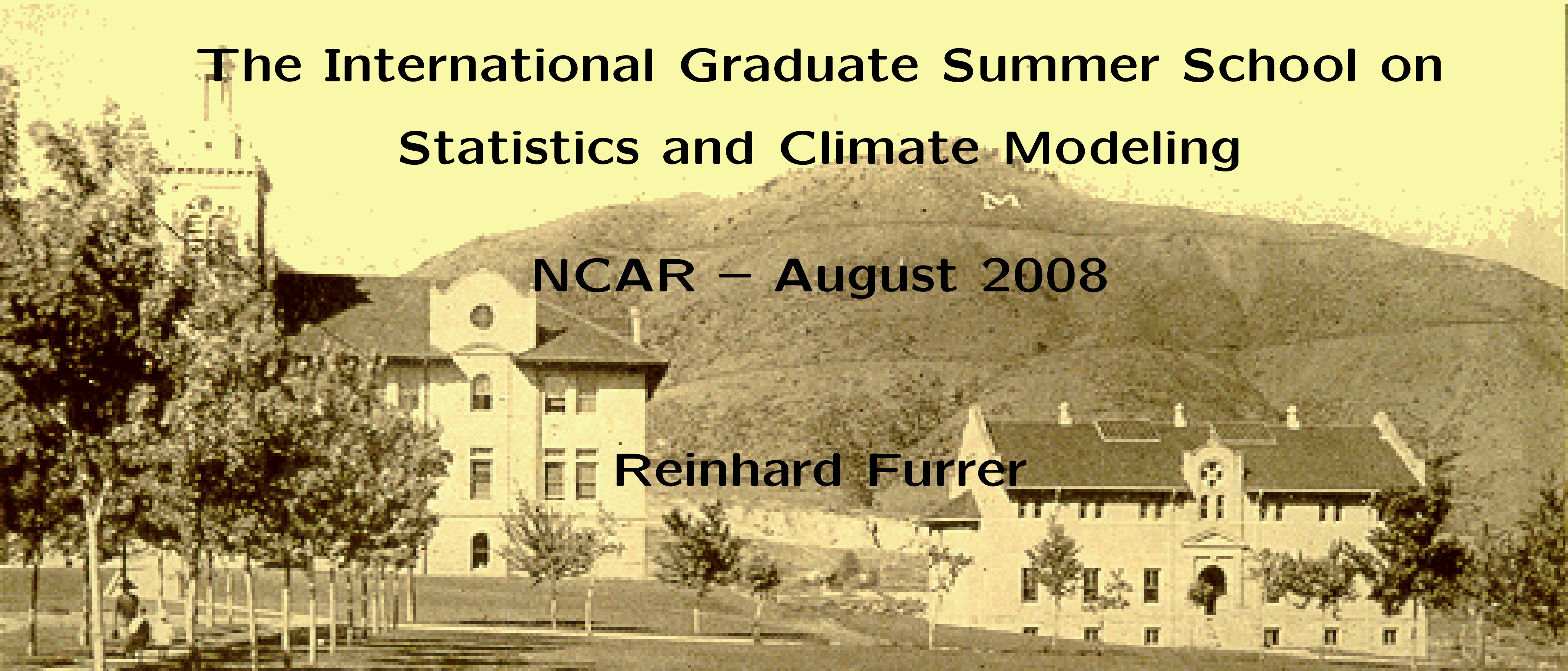
Sparse Matrix Methods and Fields

The International Graduate Summer School on Statistics and Climate Modeling

NCAR – August 2008

Reinhard Furrer



Outline

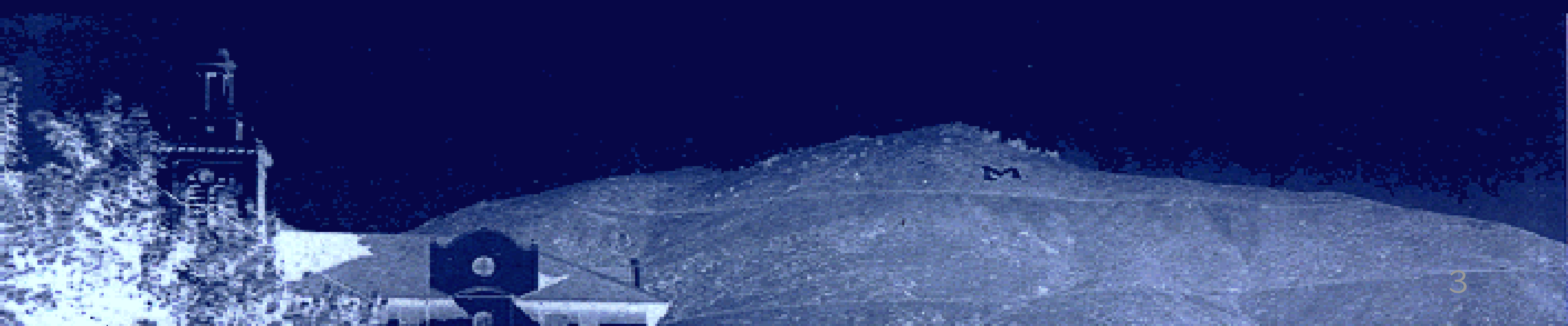
WWW

HTML

URL

Outline

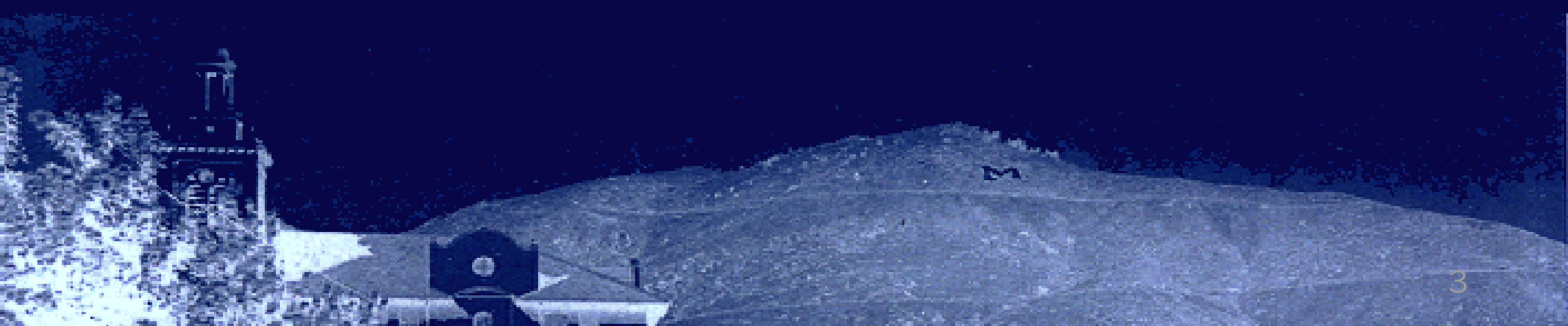
What are sparse matrices?
Why should we use sparse matrices?
What are sparse matrix formats?



Outline

What are sparse matrices?
Why should we use sparse matrices?
What are sparse matrix formats?

What is `spam`?
Sparse matrices in statistics
Solving linear systems
Determinants and Cholesky factorization



Outline

What are sparse matrices?
Why should we use sparse matrices?
What are sparse matrix formats?

What is `spam`?
Sparse matrices in statistics
Solving linear systems
Determinants and Cholesky factorization

`fields` upon `spam`
Finally, real examples
Beyond large sparse matrices

Thanks NSF!
DMS-0621118

Sparse Matrices

What is “sparse” or a sparse matrix?

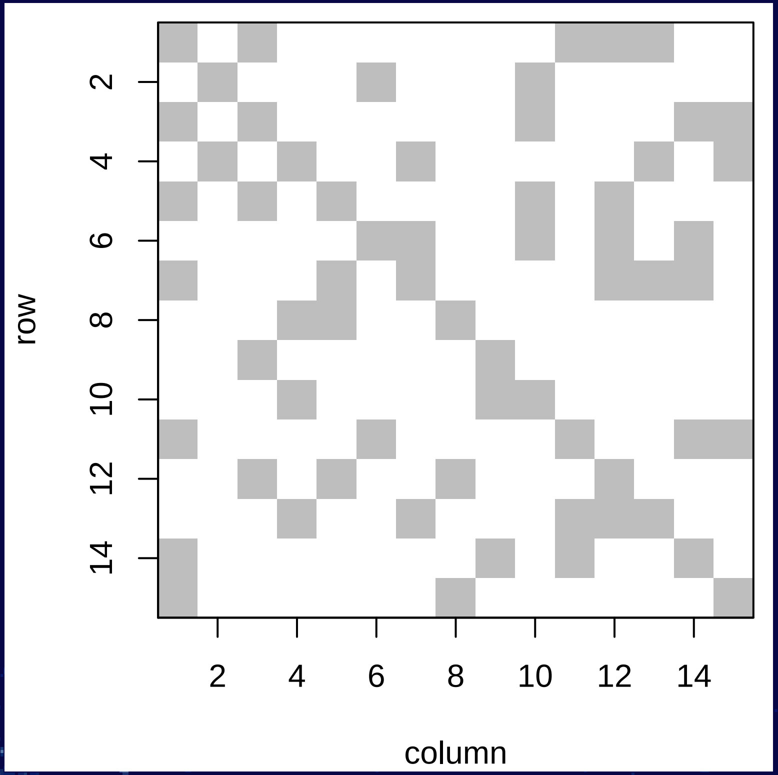
According to Wiktionary/Wikipedia:

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

Sparse matrix:
a matrix populated primarily with zeros.

