Statistics 1 Unit 1: Numerical Linear Algebra

Kurt Hornik

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

Outline

- Matrix basics
- Matrix decompositions and linear systems

Outline

- Matrix basics
  - Matrix basics
  - Subscripting
  - Matrix operations
  - Tasks
- Matrix decompositions and linear systems

Matrix basics

Matrices and arrays are represented as “structures”: vectors (can therefore also be character or list) with a `dim` and optionally a `dimnames` attribute.

Matrix basics

Matrices and arrays are represented as “structures”: vectors (can therefore also be character or list) with a `dim` and optionally a `dimnames` attribute.

Creation via `matrix()`, `rbind()` and `cbind()`; `diag()` for creating diagonal matrices.

```
R> m <- matrix(1 : 6, 2, 3)
R> m

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrix basics

Matrices and arrays are represented as “structures”: vectors (can therefore also be character or list) with a `dim` and optionally a `dimnames` attribute.

Creation via `matrix()`, `rbind()` and `cbind()`; `diag()` for creating diagonal matrices.

```
R> m <- matrix(1 : 6, 2, 3)
R> m

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Note that elements are filled by *columns* by default (“column major ordering”): one can fill by rows using `byrow = TRUE`.

Matrix basics

Can get the dimensions via `dim()`:

```
R> dim(m)
```

```
[1] 2 3
```

Matrix basics

Can get the dimensions via `dim()`:

```
R> dim(m)
[1] 2 3
```

Can get the elements via `c()`:

```
R> c(m)
[1] 1 2 3 4 5 6
```

Matrix basics

Can also manipulation dimensions via `dim()` (connaisseurs: dim getter and dim setter):

```
R> dim(m) <- c(3, 2)
R> m
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Matrix basics

Can also manipulation dimensions via `dim()` (connaisseurs: dim getter and dim setter):

```
R> dim(m) <- c(3, 2)
R> m
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Or even: “matrix, go away”:

```
R> dim(m) <- NULL
R> m
```

```
[1] 1 2 3 4 5 6
```

Matrix basics

rbind() combines its arguments by *rows*:

```
R> ## Turn a sequence into a "row vector":
R> rbind(c(1, 3, 5))

     [,1] [,2] [,3]
[1,]    1    3    5

R> ## Create a matrix from its rows:
R> rbind(c(1, 3, 5), c(2, 4, 6))

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrix basics

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

cbind() combines its arguments by *c*olumns:

```
R> ## Turn a sequence into a "column vector":
R> cbind(c(1, 2))

     [,1]
[1,]    1
[2,]    2

R> ## Create a matrix from its columns:
R> cbind(c(1, 2), c(3, 4), c(5, 6))

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrix basics

`diag()` creates diagonal matrices (or extracts diagonals):

```
R> diag(1 : 3)

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3

R> ## Unit matrix:
R> diag(1, nrow = 3)

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

(Or use `diag(rep(1, 3))`.)

Matrix basics

Basic matrix functions:

- `c()` extracts the elements
- `dim()` getter/setter for the `dim` attribute
- `nrow()` and `ncol()` for getting the number of rows or columns
- `dimnames()` getter/setter for the `dimnames` attribute
- `rownames()` and `colnames()` getters and setters for the row and column names

Matrix basics

```
R> m <- matrix(1 : 6, 2, 3)
R> dimnames(m) <- list(c("R1", "R2"), c("C1", "C2", "C3"))
R> m

   C1 C2 C3
R1  1  3  5
R2  2  4  6
```

Can also give the dimnames in the `dimnames` argument to `matrix()`.

```
R> dimnames(m)

[[1]]
[1] "R1" "R2"

[[2]]
[1] "C1" "C2" "C3"
```

Matrix basics

```
R> rownames(m) <- letters[1 : 2]
R> colnames(m) <- NULL
R> m
```

```
  [,1] [,2] [,3]
a    1    3    5
b    2    4    6
```

Note:

```
R> dimnames(m)
```

```
[[1]]
[1] "a" "b"

[[2]]
NULL
```

Outline

- Matrix basics
  - Matrix basics
  - Subscripting
  - Matrix operations
  - Tasks
- Matrix decompositions and linear systems

Subscripting

- Extract sub-matrices by subscripting rows and columns using vectors of integers or logicals or characters (if the matrix has the appropriate dimnames) (“2-argument subscripting”).
  Note that by default this drops dimensions if possible.

Subscripting

- Extract sub-matrices by subscripting rows and columns using vectors of integers or logicals or characters (if the matrix has the appropriate dimnames) (“2-argument subscripting”).
  Note that by default this drops dimensions if possible.
- Extract elements by subscripting with a single vector of integers or logicals, or a 2-column index matrix.

2-argument subscripting

```
R> (m <- matrix(1 : 6, 2, 3))

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

R> m[1, 2 : 3]

[1] 3 5

R> m[-1, 2 : 3, drop = FALSE]

     [,1] [,2]
[1,]    4    6

R> m[2, 2]

[1] 4
```

1-argument subscripting

```
R> (m <- matrix(1 : 4, 2, 2))

     [,1] [,2]
[1,]    1    3
[2,]    2    4

R> m[c(1, 4)]

[1] 1 4

R> m[-3]

[1] 1 2 4
```

1-argument subscripting

```
R> ## Extract even elements, variant 1:
R> i <- ((m %% 2) == 0)
R> m[i]

[1] 2 4

R> ## Alternatively, use an index matrix:
R> i <- which((m %% 2) == 0, arr.ind = TRUE)
R> i

     row col
[1,]   2   1
[2,]   2   2

R> m[i]

[1] 2 4
```

Subscripting

`diag()` can also be used for extracting the diagonal of a matrix.

`lower.tri()` and `upper.tri()` can be employed for extracting the lower and upper triangular parts of a matrix:

```
R> m <- matrix(1 : 9, 3, 3)
R> m

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

R> ## Extract diagonal elements.
R> diag(m)

[1] 1 5 9
```

Subscripting

```
R> ## Extract elements below the main diagonal.
R> m[lower.tri(m)]

[1] 2 3 6

R> ## Extract elements not above the main diagonal.
R> m[lower.tri(m, diag = TRUE)]

[1] 1 2 3 5 6 9

R> ## Extract elements above the main diagonal.
R> m[upper.tri(m)]

[1] 4 7 8
```

Subscripting

How does this work?

```
R> lower.tri(m)

      [,1]  [,2]  [,3]
[1,] FALSE FALSE FALSE
[2,]  TRUE FALSE FALSE
[3,]  TRUE  TRUE FALSE

R> lower.tri(m, diag = TRUE)

     [,1]  [,2]  [,3]
[1,] TRUE FALSE FALSE
[2,] TRUE  TRUE FALSE
[3,] TRUE  TRUE  TRUE
```

Simply uses 1-argument subscripting.

Subscripting

In fact, one can “do it yourself” using `row()` and `col()`:

```
R> row(m)

     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3

R> col(m)

     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
```

Subscripting

```
R> ## Elements below the main diagonal:
R> row(m) > col(m)

      [,1]  [,2]  [,3]
[1,] FALSE FALSE FALSE
[2,]  TRUE FALSE FALSE
[3,]  TRUE  TRUE FALSE

R> ## elements not above the main diagonal:
R> row(m) >= col(m)

     [,1]  [,2]  [,3]
[1,] TRUE FALSE FALSE
[2,] TRUE  TRUE FALSE
[3,] TRUE  TRUE  TRUE
```

Subscripting

Using `row()` and `col()`, we can also split a matrix into its rows or columns:

```
R> m <- matrix(1 : 6, 2, 3)
R> split(m, row(m))

$`1`
[1] 1 3 5

$`2`
[1] 2 4 6
```

Subscripting

How can we get the matrix back from the list of its row vectors?

Formally: suppose we have an $m \times n$ matrix m with row vectors $r_1, \dots, r_m$. We know that

$$m = \text{rbind}(r_1, \dots, r_m)$$

but what if we have the row vectors in a *list*?

Want “call rbind with the list (of row vectors) as its arguments”.

Subscripting

How can we get the matrix back from the list of its row vectors?

Formally: suppose we have an $m \times n$ matrix m with row vectors $r_1, \ldots, r_m$. We know that

$$m = \mathrm{rbind}(r_1, \ldots, r_m)$$

but what if we have the row vectors in a *list*?

Want “call rbind with the list (of row vectors) as its arguments”.

Have `do.call()` for this.

Subscripting

```
R> m <- matrix(1 : 6, 2, 3)
R> (r <- split(m, row(m)))

$`1`
[1] 1 3 5

$`2`
[1] 2 4 6

R> do.call(rbind, r)

  [,1] [,2] [,3]
1    1    3    5
2    2    4    6
```

Outline

- Matrix basics
  - Matrix basics
  - Subscripting
  - Matrix operations
  - Tasks
- Matrix decompositions and linear systems

Basics

`t()` does transposition:

```
R> m <- matrix(1 : 6, 2, 3)
R> m

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

R> t(m)

     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Basics

The basic arithmetic and logical operations on matrices work *element-wise*, preserving dimensions where possible.

Basics

The basic arithmetic and logical operations on matrices work *element-wise*, preserving dimensions where possible.

I.e., operate on the underlying sequences of values, and hence recycle “as necessary” (as discussed).

Basics

The basic arithmetic and logical operations on matrices work *element-wise*, preserving dimensions where possible.

I.e., operate on the underlying sequences of values, and hence recycle "as necessary" (as discussed).

In particular, `A * B` is the element-wise product of A and B ("Hadamard product")!

Basics

```
R> (A <- matrix(1 : 4, 2, 2))

     [,1] [,2]
[1,]    1    3
[2,]    2    4

R> (B <- matrix(5 : 8, 2, 2))

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

Basics

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

These are “as expected”:

```
R> ## Multiplication by a scalar:
R> 2 * A

     [,1] [,2]
[1,]    2    6
[2,]    4    8

R> ## Element-wise subtraction:
R> A - B

     [,1] [,2]
[1,]   -4   -4
[2,]   -4   -4
```


EQUIS ACCREDITED AACSB ACCREDITED ASSOCIATION AMBA ACCREDITED

Basics

These are surprising when first encountered:

```
R> A - 2

     [,1] [,2]
[1,]   -1    1
[2,]    0    2

R> A / B

          [,1]      [,2]
[1,] 0.2000000 0.4285714
[2,] 0.3333333 0.5000000
```

Basics

And also matrix/vector operations do not work as expected:

```
R> x <- c(2, 3)
R> B * x

     [,1] [,2]
[1,]   10   14
[2,]   18   24

R> ## Compare to:
R> c(B) * x

[1] 10 18 14 24
```

Basics

To get the usual matrix product, use %*%.

```
R> A %*% B

     [,1] [,2]
[1,]   23   31
[2,]   34   46

R> B %*% x

     [,1]
[1,]   31
[2,]   36
```

Note that the latter nicely turns x into a column vector.

Matrix products

We have already seen that in addition to the usual matrix product, there is the element-wise Hadamard product $A \odot B$:

If $A = [\alpha_{ij}]$ and $B = [\beta_{ij}]$ have the same dimensions,

$$[A \odot B]_{ij} = \alpha_{ij}\beta_{ij}.$$

Matrix products

We have already seen that in addition to the usual matrix product, there is the element-wise Hadamard product $A \odot B$:

If $A = [\alpha_{ij}]$ and $B = [\beta_{ij}]$ have the same dimensions,

$$[A \odot B]_{ij} = \alpha_{ij}\beta_{ij}.$$

There is also the *Kronecker* product $A \otimes B$ which takes the products of all pairs of elements of A and B, arranged suitably.

This works for matrices of arbitrary sizes.

Matrix products

If $A = [\alpha_{ij}]$, the Kronecker product of A and B is defined as

$$A \otimes B = \begin{bmatrix} \alpha_{11}B & \cdots & \alpha_{1n}B \\ \vdots & \ddots & \vdots \\ \alpha_{m1}B & \cdots & \alpha_{mn}B \end{bmatrix}$$

For example:

```
R> kronecker(A, B)

     [,1] [,2] [,3] [,4]
[1,]    5    7   15   21
[2,]    6    8   18   24
[3,]   10   14   20   28
[4,]   12   16   24   32
```

Matrix products

These Kronecker products are very useful for multivariate analysis.

They have the following fundamental properties:

$$(A \otimes B)' = A' \otimes B', \quad (A \otimes B)(C \otimes D) = AC \otimes BD, \quad (A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$$

If we write vec(A) for the (column) vector obtained by stacking the columns of the matrix A one underneath the other:

$$\text{vec}(A) = [a_1', \ldots, a_n']', \qquad A = [a_1, \ldots, a_n]$$

(remember that $'$ denotes transpose), then

$$\text{vec}(ABC) = (C' \otimes A)\text{vec}(B).$$

Cross products

Let $A = [a_1, \dots, a_n]$ have columns a_i and $B = [b_1, \dots, b_n]$ have columns b_j.

Then A' has rows $a_1', \dots, a_n'$, and hence the (i, j) element of the matrix product $A'B$ is $a_i' b_j$, the inner product of the i-th column of A and the j-th column of B:

$$[A'B]_{ij} = a_i' b_j.$$

This is called the *cross-product* of A and B. In R, `crossprod()`.

Clearly, `crossprod(A, B)` is the same as `t(A) %*% B`, but computed more efficiently.

There is also `tcrossprod(A, B)` for AB'.

`apply()` and `sweep()`

`apply()` applies functions over array margins: in the simplest case, to the rows or columns of a matrix.

`sweep()` sweeps out array/matrix summaries.

E.g.,

```
R> (A <- matrix(1 : 9, 3, 3))

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

`apply()` and `sweep()`

```
R> ## Row sums:
R> apply(A, 1, sum)
```

```
[1] 12 15 18
```

```
R> ## Col sums:
R> apply(A, 2, sum)
```

```
[1]  6 15 24
```

`apply()` “always” works, but for some cases there are faster variants:

- `rowSums()`/`colSums()` for row and col sums,
- `rowMeans()`/`colMeans()` for row and col means.

apply() and sweep()

Now suppose we want to center the rows of a matrix. We can do

```
R> sweep(A, 1, rowMeans(A))
```

```
     [,1] [,2] [,3]
[1,]   -3    0    3
[2,]   -3    0    3
[3,]   -3    0    3
```

Indeed,

```
R> rowMeans( sweep(A, 1, rowMeans(A)) )
```

```
[1] 0 0 0
```

has centered rows.

