Goals

We'll consider computations involving dense matrices, those that don't have a large number of zero elements.

We need to

1. be able to efficiently manipulate matrices on computers.

2. be able to choose the right decomposition for a given task.

The Plan

- Matrix manipulation

    - how matrices are stored on computers
    - basic tools for matrix manipulation: the BLAS

- Matrix decompositions and their uses

    - LU
    - QR
      Case study: data fitting
    - rank-revealing QR
    - eigendecomposition
      Case study: matrix stability
    - SVD
      Case study: solving integral equations

- Updating decompositions

- Some tasks to avoid

    - matrix inverse
    - the matrix permanent
    - Jordan canonical form

- Software

Notation

- All vectors are column vectors.

- Matrices are upper case letters; vectors and scalars are lower case.

- The elements of a matrix or vector will be denoted by subscripted values: the element of $\mathbf{A}$ in row i and column j is a_{ij} or $A(i,j)$.

- The elements of matrices and vectors can be real or complex numbers.

- $\mathbf{I}$ is the identity matrix and $\mathbf{e}_i$ is the ith unit vector, the ith column of $\mathbf{I}$.

- $\mathbf{B} = \mathbf{A}^T$ means that $\mathbf{B}$ is the transpose of $\mathbf{A}$: $b_{ij} = a_{ji}$.

- $\mathbf{B} = \mathbf{A}^*$ means that $\mathbf{B}$ is the complex conjugate transpose of $\mathbf{A}$: $b_{ij} = \bar{a}_{ji}$.

- We'll also use MATLAB notation when convenient. For example, $\mathbf{A}(i:j, k:\ell)$ denotes the submatrix of $\mathbf{A}$ with row entries between i and j and column entries between k and ℓ (inclusive), and $\mathbf{A}(:,5)$ denotes column 5 of the matrix $\mathbf{A}$.

- An orthogonal matrix $\mathbf{U}$ satisfies the relation $\mathbf{U}^T\mathbf{U} = \mathbf{I}$.

- A unitary matrix $\mathbf{U}$ satisfies the relation $\mathbf{U}^*\mathbf{U} = \mathbf{I}$. Aside: Quantum computers are implementations of matrix multiplication by a unitary matrix.

- For any nonsingular matrix $\mathbf{X}$, the matrix $\mathbf{X}\mathbf{A}\mathbf{X}^{-1}$ is a similarity transform of $\mathbf{A}$.

Matrix manipulation

See Chapter 3.

How matrices are stored on computers

It turns out that programmers of matrix algorithms need to design their algorithms based on how matrices are stored.

We'll illustrate the idea using a simple algorithm, matrix-vector product.

Matrix-vector multiplication

Suppose we have an $m \times n$ matrix $\mathbf{A}$ and an $n \times 1$ vector $\mathbf{x}$, and we wish to form $\mathbf{y} = \mathbf{A}\mathbf{x}$.

- The vector $\mathbf{y}$ is defined by dot products between rows of $\mathbf{A}$ and $\mathbf{x}$:

$$y_i = \mathbf{A}(i,:) * \mathbf{x}\,.$$

Expressed element-by-element, the algorithm is

```
[m,n]=size(A);
y = zeros(m,1);
for i=1:m,
   for j=1:n,
       y(i) = y(i) + A(i,j)*x(j);
   end
end
```

- We can also express **Ax** in a column-oriented way, using saxpys:

$$\mathbf{Ax} = x(1) * \mathbf{A}(:,1) + x(2) * \mathbf{A}(:,2) + \ldots + x(n) * \mathbf{A}(:,n) \,.$$

This can be implemented element-by-element as

```
[m,n]=size(A);
y = zeros(m,1);
for j=1:n,
   for i=1:m,
       y(i) = y(i) + A(i,j)*x(j);
   end
end
```

What difference does it make?

Both algorithms take the same number of numeric operations: mn multiplications and mn additions.

Note: when we count operations, we are only concerned with the highest order term, since that is what matters when the matrix is very large. Thus, if the number of additions is $n^2 - n$, we will just say n^2.

Computers store information on pages of memory. Some of this information is kept handy, in cache memory. What doesn't fit there is kept in main memory, which is bigger but takes more time to access. And the leftovers are stored on disk.

To use information, it must be moved to cache, and some other page must be moved out of cache, so to make an algorithm fast, we need to make sure that we limit the number of times a page is moved into cache.

Otherwise, speed will be nothing like the megaflop rating of the floating point chip.

Note: I have blurred the distinction between blocks, which are the units of cache storage, and the larger pages or segments, which are the units of disk storage, but the point is that we need to be mindful of which matrix elements can be accessed quickly and which take more time.

Now suppose that you have stored the matrix one column per page. Which algorithm would you prefer to use?

Unquiz: Count the number of page changes for each algorithm when $m = n = 1000$ and one column is stored per page. []

Matrix storage

Therefore, if your algorithm involves matrices, the first question to ask about your programming language is are matrices stored by row or by column?