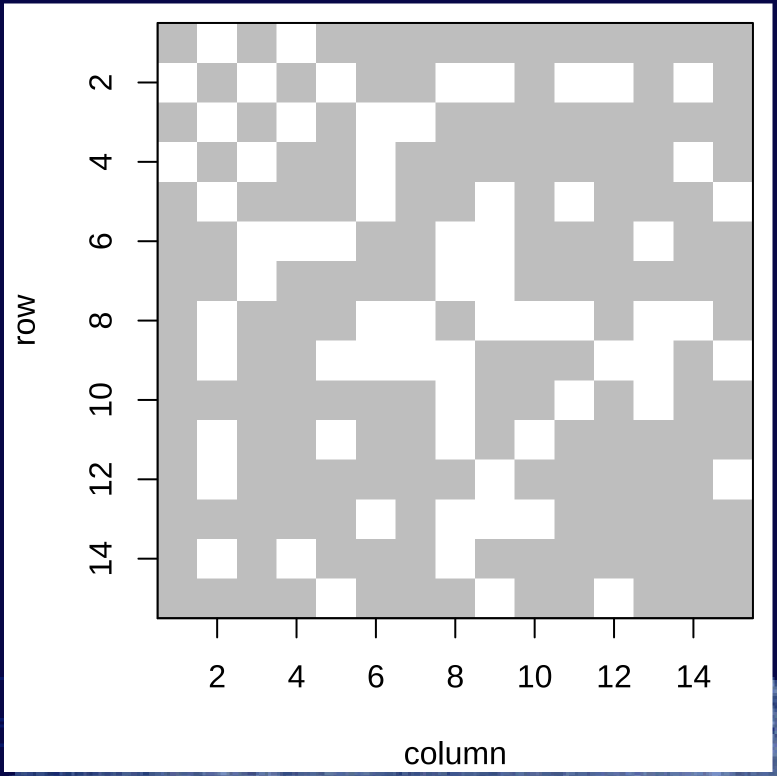
Sparse Matrices

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



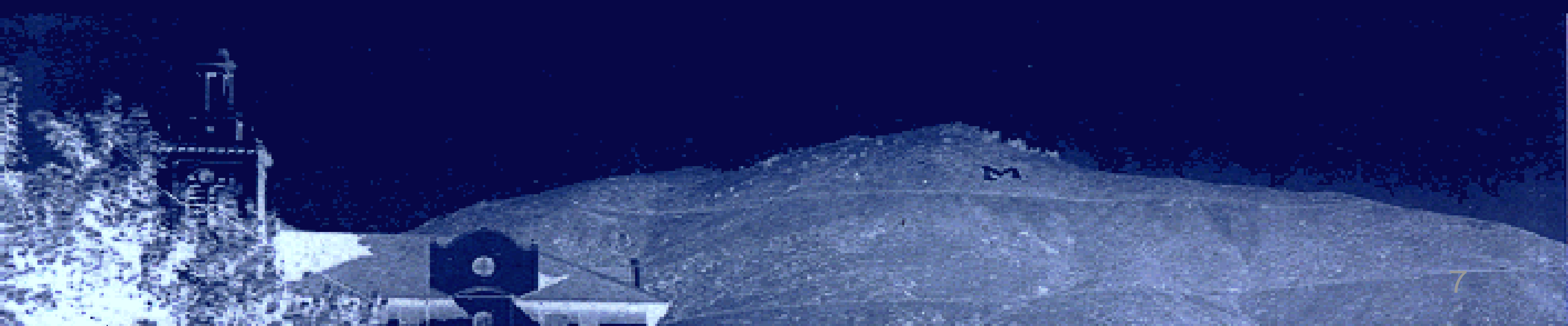
Sparse Matrices

```
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
```



Sparse Matrices

Why should we use sparse matrices?



Sparse Matrices

Why should we use sparse matrices?

1. Savings in storage

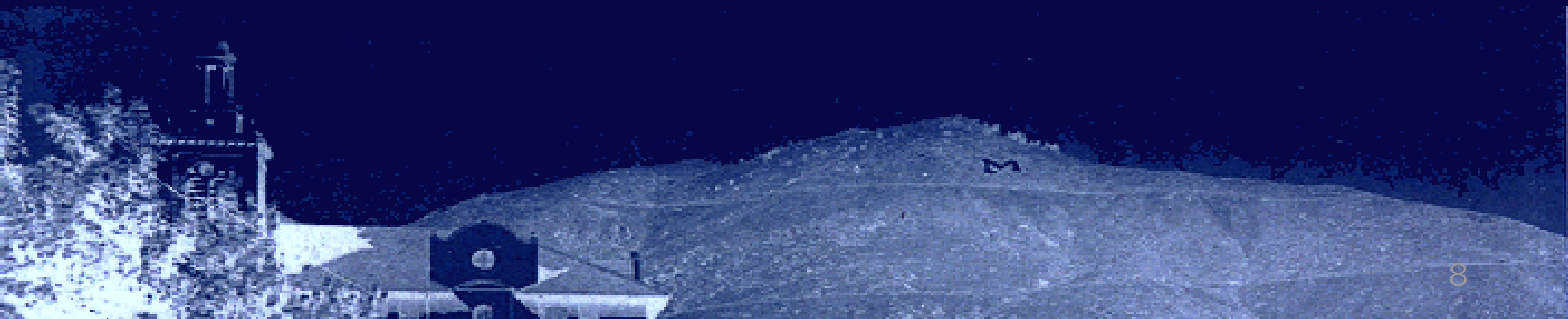
2. Savings in computing time

Sparse Matrices

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 $1,000 \times 1,000$ matrix multiplication



Sparse Matrices

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 $1,000 \times 1,000$ matrix multiplication

To exploit the savings need to exploit the sparsity.

Sparse Matrices

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 $1,000 \times 1,000$ matrix multiplication

To exploit the savings need to exploit the sparsity.

We need a clever storage format and fast algorithms.

Storage Formats

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

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

Storage Formats

Let $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{n \times m}$ 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.

Storage Formats

Let $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{n \times m}$ 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.

Storage Formats

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

1. Naive/ "traditional" /classic format:
   one vector of length $n \times m$ and a dimension attribute.

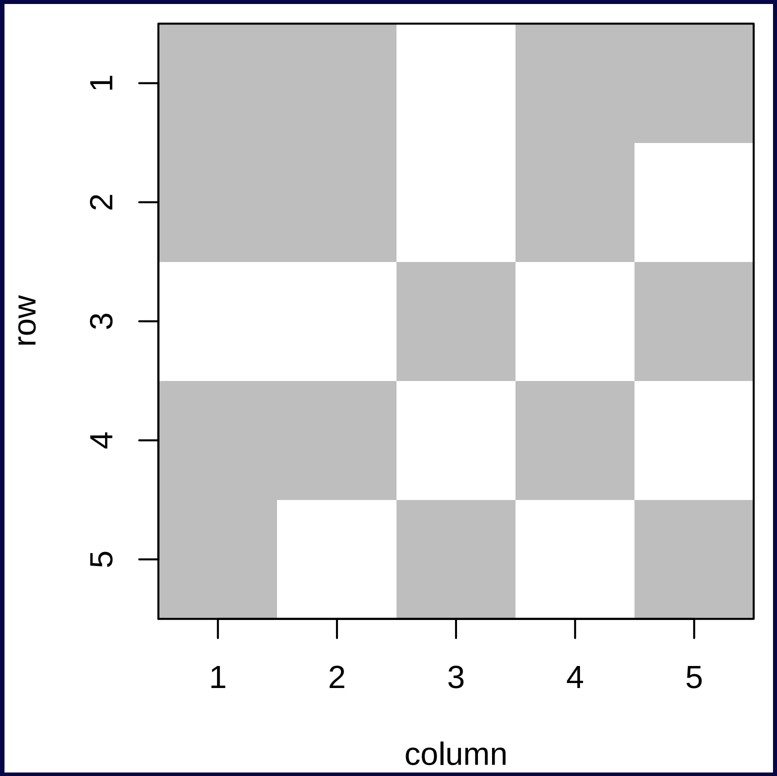
2. Triplet format:
   three vectors of length z, (i, j, a_{ij}) and a dimension attribute.

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

4. and about 10 more . . .

