + \frac{|h|}{\theta}\right) \)

• Wendland-type: \( C_\theta(h) = \left(1 - \frac{|h|}{\theta}\right)^{\ell+k} \) \cdot \text{polynomial in } \frac{|h|}{\theta} \text{ of deg } k

Positive Definite Matrices

A (large) covariance (often) appears in:

• drawing from a multivariate normal distribution
• calculating/maximizing the (log-)likelihood
• linear/quadratic discrimination analysis
• PCA, EOF, …

But all boils down to solving a linear system and possibly calculating the determinant …
Sparse Matrices and \texttt{fields} \\

- \texttt{fields} is not bound to a specific sparse matrix format \\
- All heavy lifting is done in \texttt{mKrig} or \texttt{Krig.engine.fixed} \\
- For a specific sparse format, requires the methods: 
  \texttt{chol}, \texttt{backsolve}, \texttt{forwardsolve} and \texttt{diag} 
  as well as elementary matrix operations need to exist \\
- If available uses operators to handle diagonal matrices quickly \\

\textasciitilde \texttt{The covariance matrix has to stem from particular class}. \\
\texttt{fields} uses \texttt{spam} as default package! \\

\textbf{Example mKrig} \\

With appropriate covariance function: 

\begin{verbatim}
R> x <- USprecip[ precipsubset, 1:2] # locations 
R> Y <- USprecip[ precipsubset, 4] # anomaly 
R> out <- mKrig(x,Y, m=1, cov.function="wendland.cov",theta=1.5)

R> out.p <- predict.surface( out, nx=220, ny=120) 
R> surface(out.p, type='l') 
R> us(add=T) 
R> points(x,pch='.')
\end{verbatim}
Example Krig

```r
R> out <- Krig(x,Y, m=1, cov.function="wendland.cov", theta=1.5,
+ lambda=0)
R> out.q <- predict.surface(out, nx=220, ny=120)
R> sum((out.q$z-out.p$z)^2, na.rm=T)
[1] 1.616783e-20
```

Krig/predict is slower (here 2.1/3.7 vs 10.4/3.9 seconds).

Wendland family

`wendland.cov` (based on Wendland) produces a spam matrix. All matrix functions are appropriately overloaded ...

![Covariance plot](image)

To create sparse covariance matrices based on other covariance functions, use `wendland.cov` as skeleton.

Tapering

Tapering can be performed with `stationary.taper.cov`. Arguments are (selection):

- **Covariance** = "Exponential"
- **Taper** = "Wendland"
- **Taper.args** = NULL: arguments for the taper function
- **Dist.args** = NULL: arguments passed to `nearest.dist`
- ...: arguments passed to covariance function

All arguments can also be passed from mKrig/Krig
Tapering

Compare the predicted surfaces without and with tapering:

```r
R> out1 <- mKrig( xr,Yr, m=1, theta=1.5 )
R> out1.p <- predict.surface( out1, nx=220, ny=120 )
R> out2 <- mKrig( xr,Yr, m=1, theta=1.5,
+ cov.function="stationary.taper.cov",
+ Taper.args = list(k=0, theta=3))
R> out2.p <- predict.surface( out2, nx=220, ny=120))
```
(timing yields 4/22 and 1/9 seconds)

Asides . . .

The following arguments of `mKrig/Krig` are linked to `spam`:

- **Dist.args**: arguments passed to `nearest.dist`
- **chol.args**: arguments passed to `chol`

Use their help for fine tuning.

`predict.se.Krig, predict.surface.se.Krig` are very inefficient because `nrow(x)` equations need to be solved.
How Big is Big?

Upper limit to create a large matrix is the minimum of:

1. available memory (machine and OS/shell dependent)
   Error: 'cannot allocate vector of size'
2. addressing capacity \(2^{31} - 1\)
   Error: 'cannot allocate vector of length'

However, R is based on passing by value, calls create local copies (often 3–4 times the space of the object is used).

```r
R> help("Memory-limits")
```

And Beyond?

Parallelization:

nws, snow, Rmpi, ...

Memory “Outsourcing”:
Matrices are not (entirely) kept in memory:

ff, filehash, biglm, ...

(S+ has the library BufferedMatrix)

References