Language	Storage scheme
C, C++	by row
FORTRAN	by column
Java	by row
MATLAB	by column

As we just saw, the storage scheme influences the choice of algorithm. More about this in AMSC/CMSC 662.

Basic Tools for Matrix Manipulation: the BLAS

See Section 5.1.

Basic tools for matrix manipulation: the BLAS

There are certain tasks we do all the time:

- navigate to class

- get dressed

- send email

and we develop shortcuts to minimize the amount of thought and effort that we need to give them.

There are certain tasks that are common to many matrix problems, so shortcuts have been developed so that programmers don't need to redo the work each time.

These Basic Linear Algebra Subroutines or BLAS are now available for all of the standard languages.

- bug-free code

- only need one line to accomplish a complicated task

- execution optimized

Examples:

- Level-1 BLAS: vector operations

  1. sscal computes $a\mathbf{x}$ where a is a scalar and $\mathbf{x}$ is a vector
  2. saxpy computes $a\mathbf{x} + \mathbf{y}$
  3. sdot computes $\mathbf{x}^*\mathbf{y}$

- Level-2 BLAS: matrix-vector operations

  1. matrix-vector product
  2. low-rank updates to a matrix
  3. solution of linear systems involving triangular matrix

- Level-3 BLAS: matrix-matrix operations

  1. matrix-matrix product
  2. solution of multiple linear systems involving a triangular matrix

When a BLAS exists for a task you need, it is a good idea to use it in your algorithm!

MATLAB automatically uses the BLAS. In other languages, you need to call on the subroutines yourself.

Matrix decompositions and their uses

Matrix decompositions and their uses

- LU

- QR

- rank-revealing QR

- eigendecomposition

- SVD

The LU Decomposition

See Section 5.2.

Definition: The LU decomposition of an $n \times n$ matrix **A** is defined by $\mathbf{PA} = \mathbf{LU}$ where

- **P** is a permutation matrix,

- **L** is a unit lower triangular matrix (zero above the main diagonal and ones on the main diagonal), and

- **U** is an upper triangular matrix (zero below the main diagonal).

They are computed in the process of Gauss Elimination.

How to compute the LU decomposition

- Recall from your previous course that the matrix **A** is reduced to upper triangular form by putting zeros below the main diagonal, one column at a time, by subtracting a multiple of the current pivot row from all rows below it.

- The multipliers form the entries of **L**.

- For stability, it is necessary to pivot, or interchange rows, to keep the current main diagonal element greater or equal in magnitude to the elements below it. The record of interchanges defines **P**.

Example: See 460 notes.

In MATLAB: [L,U,P] = lu(A) or [PtL,U] = lu(A) to compute $\mathbf{P}^T\mathbf{L}$ and **U**.

Cost of LU

$n^3/3$ multiplications

Uses of LU

- We can use the decomposition to solve linear systems of equations

$$\mathbf{Ax} = \mathbf{b}$$

given $\mathbf{A}$ and $\mathbf{b}$. If we factor $\mathbf{PA}$ as $\mathbf{LU}$, then we know that $\mathbf{PAx} = \mathbf{Pb}$, so

$$\mathbf{LUx} = \mathbf{Pb}$$

or, if we define $\mathbf{y} = \mathbf{Ux}$,

$$\mathbf{Ly} = \mathbf{Pb}\,.$$

Therefore, we can solve $\mathbf{Ax} = \mathbf{b}$ by forward substitution on $\mathbf{Ly} = \mathbf{Pb}$ followed by back substitution on $\mathbf{Ux} = \mathbf{y}$.

In MATLAB: The "backslash" command x = A \ b uses the LU decomposition to solve the problem.

- We can easily compute the determinant of the matrix, using facts about determinants:

  - The determinant of a matrix $\mathbf{A}$ is defined to be

  $$\det(\mathbf{A}) = \sum_{\sigma} \text{sgn}(\sigma) \prod_{i=1}^{n} a_{i\sigma(i)}$$

  where the summation runs over all permutations of the indices $\{1, 2, \ldots, n\}$, and $\text{sgn}(\sigma)$ is $+1$ if the number of interchanges in the permutation is even and -1 if it is odd.
  - The determinant of a matrix equals the product of its eigenvalues.
  - The eigenvalues of a triangular matrix appear on the main diagonal.
  - The determinant of a product of two matrices equals the product of the determinants.

  Therefore, since $\det(\mathbf{P}^T) = \pm 1$,

  $$\det(\mathbf{A}) = \det(\mathbf{P}^T\mathbf{LU}) = \det(\mathbf{P}^T)\det(\mathbf{L})\det(\mathbf{U}) = \det(\mathbf{P}^T)\prod_{i=1}^{n} l_{ii} \prod_{i=1}^{n} u_{ii}\,.$$

Limitations of LU

Use Cholesky if the matrix is symmetric, positive definite. This gives a decomposition as $\mathbf{LL}^T$ or $\mathbf{LDL}^T$ at half the cost.