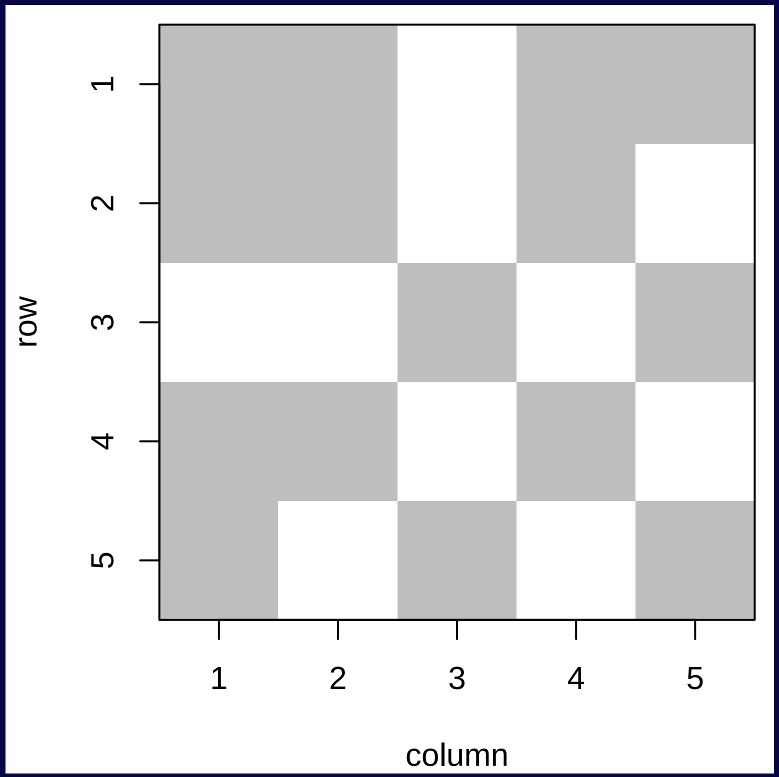
Storage Formats, Example

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



Storage Formats, Example

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

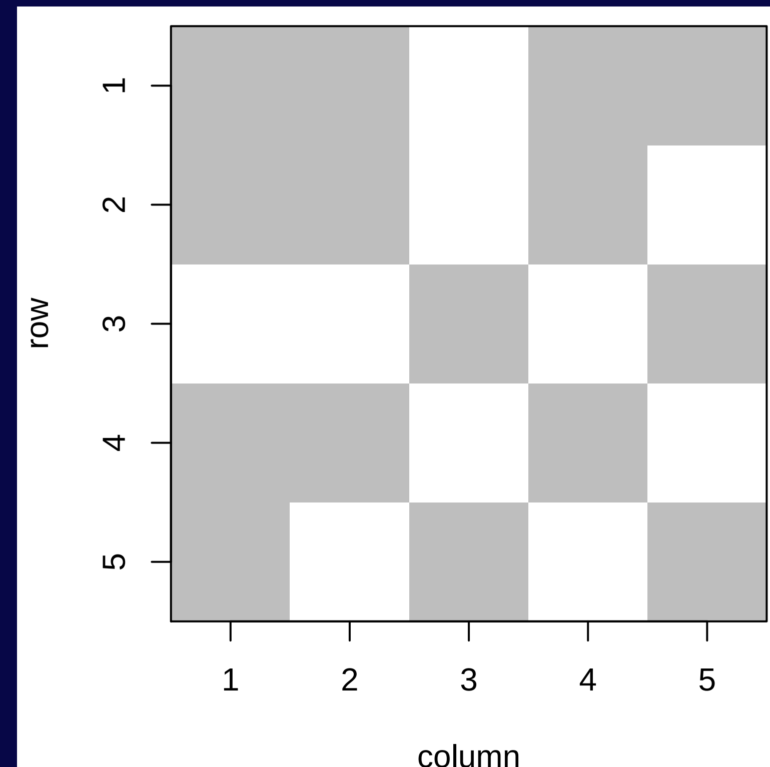


Naive/traditional/classic:

1, .4, 0, .7, .9, .1, 2, 0, .8, 0, 0, 0, 3, 0, .0, .2, .5, 0, 4, 0, .3, 0, .6, 0, 5

Storage Formats, Example

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

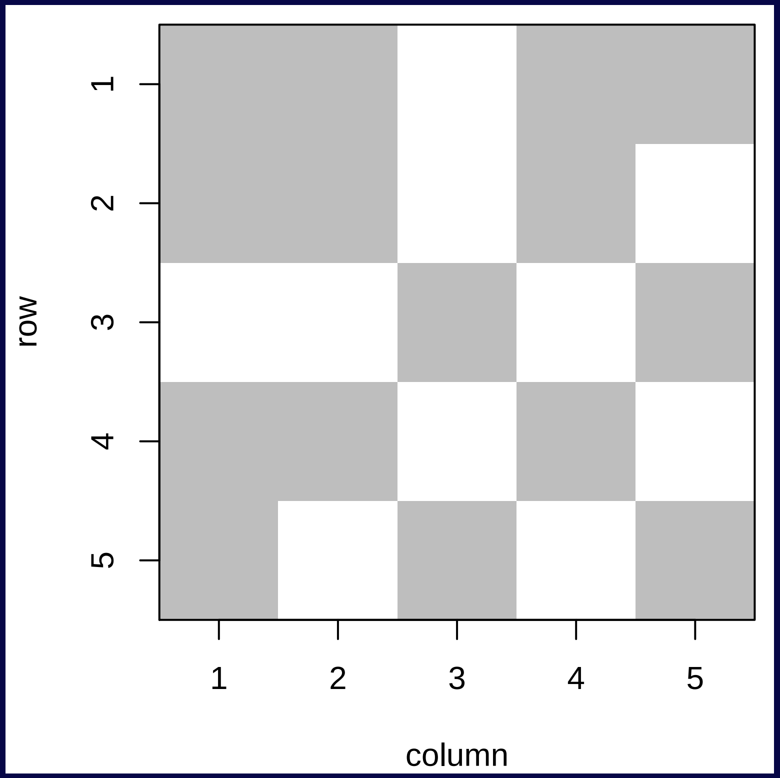


Triplet:

i :	1	1	1	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	.2	.3	.4	2	.5	3	.6	.7	.8	4	.9	.0	5

Storage Formats, Example

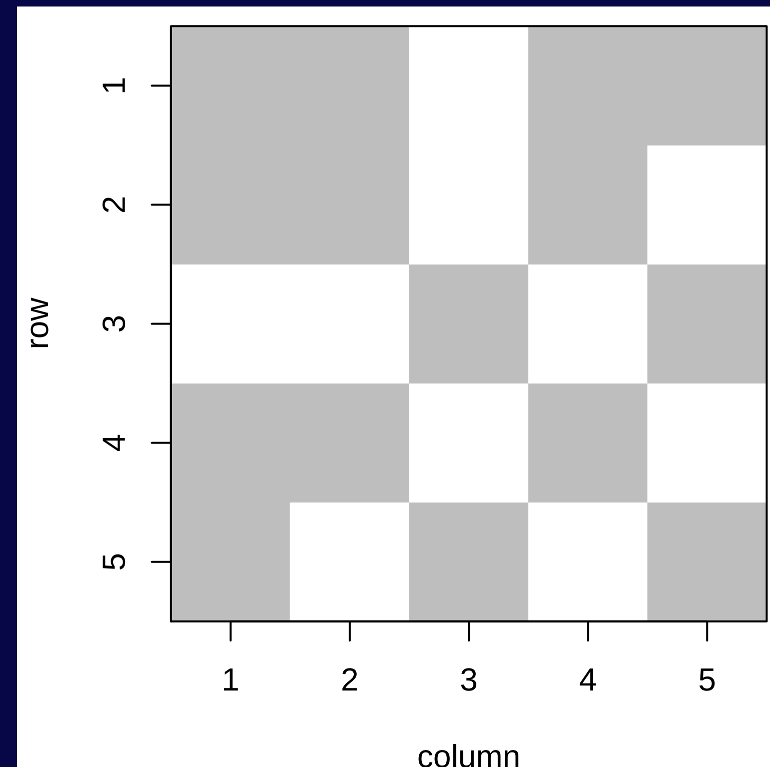
$$A = \begin{pmatrix} 1 & 0.1 & 0 & 0.2 & 0.3 \\ 0.4 & 2 & 0 & 0.5 & 0 \\ 0 & 0 & 3 & 0 & 0.6 \\ 0.7 & 0.8 & 0 & 4 & 0 \\ 0.9 & 0 & 0.0 & 0 & 5 \end{pmatrix}$$



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	4	.9	.0	5

Storage Formats, Example

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



CSR:

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	4	.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.

CSR:

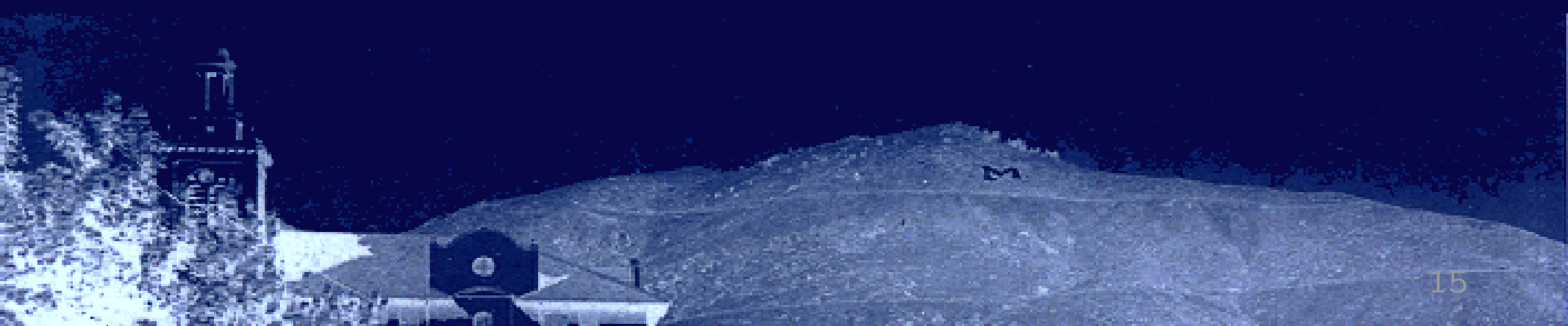
ptr :	1				5			8		10			13			16
j :	1	2	4	5	1	2	4	2	3	1	2	4	1	3	5	
a_{ij} :	1	.1	.2	.3	.4	2	.5	3	.6	.7	.8	4	.9	.0	5	

(Dis)Advantages

1. Naive format:
   No savings in storage and computation (for sparse matrices)
   Status quo

(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



(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

(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
   Available algorithms
   Arbitrary choice for “dimension”

Implications

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

Is it worthwhile???