`apply()` and `sweep()`

How does this work? Formally, if $A = [\alpha_{ij}]$ and $x = [\xi_i]$, we want to compute the matrix with entries

$$\alpha_{ij} - \xi_i.$$

There is nothing special about differences (it is used by `sweep()` by default). In general, sweeping out row summaries x computes the matrix with entries

$$f(\alpha_{ij}, \xi_i).$$

`apply()` and `sweep()`

How does this work? Formally, if $A = [\alpha_{ij}]$ and $x = [\xi_i]$, we want to compute the matrix with entries

$$\alpha_{ij} - \xi_i.$$

There is nothing special about differences (it is used by `sweep()` by default). In general, sweeping out row summaries x computes the matrix with entries

$$f(\alpha_{ij}, \xi_i).$$

Similarly, if $y = [\eta_j]$, sweeping out col summaries y computes the matrix with entries

$$f(\alpha_{ij}, \eta_j).$$

Outline

- Matrix basics
  - Matrix basics
  - Subscripting
  - Matrix operations
  - Tasks
- Matrix decompositions and linear systems

Task 1: Multiply the rows of a matrix by a vector

If $A = [\alpha_{ij}]$ is $m \times n$ and $v = [v_i]$ is $m \times 1$ (or simply a sequence of length m), we want to compute the $m \times n$ matrix with entries

$$\alpha_{ij} v_i.$$

Mathematically, we can do

$$\mathrm{rmult}(A, v) = \mathrm{diag}(v) A.$$

Task 1: Multiply the rows of a matrix by a vector

If $A = [\alpha_{ij}]$ is $m \times n$ and $v = [v_i]$ is $m \times 1$ (or simply a sequence of length m), we want to compute the $m \times n$ matrix with entries

$$\alpha_{ij} v_i.$$

Mathematically, we can do

$$\mathrm{rmult}(A, v) = \mathrm{diag}(v) A.$$

Check: write δ_{ij} for the Kronecker δ:

$$\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j. \end{cases}$$

Task 1: Multiply the rows of a matrix by a vector

Then $\mathrm{diag}(v) = [v_i \delta_{ij}]$ and hence

$$[\mathrm{diag}(v)A]_{ij} = \sum_k [\mathrm{diag}(v)]_{ik} \alpha_{kj} = \sum_k v_i \delta_{ik} \alpha_{kj} = v_i \alpha_{ij}.$$

So we could compute as `diag(v) %*% A`, but is this smart?

Task 1: Multiply the rows of a matrix by a vector

Then $\mathrm{diag}(v) = [v_i \delta_{ij}]$ and hence

$$[\mathrm{diag}(v) A]_{ij} = \sum_k [\mathrm{diag}(v)]_{ik} a_{kj} = \sum_k v_i \delta_{ik} a_{kj} = v_i a_{ij}.$$

So we could compute as `diag(v) %*% A`, but is this smart?

No! If A is $m \times n$, needs m^2 extra storage for $\mathrm{diag}(v)$ and (basic counting) mn times m multiplications and $m - 1$ additions (most of these no-ops).

But the task clearly only needs mn multiplications!

Task 1: Multiply the rows of a matrix by a vector

Then $\mathrm{diag}(v) = [v_i \delta_{ij}]$ and hence

$$[\mathrm{diag}(v)A]_{ij} = \sum_k [\mathrm{diag}(v)]_{ik} a_{kj} = \sum_k v_i \delta_{ik} a_{kj} = v_i a_{ij}.$$

So we could compute as `diag(v) %*% A`, but is this smart?

No! If A is $m \times n$, needs m^2 extra storage for $\mathrm{diag}(v)$ and (basic counting) mn times m multiplications and $m - 1$ additions (most of these no-ops).

But the task clearly only needs mn multiplications!

How can we do better?

Task 1: Multiply the rows of a matrix by a vector

We know we need to compute the matrix with entries

$$\alpha_{ij} v_i$$

so that's a row sweep with the multiplication function:

```
R> rmult <- function(A, v) sweep(A, 1, v, `*`)
```

E.g.,

```
R> A <- matrix(1 : 4, 2, 2)
R> v <- c(2, 3)
R> rmult(A, v)
```

```
     [,1] [,2]
[1,]    2    6
[2,]    6   12
```

Task 1: Multiply the rows of a matrix by a vector

For connaisseurs: we can also simply do

```
R> A * v
     [,1] [,2]
[1,]    2    6
[2,]    6   12
```

Why? A is stored in column major order:

$$\alpha_{11}, \alpha_{21}, \ldots, \alpha_{m1}, \ldots, \alpha_{1n}, \alpha_{2n}, \ldots, \alpha_{mn},$$

recycling v gives

$$v_1, v_2, \ldots, v_m, \ldots, v_1, v_2, \ldots, v_m$$

so element-wise multiplication works "as desired".

Task 2: Multiply the cols of a matrix by a vector

If $A = [\alpha_{ij}]$ is $m \times n$ and $v = [v_j]$ is $n \times 1$ (or simply a sequence of length n), we want to compute the $m \times n$ matrix with entries

$$\alpha_{ij} v_j.$$

Mathematically, we can do

$$\mathrm{cmult}(A, v) = A \,\mathrm{diag}\,(v).$$

Check:

$$[A\,\mathrm{diag}(v)]_{ij} = \sum_k \alpha_{ik} [\mathrm{diag}(v)]_{kj} = \sum_k \alpha_{ik} v_k \delta_{kj} = \alpha_{ij} v_j.$$

Task 2: Multiply the cols of a matrix by a vector

Now everyone can venture: we could compute as `A %*% diag(v)`, but this is a bad idea. Instead, we should do a col sweep with the multiplication function:

```
R> cmult <- function(A, v) sweep(A, 2, v, `*`)
```

Task 2: Multiply the cols of a matrix by a vector

Now everyone can venture: we could compute as A `%*% diag(v)`, but this is a bad idea. Instead, we should do a col sweep with the multiplication function:

```
R> cmult <- function(A, v) sweep(A, 2, v, `*`)
```

E.g.,

```
R> A <- matrix(1 : 4, 2, 2)
R> v <- c(2, 3)
R> cmult(A, v)
```

```
     [,1] [,2]
[1,]    2    9
[2,]    4   12
```

Task 2: Multiply the cols of a matrix by a vector

Connaisseurs will now wonder: is there a more direct way without sweeping?

Well, A is stored as

$$\alpha_{11}, \alpha_{21}, \ldots, \alpha_{m1}, \ldots, \alpha_{1n}, \alpha_{2n}, \ldots, \alpha_{mn},$$

but now we need

$$v_1, v_1, \ldots, v_1, \ldots, v_n, v_n, \ldots, v_n$$

with each v_j repeated m times.

So we could do `A * rep(v, each = nrow(A))`!

Task 3: Trace of the crossprod

The *trace* of a square matrix $A = [\alpha_{ij}]$ is the sum of its diagonal elements:

$$\mathrm{trace}(A) = \sum_i \alpha_{ii}.$$

We could implement the trace of the crossprod as `sum(diag(crossprod(A)))`, but can we do better?

Task 3: Trace of the crossprod

The *trace* of a square matrix $A = [\alpha_{ij}]$ is the sum of its diagonal elements:

$$\operatorname{trace}(A) = \sum_i \alpha_{ii}.$$

We could implement the trace of the crossprod as `sum(diag(crossprod(A)))`, but can we do better? Well, we have:

$$\operatorname{trace}(A'A) = \sum_i [A'A]_{ii} = \sum_i \sum_k [A']_{ik}[A]_{ki} = \sum_i \sum_k \alpha_{ki}^2,$$

hence we can do:

```
R> trace_of_crossprod <- function(A) sum(A ^ 2)
```

Task 4: Vandermonde matrix and determinant

The Vandermonde matrix of a sequence $\xi_1, \dots, \xi_n$ is

$$V(\xi_1, \dots, \xi_n) = \begin{bmatrix} 1 & \xi_1 & \xi_1^2 & \cdots & \xi_1^{n-1} \\ 1 & \xi_2 & \xi_2^2 & \cdots & \xi_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \xi_n & \xi_n^2 & \cdots & \xi_n^{n-1} \end{bmatrix}$$

I.e.,

$$[V(\xi_1, \dots, \xi_n)]_{ij} = \xi_i^{j-1}.$$

Write functions to compute the Vandermonde matrix and its determinant.