Timing

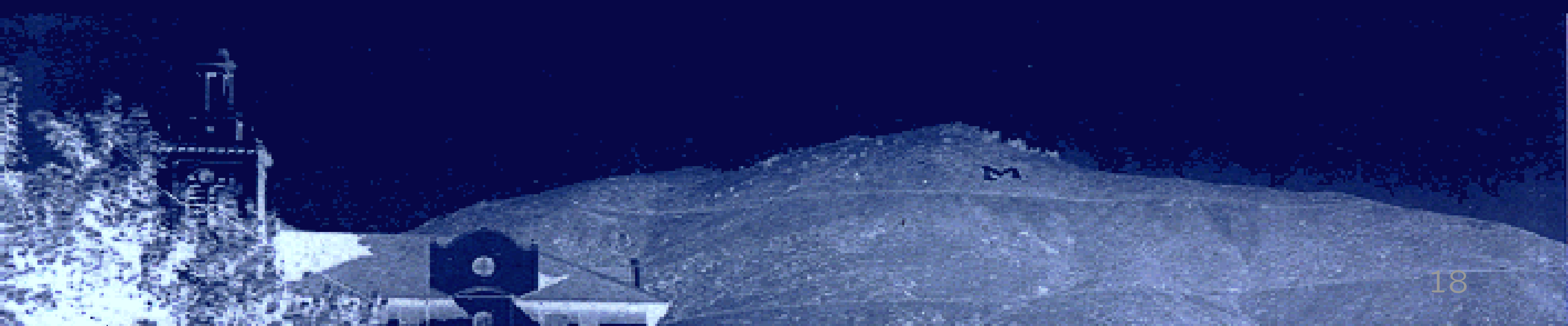
Setup:

```
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`.

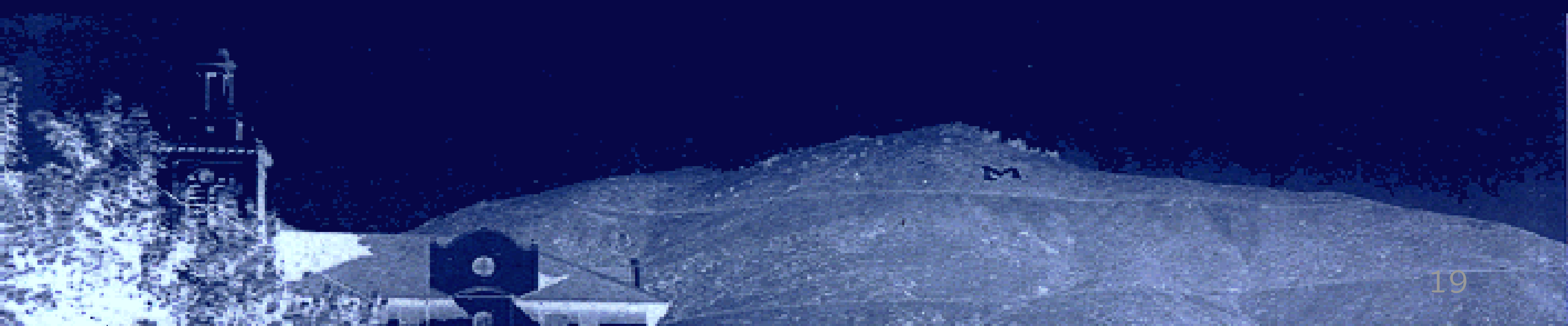
Timing

```
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
```



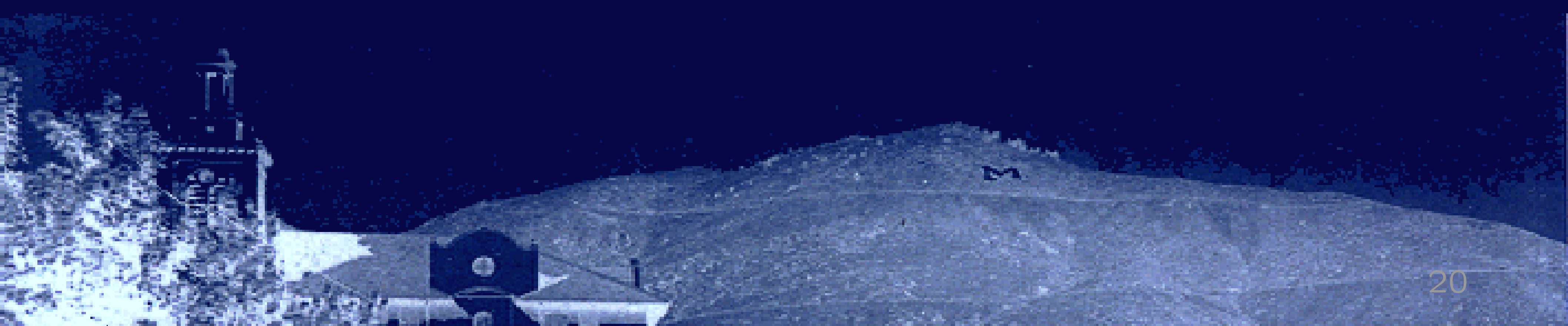
Timing

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

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



Timing

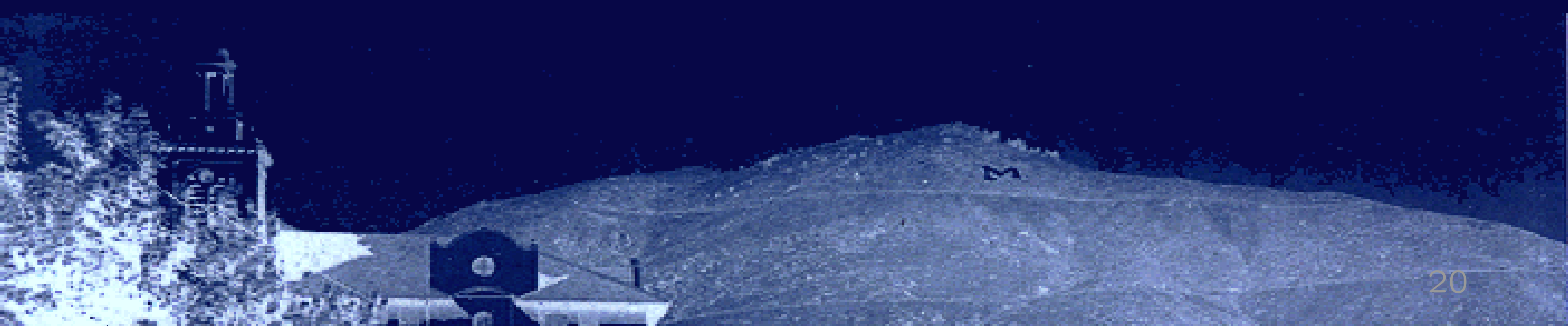
```
R> timing(solve(AtA, rep(1,n)))
[1] 1.116
R> timing(solve(StS, rep(1,n)))
[1] 1.51

R> timing(chol(AtA))
[1] 0.488
R> timing(chol(StS))
[1] 1.504
```



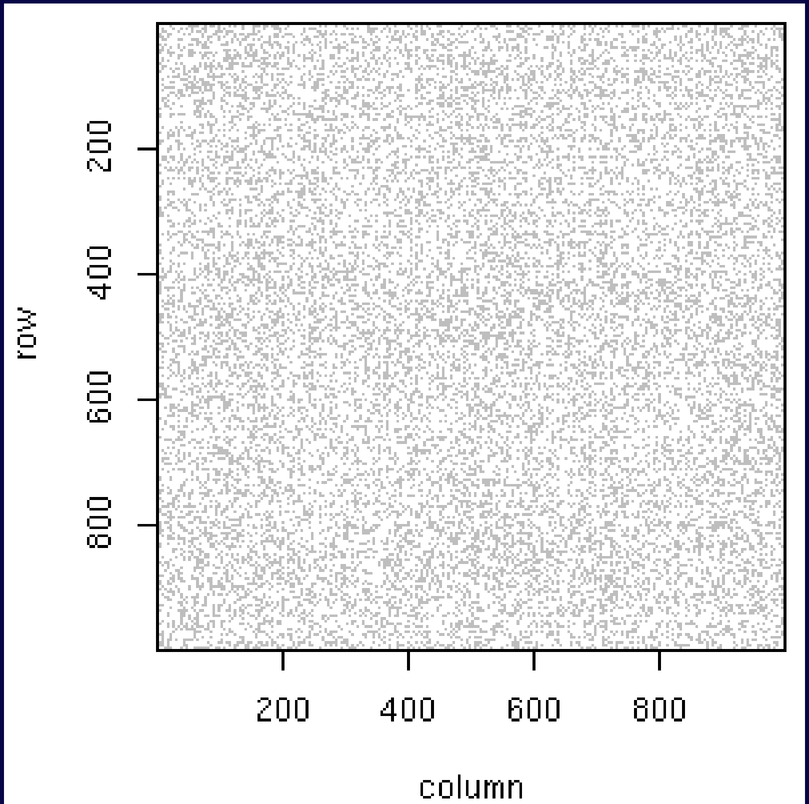
Timing

```
R> timing(solve(AtA, rep(1,n)))
[1] 1.116
R> timing(solve(StS, rep(1,n)))
[1] 1.51

R> timing(chol(AtA))
[1] 0.488
R> timing(chol(StS))
[1] 1.504
```

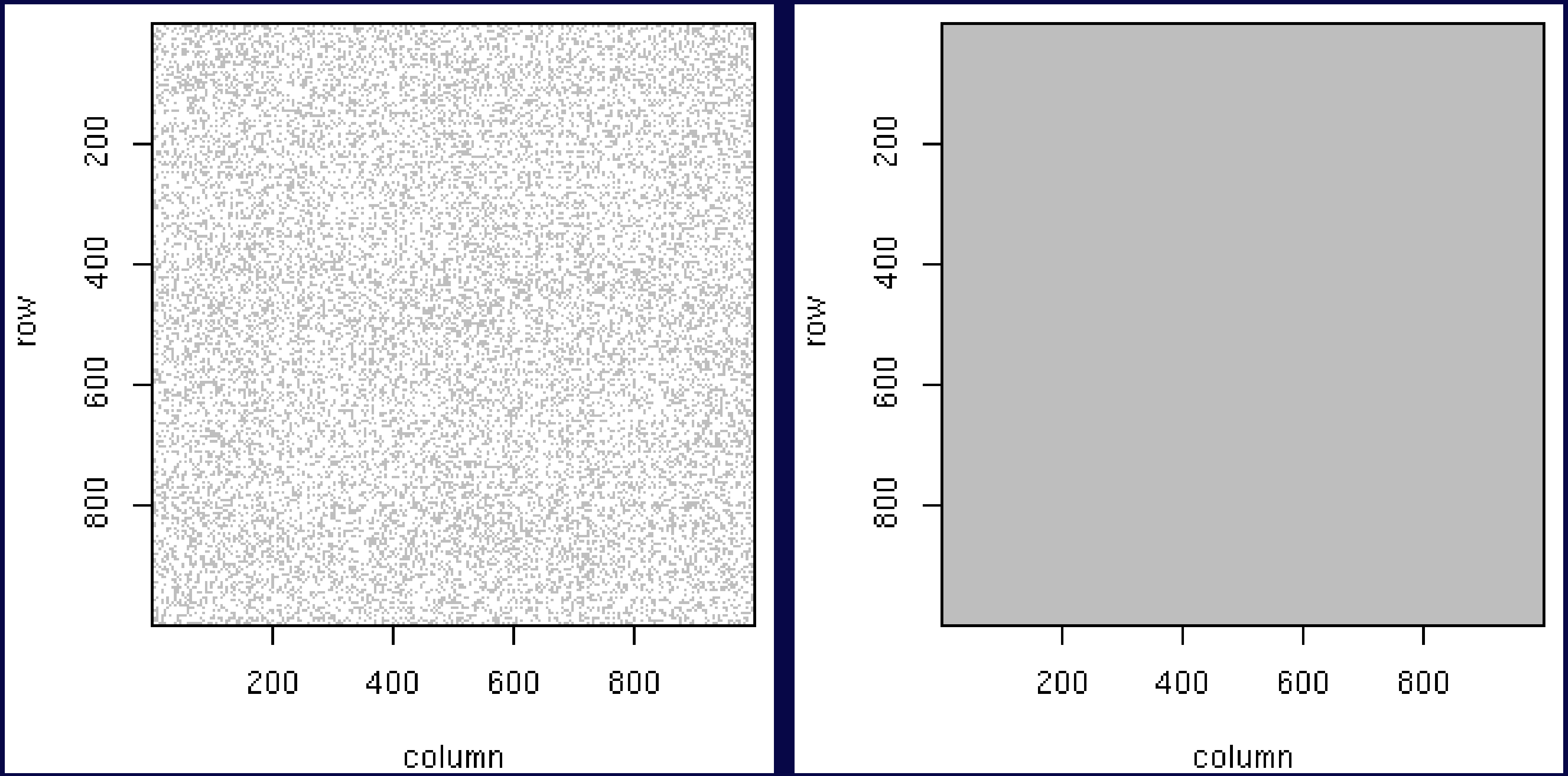
Is it really worthwhile? What is going on?



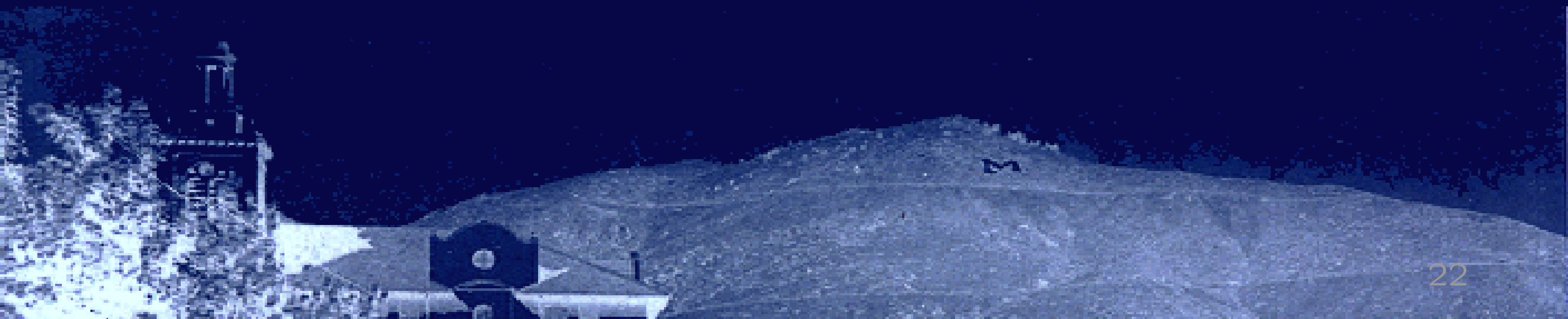
Timing



Timing



row
200
400
600
800
column
200
400
600
800
row
200
400
600
800
column
200
400
600
800

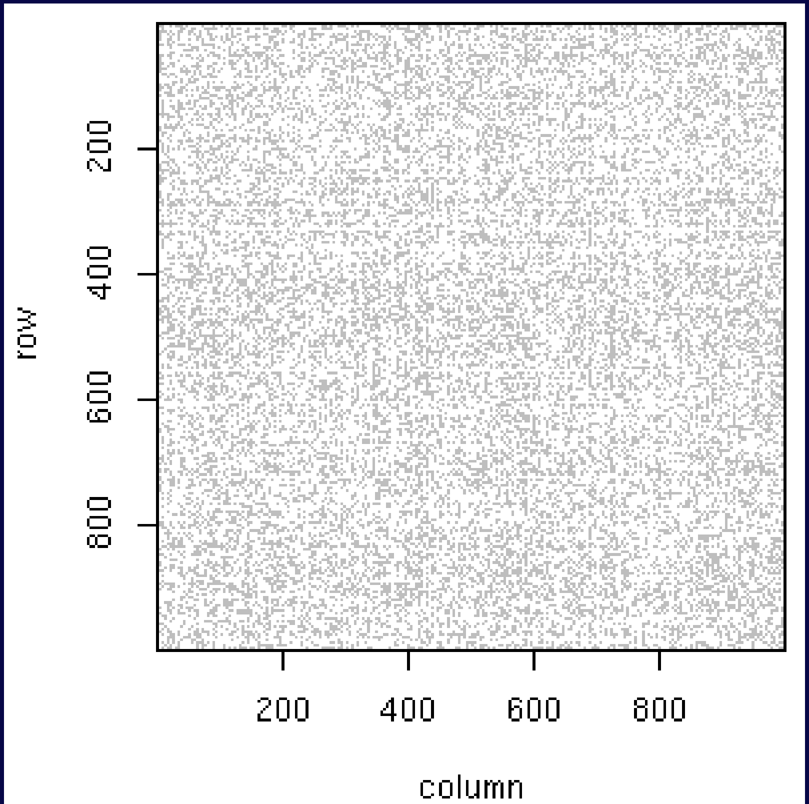


Timing

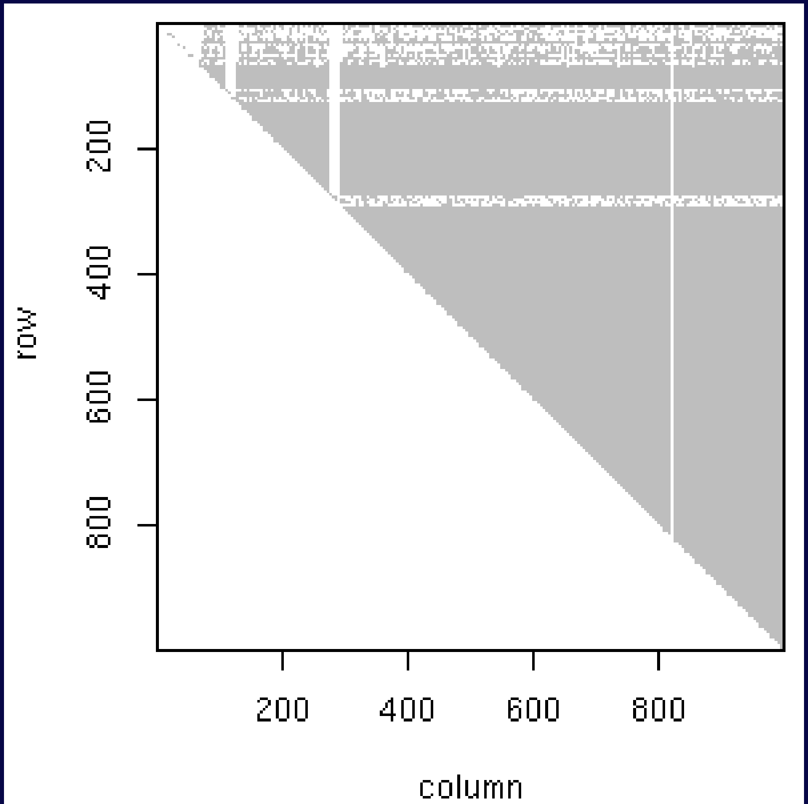
With cutoff 0.99:

```
R> timing(AtA <- t(A) %*% A)
[1] 0.106
R> timing(StS <- t(S) %*% S)
[1] 0.089