Task 4: Vandermonde matrix and determinant

How can we compute the matrix with entries ξ_i^{j-1}? Write

$$\xi_i^{j-1} = \mathrm{pow}(\xi_i, j-1)$$

(of course, in R pow is written as '^').

Remember our good old friend `outer()`: for $x = [\xi_i]$ and $y = [\eta_j]$,

$$[\mathrm{outer}(x, y, f)]_{ij} = f(\xi_i, \eta_j).$$

Task 4: Vandermonde matrix and determinant

How can we compute the matrix with entries ξ_i^{j-1}? Write

$$\xi_i^{j-1} = \mathrm{pow}(\xi_i, j-1)$$

(of course, in R pow is written as '^').

Remember our good old friend `outer()`: for $x = [\xi_i]$ and $y = [\eta_j]$,

$$[\mathrm{outer}(x, y, f)]_{ij} = f(\xi_i, \eta_j).$$

So easily,

```
R> Vandermonde <- function(x) outer(x, seq_along(x) - 1, `^`)
```

Task 4: Vandermonde matrix and determinant

```
R> Vandermonde(1 : 5)

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    1    2    4    8   16
[3,]    1    3    9   27   81
[4,]    1    4   16   64  256
[5,]    1    5   25  125  625
```

Task 4: Vandermonde matrix and determinant

```
R> Vandermonde(1 : 5)

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    1    2    4    8   16
[3,]    1    3    9   27   81
[4,]    1    4   16   64  256
[5,]    1    5   25  125  625
```

How can we compute the determinant? Simple way:

```
R> det(Vandermonde(1 : 5))

[1] 288
```

Task 4: Vandermonde matrix and determinant

For connaisseurs: verify first that

$$\det(V(\xi_1, \dots, \xi_n)) = \prod_{1 \le i < j \le n} (\xi_i - \xi_j).$$

So we can do

```
R> Vandermonde_det <- function(x) {
+     diffs <- outer(x, x, `-`)
+     prod(diffs[upper.tri(diffs)])
+ }
```

Check:

```
R> Vandermonde_det(1 : 5)
```

```
[1] 288
```

Outline

- Matrix basics
- Matrix decompositions and linear systems

Outline

- Matrix basics
- Matrix decompositions and linear systems
  - Introduction
  - LU decomposition
  - QR decomposition
  - Singular value decomposition (SVD)
  - Eigendecomposition
  - Choleski decomposition
  - Summary

Introduction

As everyone knows from kindergarden: the $n \times n$ linear system $Ax = b$ has a unique solution iff A is invertible, in which case the unique solution is given by $x = A^{-1}b$.

In R, we can get the inverse using `solve()`.

(Strange, not `inv()`? There must be a reason ...).

Introduction

E.g.,

```
R> A <- matrix(1 : 4, 2, 2)
R> (A_inv <- solve(A))

     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5

R> A %*% A_inv

     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Introduction

So formally, we could solve the linear system $Ax = b$ via literally translating $x = A^{-1}b$ as

```
solve(A) %*% b
```

but *do not do this*!

Instead, one should use one of

```
solve(A, b)
qr.solve(A, b)
```

In the following, we illustrate why. More precisely, we review the basic matrix decompositions and how to use these for solving linear systems.

Introduction

To illustrate matters, we use the linear system

$$H_6 x = b$$

where

$$b = [1, 2, 3, 4, 5, 6]'$$

and H_6 is the 6×6 Hilbert matrix

$$H_6 = [1/(i + j - 1)]_{1 \le i,j \le 6}.$$

Introduction

```
R> b <- 1 : 6
R> H <- 1 / (outer(b, b, `+`) - 1)
R> H

          [,1]      [,2]      [,3]      [,4]      [,5]       [,6]
[1,] 1.0000000 0.5000000 0.3333333 0.2500000 0.2000000 0.16666667
[2,] 0.5000000 0.3333333 0.2500000 0.2000000 0.1666667 0.14285714
[3,] 0.3333333 0.2500000 0.2000000 0.1666667 0.1428571 0.12500000
[4,] 0.2500000 0.2000000 0.1666667 0.1428571 0.1250000 0.11111111
[5,] 0.2000000 0.1666667 0.1428571 0.1250000 0.1111111 0.10000000
[6,] 0.1666667 0.1428571 0.1250000 0.1111111 0.1000000 0.09090909
```

Introduction

Compute the inverse:

```
R> H_inv <- solve(H)
R> H_inv
```

```
      [,1]    [,2]     [,3]     [,4]     [,5]     [,6]
[1,]    36    -630     3360    -7560     7560    -2772
[2,]  -630   14700   -88200   211680  -220500    83160
[3,]  3360  -88200   564480 -1411200  1512000  -582120
[4,] -7560  211680 -1411200  3628800 -3969000  1552320
[5,]  7560 -220500  1512000 -3969000  4410000 -1746360
[6,] -2772   83160  -582120  1552320 -1746360   698544
```

Check whether it does a reasonable job:

Introduction

```
R> H_inv %*% H
```

```
                  [,1]          [,2]          [,3]          [,4]
[1,]  1.000000e+00 -1.062403e-10 -9.003998e-11 -7.804601e-11
[2,]  2.764864e-10  1.000000e+00  1.909939e-10  1.655280e-10
[3,] -1.455192e-10 -5.820766e-11  1.000000e+00 -8.003553e-11
[4,]  1.746230e-10  1.164153e-10  5.820766e-11  1.000000e+00
[5,]  2.328306e-10  2.910383e-11  8.731149e-11  8.731149e-11
[6,] -5.820766e-11  0.000000e+00  0.000000e+00 -4.365575e-11
                  [,5]          [,6]
[1,] -6.889422e-11 -6.178880e-11
[2,]  1.418812e-10  1.246008e-10
[3,] -4.365575e-11 -4.365575e-11
[4,]  2.910383e-11  5.820766e-11
[5,]  1.000000e+00  8.731149e-11
[6,] -4.365575e-11  1.000000e+00
```

Introduction

```
R> max(abs((H_inv %*% H) - diag(6)))
[1] 2.764864e-10
```

Hmm. Only up to 10 digits for a 6×6 matrix? This is not really impressive.

Introduction

Now compute "solutions" of $H_6 x = b$ using the 3 indicated methods:

```
R> x1 <- c(H_inv %*% b)
R> ## (Use c() to obtain a dim-less vector.)
R> x2 <- solve(H, b)
R> x3 <- qr.solve(H, b)
```

Introduction

How close are these?

```
R> x1 - x2
```

```
[1] -2.787147e-09  5.567017e-09 -9.094947e-10  1.804437e-09
[5]  4.365575e-10 -2.473826e-10
```

```
R> max(abs(x1 - x2))
```

```
[1] 5.567017e-09
```

and compactly:

```
R> dist(rbind(x1, x2, x3), "maximum")
```

```
             x1           x2
x2 5.567017e-09
x3 1.414525e-05 1.414568e-05
```

Introduction

But how "good" are the solutions?