R> timing(chol(AtA))
[1] 0.494
R> timing(chol(StS))
[1] 0.551
```

Timing



row
200
400
600
800
200
400
600
800
column



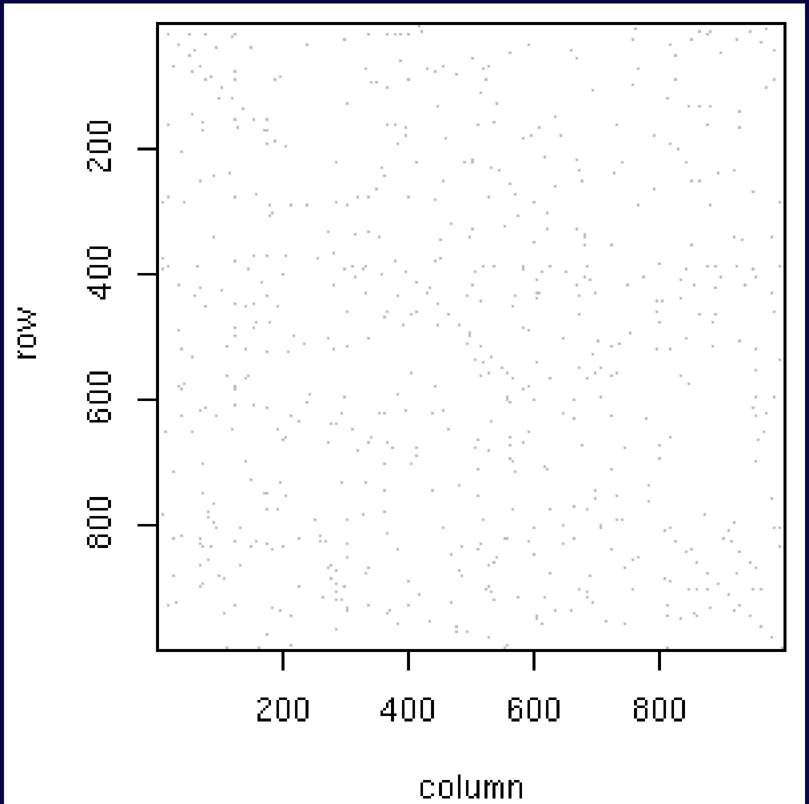
row
200
400
600
800
200
400
600
800
column

Timing

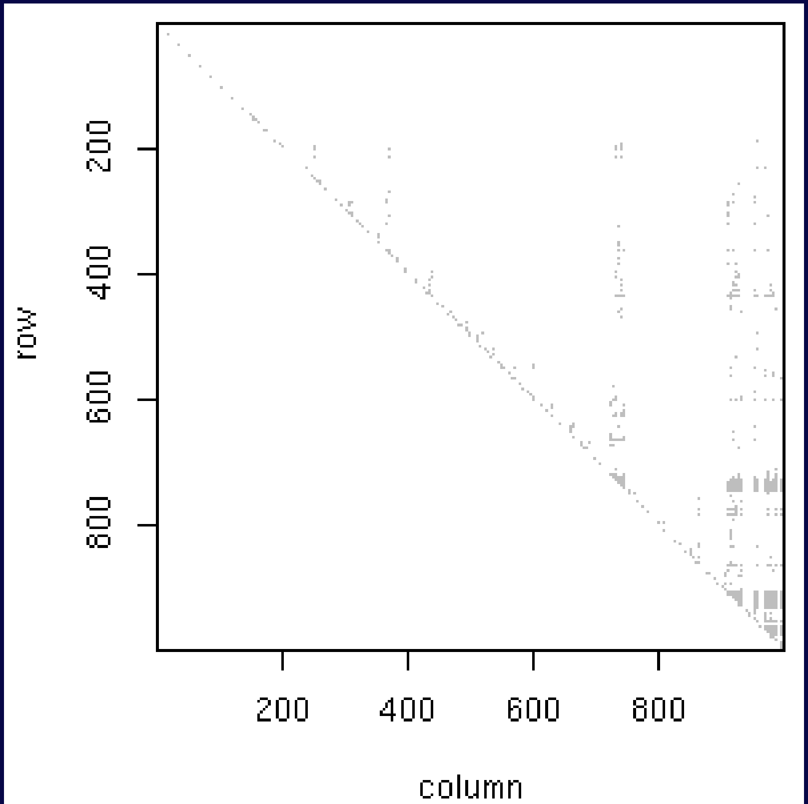
With cutoff 0.99:

```
R> timing(AtA <- t(A) %*% A)
[1] 0.059
R> timing(StS <- t(S) %*% S)
[1] 0.002

R> timing(chol(AtA))
[1] 0.466
R> timing(chol(StS))
[1] 0.007
```

Timing



row
200
400
600
800
200
400
600
800
column

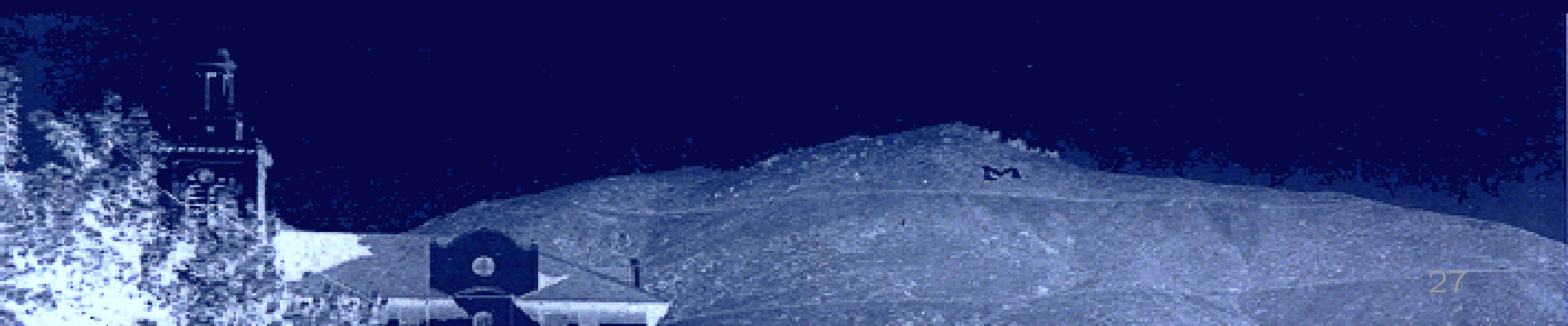


row
200
400
600
800
200
400
600
800
column

Implications

With a new storage format new “algorithms” are required . . .

Is it worthwhile??? Yes!



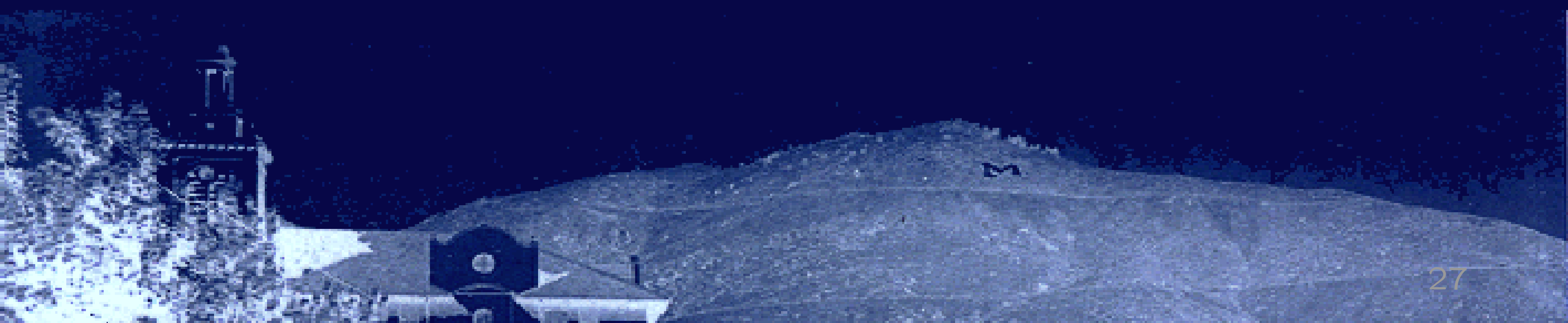
Implications

With a new storage format new “algorithms” are required . . .

Is it worthwhile??? Yes!

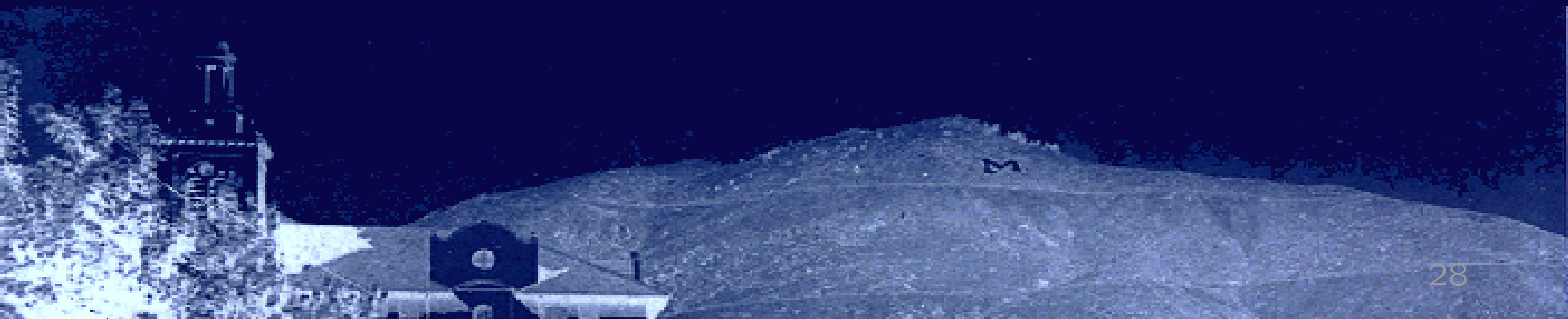
Especially since

`spam`: R package for sparse matrix algebra.



Some slides about `spam`?

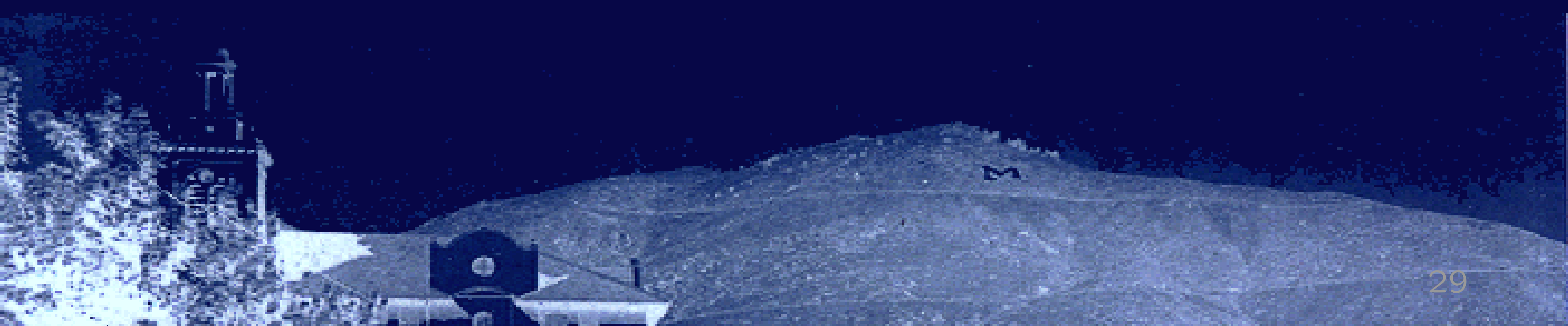
. . . see inlet one.



Sparse Matrices in Statistics

Where do large matrices occur?

- Location matrices
- Design matrices



Sparse Matrices in Statistics

Where do large matrices occur?

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

Sparse Matrices in Statistics

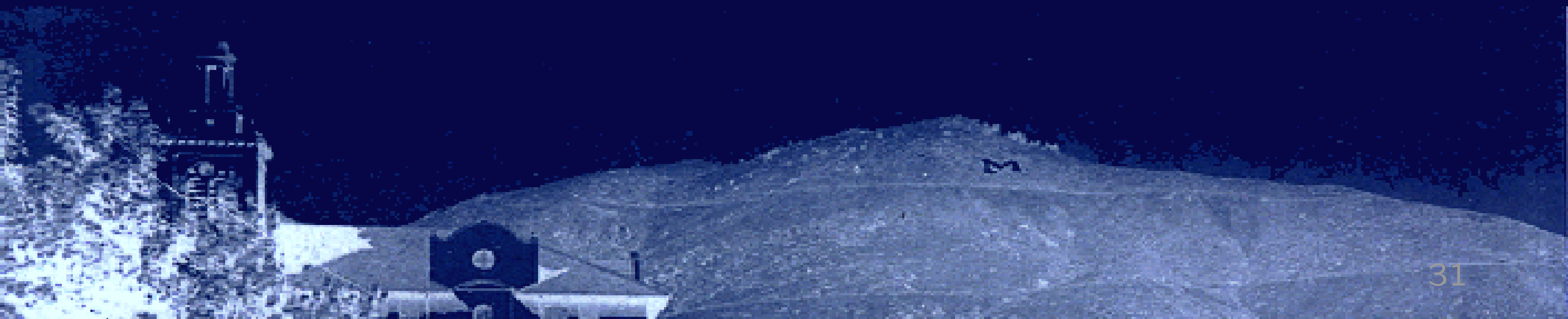
- Covariance matrices:
  Compactly supported covariance functions
  Tapering

- Precision matrices:
  (Gaussian) Markov random fields
  (Tapering???)

We have symmetric positive definite (spd) matrices.

Some slides about tapering?

. . . see inlet two.



Positive Definite Matrices

A (large dimensional) 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 ...

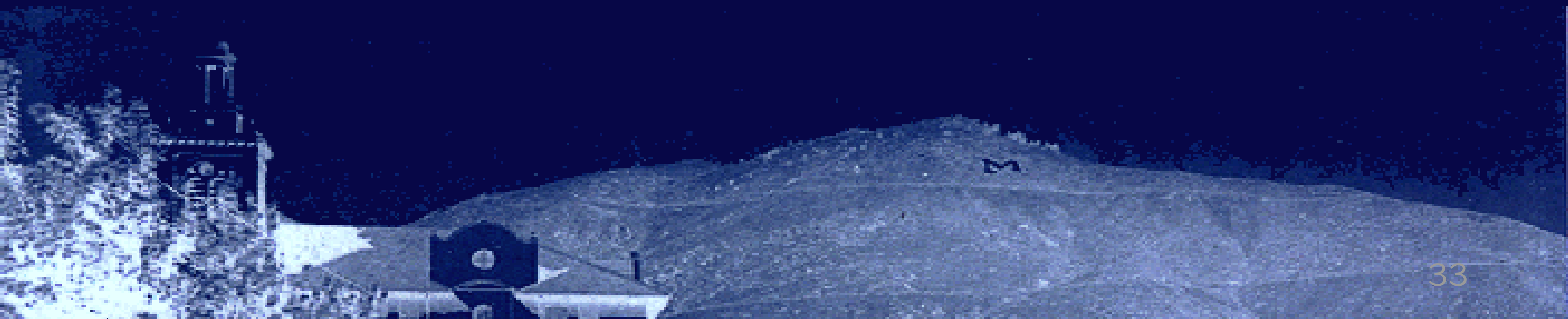
Sparse PCA is sparse in a different sense ...

Solving Linear Systems

To solve the system $\mathbf{Ax} = \mathbf{b}$, we

- perform a Cholesky factorisation $\mathbf{A} = \mathbf{U}^\mathsf{T}\mathbf{U}$
- solve two triangular systems $\mathbf{U}^\mathsf{T}\mathbf{z} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{z}$

But we need to “ensure” that $\mathbf{U}$ is as sparse as possible!



Solving Linear Systems

To solve the system $\mathbf{Ax} = \mathbf{b}$, we

- perform a Cholesky factorisation $\mathbf{A} = \mathbf{U}^T\mathbf{U}$
- solve two triangular systems $\mathbf{U}^T\mathbf{z} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{z}$

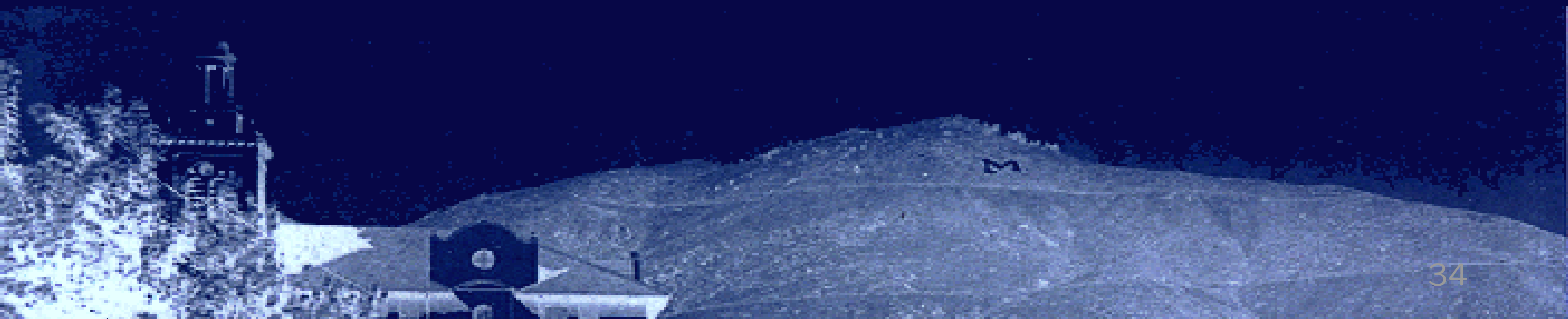
But we need to "ensure" that $\mathbf{U}$ is as sparse as possible!

Permute the rows and columns of $\mathbf{A}$: $\mathbf{P}^T\mathbf{AP} = \mathbf{U}^T\mathbf{U}$.

`spam` performs Cholesky factorization very efficiently!

Determinant

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



Sparse Matrices and `fields`

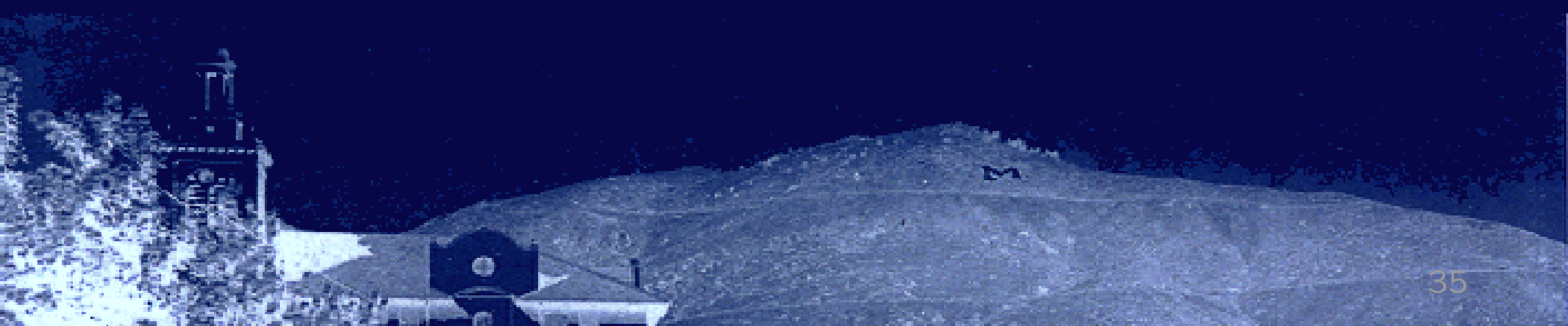
- `fields` is not bound to a specific sparse matrix format
- All heavy lifting is done in `mKrig` or `Krig.engine.fixed`
- For a specific sparse format, requires the methods:

  `chol`, `backsolve`, `forwardsolve` and `diag`

  as well as elementary matrix operations

  need to exist
- If available uses operators to handle diagonal matrices quickly

$\rightsquigarrow$ The covariance matrix has to stem from particular class.



Sparse Matrices and `fields`

- `fields` is not bound to a specific sparse matrix format
- All heavy lifting is done in `mKrig` or `Krig.engine.fixed`
- For a specific sparse format, requires the methods:

  `chol`, `backsolve`, `forwardsolve` and `diag`

  as well as elementary matrix operations

  need to exist
- If available uses operators to handle diagonal matrices quickly

$\rightsquigarrow$ The covariance matrix has to stem from particular class.

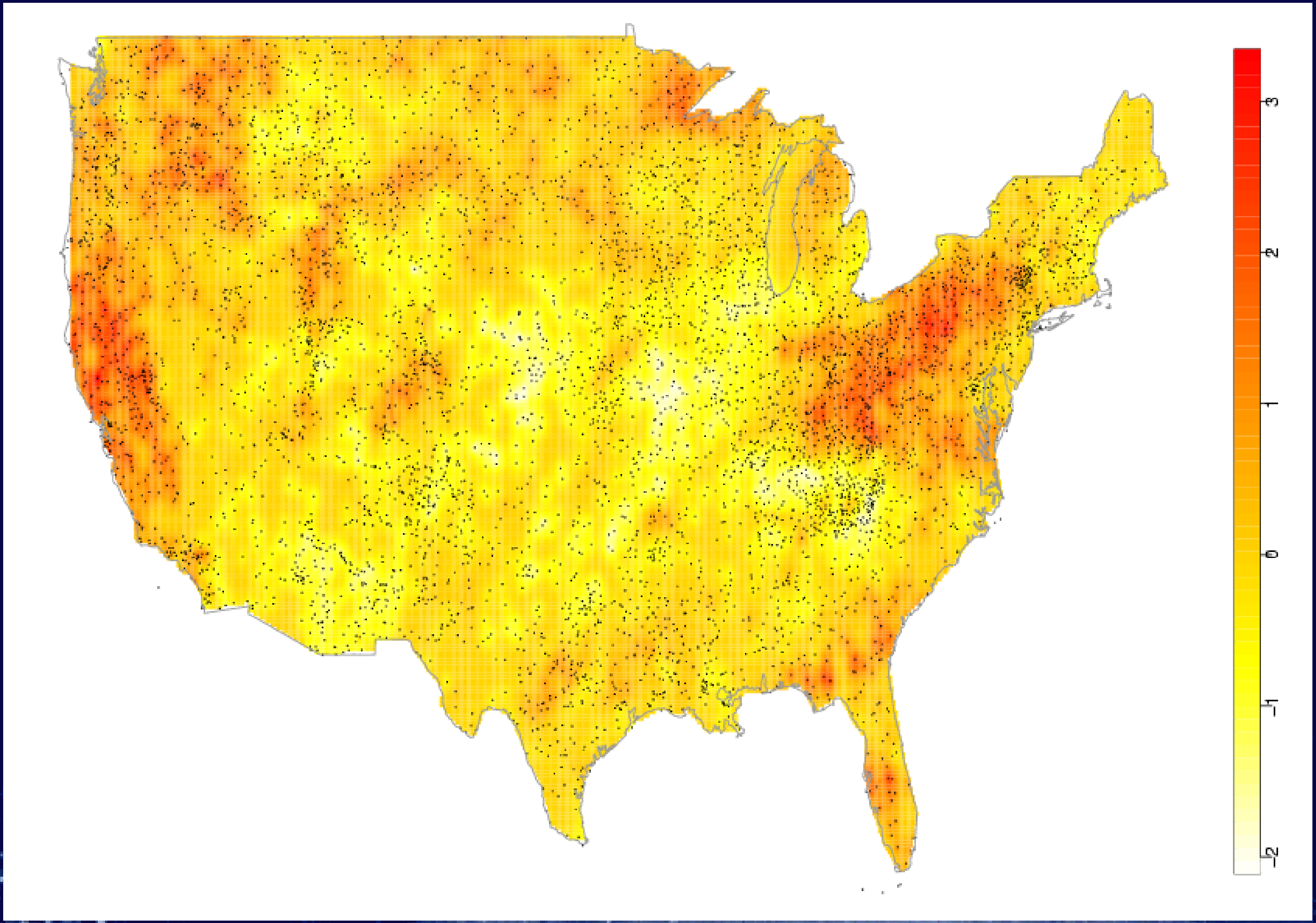
`fields` uses `spam` as default package!

Example

With appropriate covariance function:

```
R> x <- USprecip[ precipsubset,1:2] # locations
R> Y <- USprecip[ precipsubset,4]   # anomaly

R> lon <- seq(-125, to=-68, by=blat)
R> lat <- seq( 24.5, to=49, by=blat)

R> pred.x <- expand.grid(lon,lat)

R> out <- mKrig(x,Y, m=1, cov.function="wendland.cov")
R> pred.out <- predict(out, xnew=pred.x)
```

Example



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

And Beyond?

Parallelization:

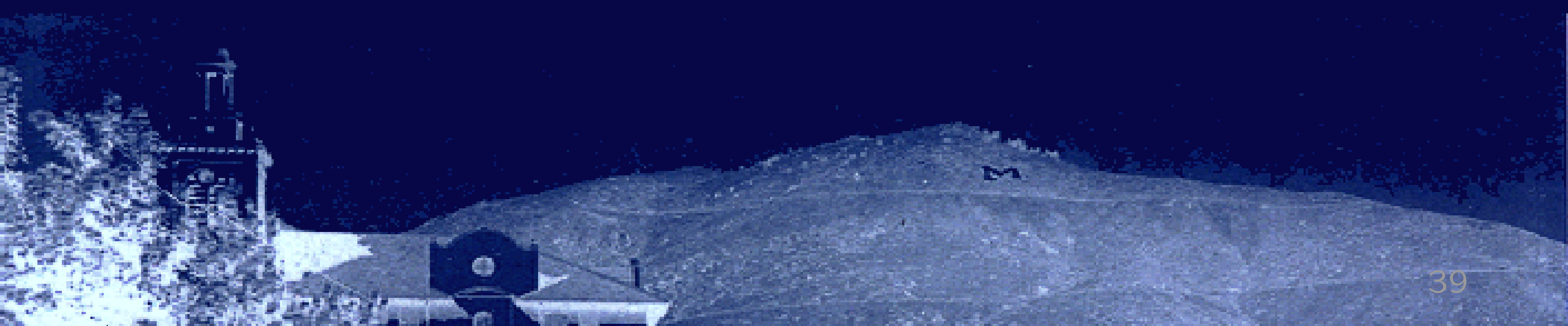
`nws`, `snow`, `Rmpi`, . . .

Memory "Outsourcing":

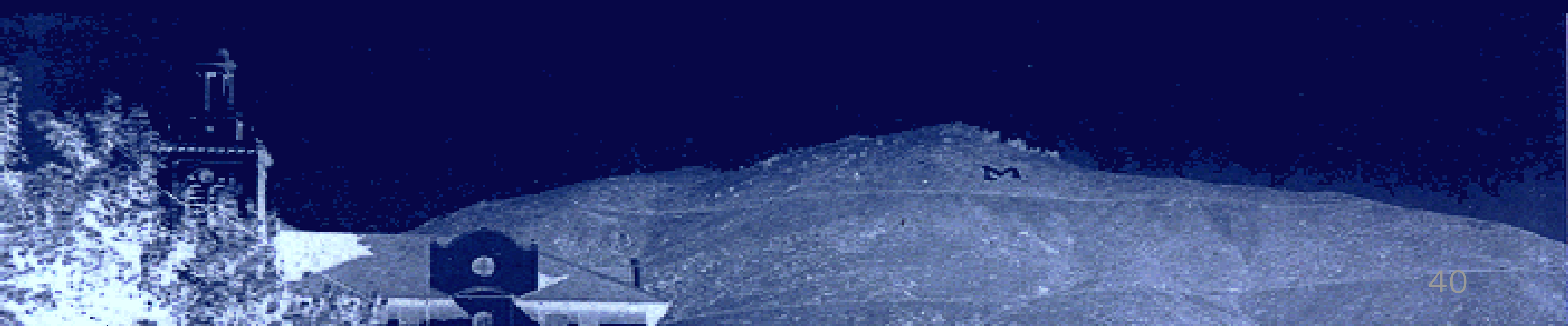
Matrices are not (entirely) kept in memory:

`ff`, `filehash`, `biglm`, . . .

(S+ has the library `BufferedMatrix`)



And Now?



And Now?

Mixer!!