```
R> b1 <- H %*% x1
R> b2 <- H %*% x2
R> b3 <- H %*% x3
```

Inspect the difference to *b*:

```
R> cbind(b1, b2, b3) - b
```

```
             [,1]         [,2]          [,3]
[1,] 1.909939e-10 0.000000e+00 -3.637979e-12
[2,] 6.311893e-10 3.637979e-12 -5.456968e-12
[3,] 6.111804e-10 1.818989e-12  1.818989e-12
[4,] 5.511538e-10 0.000000e+00 -5.456968e-12
[5,] 4.893081e-10 1.818989e-12 -1.818989e-12
[6,] 4.383764e-10 2.728484e-12 -3.637979e-12
```

Introduction

Inspect the maximal differences:

```
R> apply(abs(cbind(b1, b2, b3) - b), 2, max)
[1] 6.311893e-10 3.637979e-12 5.456968e-12
```

So in some sense, solutions 2 and 3 are “better”, although they are “rather different”. Strange.

Outline

- Matrix basics
- Matrix decompositions and linear systems
  - Introduction
  - LU decomposition
  - QR decomposition
  - Singular value decomposition (SVD)
  - Eigendecomposition
  - Choleski decomposition
  - Summary

LU decomposition

The LU decomposition of a quadratic matrix A is

$$A = LU$$

where L is lower and U is upper triangular.

Not all square matrices have such a decomposition.

LU decomposition

The LU decomposition of a quadratic matrix A is

$$A = LU$$

where L is lower and U is upper triangular.

Not all square matrices have such a decomposition.

Why useful? Consider the linear system

$$Ax = LUx = b.$$

This can be solved as

$$Ly = b, \quad Ux = y.$$

LU decomposition

So

$$x = U^{-1}y = U^{-1}L^{-1}b$$

as of course

$$A^{-1} = (LU)^{-1} = U^{-1}L^{-1}.$$

How can we solve

$$Ly = b, \quad Ux = y?$$

LU decomposition

As L is lower triangular, $Ly = b$ can be written as

$$\begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_n \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix}.$$

Clearly, we can solve this *forward*: obtain η_1 from the first eqn, then η_2 from the second, and so on.

In R, we could do

```
y <- forwardsolve(L, b)
```

LU decomposition

As U is upper triangular, $Ux = y$ can be written as

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \cdots & u_{2n} \\ & & \ddots & \vdots \\ & & & u_{nn} \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{bmatrix} = \begin{bmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_n \end{bmatrix}.$$

Clearly, we can solve this *backward*: obtain ξ_n from the last eqn, then η_{n-1} from the last but one, and so on.

In R, we could do

```
x <- backsolve(U, y)
```

LU decomposition

In fact, the above also shows: if L (R) is a regular lower (upper) triangular matrix, its inverse L^{-1} (R^{-1}) is lower (upper) triangular.

If we do full Gauss elimination:

$$A|I \longrightarrow U|L$$

we compute the LU decomposition.

Interestingly, although we've learned to always do this by hand, one never does this using the computer, as computing the LU decomposition (when it exists) is numerically unstable.

In case R there is no function to compute the LU decomposition.

Outline

- Matrix basics
- Matrix decompositions and linear systems
  - Introduction
  - LU decomposition
  - QR decomposition
  - Singular value decomposition (SVD)
  - Eigendecomposition
  - Choleski decomposition
  - Summary

QR decomposition

The QR decomposition of a quadratic matrix A is

$$A = QR$$

where Q is orthogonal and R is upper triangular.

The inverse of A can be computed as

$$A^{-1} = (QR)^{-1} = R^{-1}Q^{-1} = R^{-1}Q'$$

(remember the inverse of an orthogonal matrix is its transpose!).

QR decomposition

The linear system $Ax = QRx = b$ can be solved via the QR decomposition as

$$Qy = b, \quad Rx = y$$

via

$$y = Q'b, \quad x = \text{backsolve}(R, y).$$

QR decomposition I

In R, we can compute the QR decomposition via `qr()`, which returns something “strange”.

```
R> (H_qr <- qr(H))

$qr
           [,1]       [,2]         [,3]          [,4]          [,5]
[1,] -1.2212243 -0.7018717 -0.504470316 -0.3969691267 -3.284337e-01
[2,]  0.4094252 -0.1384670 -0.151130170 -0.1443643562 -1.340082e-01
[3,]  0.2729501  0.5029231 -0.009561613 -0.0151932381 -1.813029e-02
[4,]  0.2047126  0.4674665  0.419825664  0.0004802815  9.942382e-04
[5,]  0.1637701  0.4221195  0.595589435 -0.3630074314  1.733898e-05
[6,]  0.1364751  0.3804247  0.680569302 -0.8985477890  3.663121e-01
              [,6]
[1,] -2.806128e-01
[2,] -1.236690e-01
[3,] -1.953170e-02
```

QR decomposition II

```
[4,]  1.419101e-03
[5,]  4.403070e-05
[6,]  3.986241e-07

$rank
[1] 6

$qraux
[1] 1.818850e+00 1.453471e+00 1.076453e+00 1.246653e+00 1.930492e+00
[6] 3.986070e-07

$pivot
[1] 1 2 3 4 5 6

attr(,"class")
[1] "qr"
```

QR decomposition

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

The upper triangle contains the R of the decomposition and the lower triangle contains information on the Q of the decomposition, stored in compact form.

The Q and R can be retrieved using `qr.Q()` and `qr.R()`, respectively.

```
R> Q <- qr.Q(H_qr)
R> R <- qr.R(H_qr)
```

We can then verify that Q is orthogonal and R is upper triangular:

EQUIS ACCREDITED AACSB ACCREDITED ASSOCIATION AMBA ACCREDITED

QR decomposition

```
R> crossprod(Q)
              [,1]          [,2]          [,3]          [,4]
[1,]  1.000000e+00  7.632783e-17  2.775558e-17 -2.775558e-17
[2,]  7.632783e-17  1.000000e+00  1.387779e-16  2.775558e-17
[3,]  2.775558e-17  1.387779e-16  1.000000e+00 -1.110223e-16
[4,] -2.775558e-17  2.775558e-17 -1.110223e-16  1.000000e+00
[5,]  0.000000e+00 -5.551115e-17 -8.326673e-17 -1.665335e-16
[6,]  6.938894e-18  1.387779e-17  0.000000e+00  1.110223e-16
              [,5]         [,6]
[1,]  0.000000e+00 6.938894e-18
[2,] -5.551115e-17 1.387779e-17
[3,] -8.326673e-17 0.000000e+00
[4,] -1.665335e-16 1.110223e-16
[5,]  1.000000e+00 2.775558e-17
[6,]  2.775558e-17 1.000000e+00
```

QR decomposition

```
R> R
```

```
         [,1]       [,2]         [,3]          [,4]          [,5]
[1,] -1.221224 -0.7018717 -0.504470316 -0.3969691267 -3.284337e-01
[2,]  0.000000 -0.1384670 -0.151130170 -0.1443643562 -1.340082e-01
[3,]  0.000000  0.0000000 -0.009561613 -0.0151932381 -1.813029e-02
[4,]  0.000000  0.0000000  0.000000000  0.0004802815  9.942382e-04
[5,]  0.000000  0.0000000  0.000000000  0.0000000000  1.733898e-05
[6,]  0.000000  0.0000000  0.000000000  0.0000000000  0.000000e+00
              [,6]
[1,] -2.806128e-01
[2,] -1.236690e-01
[3,] -1.953170e-02
[4,]  1.419101e-03
[5,]  4.403070e-05
[6,]  3.986241e-07
```

QR decomposition

How well can we recover H_6 from its QR decomposition?

```
R> max(abs((Q %*% R) - H))
[1] 2.220446e-16
```

(not bad).

QR decomposition

To solve $H_6 x = b$ using the QR decomposition "by hand", we can do

```
R> x3a <- c(backsolve(R, crossprod(Q, b)))
R> ## Compare to result of qr.solve():
R> x3a - x3

[1]  2.904699e-11 -1.182343e-10  6.839400e-10 -1.746230e-09
[5]  1.833541e-09 -6.839400e-10
```

It is more correct to compute $Q'b$ in one step:

```
R> x3a <- c(backsolve(R, qr.qty(H_qr, b)))
R> ## Compare to result of qr.solve():
R> x3a - x3

[1] 0 0 0 0 0 0
```

So this is what `qr.solve()` does.

QR decomposition

How can we find the (absolute value) of the determinant of a matrix from its QR decomposition?

QR decomposition

How can we find the (absolute value) of the determinant of a matrix from its QR decomposition?

Clearly.

$$\det(A) = \det(Q)\det(R)$$

where $\det(Q) = \pm 1$ and $\det(R)$ is the product of the diagonal elements of R.

Hence, $|\det(A)|$ is

$$\mathrm{prod}(\mathrm{diag}(R))$$

which is rather close to zero (so H_6 is close to singular)!

Outline

- Matrix basics
- Matrix decompositions and linear systems
  - Introduction
  - LU decomposition
  - QR decomposition
  - Singular value decomposition (SVD)
  - Eigendecomposition
  - Choleski decomposition
  - Summary

Singular value decomposition (SVD)

The SVD of a quadratic matrix is

$$A = UDV',$$

where U and V are orthogonal and $D = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$ is diagonal with non-negative entries.

Singular value decomposition (SVD)

The SVD of a quadratic matrix is

$$A = UDV',$$

where U and V are orthogonal and $D = \mathrm{diag}(\sigma_1, \ldots, \sigma_n)$ is diagonal with non-negative entries.

Note 1: the SVD also works for rectangular $m \times n$ matrices. In this cases D is "rectangular diagonal".

Note 2: the SVD also works for complex matrices. In this case U and V are unitary.

Note 3: If A has rank r, there is also the *compact* SVD $A = U_r D_r V_r'$, where U_r is $m \times r$, D_r is $r \times r$ diagonal, and V_r is $n \times r$, with $U_r' U_r = V_r' V_r = I_r$.

Singular value decomposition (SVD)

Let us first understand the SVD.

As $U'U = I$, we have

$$A'A = (UDV')'UDV' = VDU'UDV' = VD^2V'$$

where $D^2 = D \cdot D = \text{diag}(\sigma_1^2, \ldots, \sigma_n^2)$.

Thus,

$$A'AV = VD^2V'V = VD^2.$$

Singular value decomposition (SVD)

Write v_j for the j-th column of V. Then

$$A'AV = A'A[v_1, \dots, v_n] = [A'Av_1, \dots, A'Av_n]$$

and

$$VD^2 = [v_1, \dots, v_n]\,\mathrm{diag}(\sigma_1^2, \dots, \sigma_n^2) = [\sigma_1^2 v_1, \dots, \sigma_n^2 v_n].$$

Putting together, for all j

$$A'Av_j = \sigma_j^2 v_j.$$

Singular value decomposition (SVD)

Write v_j for the j-th column of V. Then

$$A'AV = A'A[v_1, \dots, v_n] = [A'Av_1, \dots, A'Av_n]$$

and

$$VD^2 = [v_1, \dots, v_n]\,\mathrm{diag}(\sigma_1^2, \dots, \sigma_n^2) = [\sigma_1^2 v_1, \dots, \sigma_n^2 v_n].$$

Putting together, for all j

$$A'Av_j = \sigma_j^2 v_j.$$

I.e., the columns v_j of V are the *eigenvectors* of $A'A$, and the singular values σ_j^2 the corresponding eigenvalues.

Singular value decomposition (SVD)

Similarly,

$$AA' = UDV'(UDV')' = UDV'VDU' = UD^2U'$$

so that

$$AA'U = UD^2U'U = UD^2.$$

Thus, writing u_j for the j-th column of U, we have

$$AA'u_j = \sigma_j^2 u_j$$

so that the u_j are the eigenvectors of AA' and the σ_j^2 the corresponding eigenvalues.

Singular value decomposition (SVD)

What is the geometric interpretation of the SVD?

Singular value decomposition (SVD)

What is the geometric interpretation of the SVD?

If U is orthogonal, $x \mapsto Ux$ performs a *rotation*.

Singular value decomposition (SVD)

What is the geometric interpretation of the SVD?

If U is orthogonal, $x \mapsto Ux$ performs a *rotation*.

If D is diagonal, $x \mapsto Dx$ performs *coordinate scaling*.

Singular value decomposition (SVD)

What is the geometric interpretation of the SVD?

If U is orthogonal, $x \mapsto Ux$ performs a *rotation*.

If D is diagonal, $x \mapsto Dx$ performs *coordinate scaling*.

Hence, if A has SVD UDV',

$$x \mapsto Ax = UDV'x$$

factors the linear transformation corresponding to A into a rotation, a scaling, and another rotation.

Singular value decomposition (SVD)

In R, we can compute the SVD via `svd()`, which returns things "as expected":

```
R> H_svd <- svd(H)
R> typeof(H_svd)
```

```
[1] "list"
```

```
R> length(H_svd)
```

```
[1] 3
```

```
R> names(H_svd)
```

```
[1] "d" "u" "v"
```

Singular value decomposition (SVD) I

```
R> H_svd

$d
[1] 1.618900e+00 2.423609e-01 1.632152e-02 6.157484e-04 1.257076e-05
[6] 1.082799e-07

$u
           [,1]       [,2]       [,3]        [,4]        [,5]
[1,] -0.7487192  0.6145448 -0.2403254 -0.06222659  0.01114432
[2,] -0.4407175 -0.2110825  0.6976514  0.49083921 -0.17973276
[3,] -0.3206969 -0.3658936  0.2313894 -0.53547692  0.60421221
[4,] -0.2543114 -0.3947068 -0.1328632 -0.41703769 -0.44357472
[5,] -0.2115308 -0.3881904 -0.3627149  0.04703402 -0.44153664
[6,] -0.1814430 -0.3706959 -0.5027629  0.54068156  0.45911482
             [,6]
[1,] -0.001248194
```

Singular value decomposition (SVD) II

```
[2,]  0.035606643
[3,] -0.240679080
[4,]  0.625460387
[5,] -0.689807199
[6,]  0.271605453

$v
           [,1]       [,2]       [,3]        [,4]        [,5]
[1,] -0.7487192  0.6145448 -0.2403254 -0.06222659  0.01114432
[2,] -0.4407175 -0.2110825  0.6976514  0.49083921 -0.17973276
[3,] -0.3206969 -0.3658936  0.2313894 -0.53547692  0.60421221
[4,] -0.2543114 -0.3947068 -0.1328632 -0.41703769 -0.44357472
[5,] -0.2115308 -0.3881904 -0.3627149  0.04703402 -0.44153664
[6,] -0.1814430 -0.3706959 -0.5027629  0.54068156  0.45911482
             [,6]
[1,] -0.001248194
[2,]  0.035606643
```

Singular value decomposition (SVD) III

```
[3,] -0.240679080
[4,]  0.625460387
[5,] -0.689807199
[6,]  0.271605453
```

Singular value decomposition (SVD)

Extract the elements of the SVD:

```
R> U <- H_svd$u
R> s <- H_svd$d
R> V <- H_svd$v
```

Verify that U and V are orthogonal:

Singular value decomposition (SVD)

```
R> crossprod(U)
              [,1]          [,2]          [,3]          [,4]
[1,]  1.000000e+00 -8.326673e-17  8.326673e-17  0.000000e+00
[2,] -8.326673e-17  1.000000e+00 -2.775558e-17  0.000000e+00
[3,]  8.326673e-17 -2.775558e-17  1.000000e+00 -1.665335e-16
[4,]  0.000000e+00  0.000000e+00 -1.665335e-16  1.000000e+00
[5,] -1.387779e-17  8.326673e-17  2.498002e-16  1.665335e-16
[6,]  2.081668e-17  4.163336e-17 -5.551115e-17  5.551115e-17
              [,5]          [,6]
[1,] -1.387779e-17  2.081668e-17
[2,]  8.326673e-17  4.163336e-17
[3,]  2.498002e-16 -5.551115e-17
[4,]  1.665335e-16  5.551115e-17
[5,]  1.000000e+00 -4.163336e-17
[6,] -4.163336e-17  1.000000e+00
```

Singular value decomposition (SVD)

```
R> crossprod(V)
              [,1]          [,2]          [,3]          [,4]
[1,]  1.000000e+00  2.914335e-16 -2.914335e-16  1.387779e-17
[2,]  2.914335e-16  1.000000e+00 -2.775558e-16 -1.110223e-16
[3,] -2.914335e-16 -2.775558e-16  1.000000e+00 -1.110223e-16
[4,]  1.387779e-17 -1.110223e-16 -1.110223e-16  1.000000e+00
[5,]  1.387779e-17 -2.775558e-17 -1.665335e-16  8.326673e-17
[6,]  6.938894e-18  1.387779e-17  2.775558e-17 -2.220446e-16
              [,5]          [,6]
[1,]  1.387779e-17  6.938894e-18
[2,] -2.775558e-17  1.387779e-17
[3,] -1.665335e-16  2.775558e-17
[4,]  8.326673e-17 -2.220446e-16
[5,]  1.000000e+00  1.387779e-17
[6,]  1.387779e-17  1.000000e+00
```

Singular value decomposition (SVD)

More compactly,

```
R> max(abs(crossprod(U) - diag(6)))
[1] 5.551115e-16
R> max(abs(crossprod(V) - diag(6)))
[1] 1.110223e-15
```

Singular value decomposition (SVD)

How well can H_6 be recovered from its SVD?

Note that

$$U \operatorname{diag}(s) V' = \operatorname{cmult}(U, s) \cdot V' = \operatorname{tcrossprod}(\operatorname{cmult}(U, s), V).$$

Numerically,

```
R> max(abs(tcrossprod(cmult(U, s), V) - H))
[1] 2.220446e-16
```

which is quite impressive!

Singular value decomposition (SVD)

If A is regular with SVD $A = UDV'$, its inverse is given by

$$A^{-1} = (UDV')^{-1} = (V')^{-1} D^{-1} U^{-1} = V D^{-1} U'$$

where $D^{-1} = \mathrm{diag}(1/\sigma_1, \ldots, 1/\sigma_n)$.

Singular value decomposition (SVD)

If A is regular with SVD $A = UDV'$, its inverse is given by

$$A^{-1} = (UDV')^{-1} = (V')^{-1} D^{-1} U^{-1} = V D^{-1} U'$$

where $D^{-1} = \mathrm{diag}(1/\sigma_1, \ldots, 1/\sigma_n)$.

Geometrically, this makes perfect sense: to invert, need to invert the rotation by U, then the scaling by D, and finally the rotation by V'.

Singular value decomposition (SVD)

If A is regular with SVD $A = UDV'$, its inverse is given by

$$A^{-1} = (UDV')^{-1} = (V')^{-1} D^{-1} U^{-1} = V D^{-1} U'$$

where $D^{-1} = \mathrm{diag}(1/\sigma_1, \dots, 1/\sigma_n)$.

Geometrically, this makes perfect sense: to invert, need to invert the rotation by U, then the scaling by D, and finally the rotation by V'.

Writing $D = \mathrm{diag}(s)$, to compute

$$V D^{-1} U' b = V \,\mathrm{diag}(1/s) U' b$$

we can do

$$\mathrm{cmult}(V, 1/s) \cdot \mathrm{crossprod}(U, b).$$

Singular value decomposition (SVD)

To solve $H_6 x = b$ via the SVD, we can thus do

```
R> x4 <- cmult(V, 1 / s) %*% crossprod(U, b)
R> b4 <- H %*% x4
R> b4 - b

              [,1]
[1,]  1.091394e-11
[2,]  5.456968e-12
[3,]  5.456968e-12
[4,]  5.456968e-12
[5,]  3.637979e-12
[6,] -1.818989e-12
```

Again, very impressive.

Singular value decomposition (SVD)

Finally, clearly

$$\det(A) = \det(U)\det(D)\det(V') = \pm\det(D) = \pm\prod_j \sigma_j.$$

In our case, this gives $|\det(H_6)|$ as

```
R> prod(s)
[1] 5.3673e-18
```

(again, very small).

Outline

- Matrix basics
- Matrix decompositions and linear systems
  - Introduction
  - LU decomposition
  - QR decomposition
  - Singular value decomposition (SVD)
  - Eigendecomposition
  - Choleski decomposition
  - Summary

Eigendecomposition

The eigendecomposition (or spectral decomposition) of a *symmetric square* matrix A is

$$A = UDU'$$

where U is orthogonal and $D = \mathrm{diag}(\delta_1, \ldots, \delta_n)$ is diagonal.

Eigendecomposition

The eigendecomposition (or spectral decomposition) of a *symmetric square* matrix A is

$$A = UDU'$$

where U is orthogonal and $D = \mathrm{diag}(\delta_1, \ldots, \delta_n)$ is diagonal. Then

$$AU = UDU'U = UD$$

so writing u_j for the j-the column of U,

$$Au_j = \delta_j u_j.$$

Eigendecomposition

The eigendecomposition (or spectral decomposition) of a *symmetric square* matrix A is

$$A = UDU'$$

where U is orthogonal and $D = \mathrm{diag}(\delta_1, \ldots, \delta_n)$ is diagonal. Then

$$AU = UDU'U = UD$$

so writing u_j for the j-the column of U,

$$Au_j = \delta_j u_j.$$

I.e., the u_j are the eigenvectors of A, and the δ_j the corresponding eigenvalues.

Eigendecomposition

Note that the eigendecomposition can only work for symmetric A:

$$(UDU')' = (U')'D'U' = UDU'.$$

Note that for symmetric matrices, the eigendecomposition is “like the SVD”, but not quite the same: taking $V = U$ no longer allows to fix the signs of the elements in the diagonal matrix!

Eigendecomposition

Geometric interpretation: if A has eigendecomposition $A = UDU'$, then

$$x \mapsto UDU'x$$

perform rotation (by U'), scaling, and inverse rotation.

Eigendecomposition

Geometric interpretation: if A has eigendecomposition $A = UDU'$, then

$$x \mapsto UDU'x$$

perform rotation (by U'), scaling, and inverse rotation.

Clearly,

$$A^2 = UDU'UDU' = UD^2U'$$

and generally,

$$A^k = UD^kU'$$

where $D^k = \mathrm{diag}(\delta_1^k, \dots, \delta_n^k)$.

Eigendecomposition

In R, we can compute the eigendecomposition via `eigen()`, which again returns things “as expected”:

```
R> H_eigen <- eigen(H)
R> typeof(H_eigen)

[1] "list"

R> length(H_eigen)

[1] 2

R> names(H_eigen)

[1] "values"  "vectors"
```

Eigendecomposition I

```
R> H_eigen

eigen() decomposition
$values
[1] 1.618900e+00 2.423609e-01 1.632152e-02 6.157484e-04 1.257076e-05
[6] 1.082799e-07

$vectors
           [,1]       [,2]       [,3]        [,4]        [,5]
[1,] -0.7487192  0.6145448 -0.2403254 -0.06222659  0.01114432
[2,] -0.4407175 -0.2110825  0.6976514  0.49083921 -0.17973276
[3,] -0.3206969 -0.3658936  0.2313894 -0.53547692  0.60421221
[4,] -0.2543114 -0.3947068 -0.1328632 -0.41703769 -0.44357472
[5,] -0.2115308 -0.3881904 -0.3627149  0.04703402 -0.44153664
[6,] -0.1814430 -0.3706959 -0.5027629  0.54068156  0.45911482
             [,6]
```

Eigendecomposition II

```
[1,] -0.001248194
[2,]  0.035606643
[3,] -0.240679080
[4,]  0.625460387
[5,] -0.689807199
[6,]  0.271605453
```

Eigendecomposition

Extract the elements of the eigendecomposition:

```
R> U <- H_eigen$vectors
R> d <- H_eigen$values
```

Verify that U is orthogonal:

```
R> max(abs(crossprod(U) - diag(6)))
```

```
[1] 4.996004e-16
```

Eigendecomposition

How well can H_6 be recovered from its eigendecomposition?

As before,

```
R> max(abs(tcrossprod(cmult(U, d), U) - H))
[1] 6.661338e-16
```

Eigendecomposition

If A is regular with eigendecomposition $A = UDU'$, its inverse is given by

$$A^{-1} = (UDU')^{-1} = (U')^{-1}D^{-1}U^{-1} = UD^{-1}U'$$

where $D^{-1} = \mathrm{diag}(1/\delta_1, \ldots, 1/\delta_n)$.

Geometrically: rotate, invert the scaling, rotate back.

Eigendecomposition

As for the SVD, we can thus solve via eigendecomposition as

```
R> x5 <- cmult(U, 1 / d) %*% crossprod(U, b)
R> b5 <- H %*% x5
R> b5 - b
```

```
              [,1]
[1,] -3.637979e-12
[2,]  3.637979e-12
[3,] -3.637979e-12
[4,]  1.818989e-12
[5,] -1.818989e-12
[6,]  0.000000e+00
```

(Of course, we get the same as for the SVD.)

Eigendecomposition

Finally, clearly

$$\det(A) = \det(U)\det(D)\det(U') = \det(D) = \prod_j \delta_j$$

In our case, this gives $\det(H_6)$ as

```
R> prod(d)
```

```
[1] 5.3673e-18
```

Did we already point out that this rather small?

Outline

- Matrix basics
- Matrix decompositions and linear systems
  - Introduction
  - LU decomposition
  - QR decomposition
  - Singular value decomposition (SVD)
  - Eigendecomposition
  - Choleski decomposition
  - Summary

Choleski decomposition

The Choleski decomposition of a non-negative definite symmetric square matrix A is

$$A = LL'$$

where L is lower triangular.

Equivalently (as used by R),

$$A = R'R$$

where R is upper triangular.

Note: named after the French military officer and mathematician André-Louis Cholesky (Wikipedia writes a ‘y’ at the end, the R docs write ‘i’).

Choleski decomposition

Note that the Choleski decomposition can only work for non-negative definite symmetric matrices:

$$A = R'R \Rightarrow A' = (R'R)' = R'(R')' = R'R = A$$

and

$$x'Ax = x'R'Rx = (Rx)'(Rx) = \|Rx\|_2^2 \geq 0.$$

Choleski decomposition

In R, we can compute the Choleski decomposition/factor using `chol()`:

```
R> (R <- chol(H))
     [,1]      [,2]      [,3]       [,4]        [,5]        [,6]
[1,]    1 0.5000000 0.3333333 0.25000000 0.200000000 0.166666667
[2,]    0 0.2886751 0.2886751 0.25980762 0.230940108 0.206196525
[3,]    0 0.0000000 0.0745356 0.11180340 0.127775313 0.133099284
[4,]    0 0.0000000 0.0000000 0.01889822 0.037796447 0.052495066
[5,]    0 0.0000000 0.0000000 0.00000000 0.004761905 0.011904762
[6,]    0 0.0000000 0.0000000 0.00000000 0.000000000 0.001196474
```

Choleski decomposition

How well can we recover H_6 from its Choleski factor?

```
R> crossprod(R) - H

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    0    0    0
[2,]    0    0    0    0    0    0
[3,]    0    0    0    0    0    0
[4,]    0    0    0    0    0    0
[5,]    0    0    0    0    0    0
[6,]    0    0    0    0    0    0
```

Choleski decomposition

The linear system $Ax = R'Rx = b$ can be solved via the Choleski decomposition as

$$R'y = b, \quad Rx = y$$

via

$$\text{backsolve}(R, \text{forwardsolve}(R', b)).$$

Choleski decomposition

To solve $H_6 x = b$ via the Choleski decomposition, we can thus do

```
R> x6 <- backsolve(R, forwardsolve(t(R), b))
R> b6 <- H %*% x6
R> b6 - b

              [,1]
[1,]  1.818989e-12
[2,]  1.091394e-11
[3,]  5.456968e-12
[4,]  1.818989e-12
[5,] -1.818989e-12
[6,]  2.728484e-12
```

Choleski decomposition

Finally, what about the determinant?

$$\det(A) = \det(R'R) = \det(R')\det(R) = \det(R)^2 = (\mathrm{prod}(\mathrm{diag}(R)))^2.$$

In our case, this gives $\det(H_6)$ as

```
R> prod(diag(R)) ^ 2
[1] 5.3673e-18
```

Outline

- Matrix basics
- Matrix decompositions and linear systems
  - Introduction
  - LU decomposition
  - QR decomposition
  - Singular value decomposition (SVD)
  - Eigendecomposition
  - Choleski decomposition
  - Summary

Summary

The solutions we obtained were rather different:

```
R> dist(t(cbind(x1, x2, x3, x4, x5, x6)), "maximum")
```

```
              x1           x2           x3
x2 5.567017e-09
x3 1.414525e-05 1.414568e-05
   1.146636e-05 1.146679e-05 2.678891e-06
   2.000082e-05 2.000039e-05 3.414607e-05 3.146718e-05
x6 8.379517e-06 8.379080e-06 2.252476e-05 1.984587e-05 1.162131e-05
```

Summary

Qualitatively, the "straightforward" translation of $A^{-1}b$ works worst:

```
R> apply(abs(cbind(b1, b2, b3, b4, b5, b6) - b), 2, max)

[1] 6.311893e-10 3.637979e-12 5.456968e-12 1.091394e-11 3.637979e-12
[6] 1.091394e-11
```

Interestingly, for a simple 6×6 system with apparently an all-integer solution the solutions are "not too good":

```
R> X <- cbind(x1, x2, x3, x4, x5, x6)
R> apply(abs(X - round(X)), 2, max)

          x1           x2           x3
2.104789e-05 2.104833e-05 6.902643e-06 9.581534e-06 4.104871e-05
          x6
2.942741e-05
```

Summary

How come?

Summary

How come?

Well, we repeatedly showed that $\det(H_6) \approx 10^{-18}$. So, in some sense, H_6 is “close to singular”, which has consequences.

Intuitively, the closer the det is to zero, the closer to singular.

Mathematically, what matters (most) is how “well-conditioned” a linear system is, which can be measured by its condition number.

See the homeworks.