The QR Decomposition

See Section 5.3.

The QR Decomposition

Definition: The QR decomposition of an $m \times n$ matrix $\mathbf{A}$ ($m \geq n$) is defined by $\mathbf{A} = \mathbf{QR}$ where

- **Q** is an $m \times m$ unitary matrix (orthogonal, if **A** is real) and **R** is an $m \times n$ matrix with zeros below the main diagonal or

- **Q** is an $m \times n$ unitary matrix (orthogonal, if **A** is real) and **R** is an $n \times n$ upper triangular matrix.

How to compute QR

We'll discuss two algorithms:

- Givens rotations, (good for **Q** $m \times m$)

- Gram-Schmidt (good for **Q** $m \times n$).

QR decomposition by Givens rotations

A simple orthogonal matrix, a rotation, can be used to introduce one zero at a time into a real matrix.

We'll write the Givens matrix as

$$\mathbf{G} = \left[\begin{array}{cc} c & s \\ -s & c \end{array} \right]$$

where $c^2 + s^2 = 1$. (Thus, c and s have the geometric interpretation of the cosine and sine of an angle.)

A vector multiplied by **G** is rotated through an angle θ.

How Givens rotations can be used

Problem: Given a vector $\mathbf{z} \neq \mathbf{0}$ of dimension 2×1, find **G** so that $\mathbf{Gz} = x\mathbf{e}_1$ where $x = \|\mathbf{z}\|$.

Solution:
$$\mathbf{Gz} = \left[\begin{array}{c} cz_1 + sz_2 \\ -sz_1 + cz_2 \end{array} \right] = x\mathbf{e}_1$$

Multiplying the first equation by c, the second by s, and subtracting yields

$$(c^2 + s^2)z_1 = cx \,,$$

so

$$c = z_1/x \,.$$

Similarly, we can determine that

$$s = z_2/x \,.$$

Since $c^2 + s^2 = 1$, we conclude that

$$z_1^2 + z_2^2 = x^2 \,,$$

so

$$
\begin{aligned}
c &= \frac{z_1}{\sqrt{z_1^2 + z_2^2}} \\
s &= \frac{z_2}{\sqrt{z_1^2 + z_2^2}}
\end{aligned}
$$

(A similar derivation can be done if $\mathbf{z}$ is complex.)

Givens QR decomposition

So now we know how to use Givens matrices to zero out single components of a matrix. We will use the notation $\mathbf{G}_{ij}$ to denote an $n \times n$ identity matrix with its ith and jth rows modified to include the Givens rotation: for example, if $n = 6$, then

$$
\mathbf{G}_{25} =
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & c & 0 & 0 & s & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & -s & 0 & 0 & c & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} ,
$$

and multiplication of a vector by this matrix leaves all but rows 2 and 5 of the vector unchanged.

So we can reduce a matrix $\mathbf{A}$ to upper trapezoidal form by the following sequence of rotations:

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

    for $j = i + 1, \ldots, m$,

        Choose the matrix $\mathbf{G}_{ij}$ to zero out position (j, i) of the matrix, using the current value in position (i, i):

$$\mathbf{A} \leftarrow \mathbf{G}_{ij}\mathbf{A}$$

    end for

end for

QR by Gram-Schmidt

From the columns $[\mathbf{a}_1, \ldots, \mathbf{a}_n]$ of the matrix $\mathbf{A}$, we create an orthonormal basis $\{\mathbf{q}_1, \ldots, \mathbf{q}_n\}$ and save the coefficients that accomplish this goal in an upper triangular matrix $\mathbf{R}$.

Set $r_{11} = \|\mathbf{a}_1\|$.
Set $\mathbf{q}_1 = \mathbf{a}_1/r_{11}$.
for $k = 1, \ldots n - 1$,

    Set $\mathbf{q}_{k+1} = \mathbf{a}_{k+1}$.
    for $i = 1, \ldots, k$,

        $r_{i,k+1} = \mathbf{q}_i^* \mathbf{q}_{k+1}$
        $\mathbf{q}_{k+1} = \mathbf{q}_{k+1} - r_{i,k+1}\mathbf{q}_i$

    end for
    $r_{k+1,k+1} = \|\mathbf{q}_{k+1}\|$
    $\mathbf{q}_{k+1} = \mathbf{q}_{k+1}/r_{k+1,k+1}$

end for

Then you can convince yourself that we have formed a matrix factorization
$\mathbf{A} = \mathbf{QR}$.

Cost of QR

$2mn^2 - 2/3n^3$ multiplications, using Givens rotations.

$(mn^2 - 1/3n^3$ multiplications, using Householder reflections.)

mn^2 multiplications, using Gram-Schmidt.

Uses of QR

In MATLAB: `[Q,R] = qr(A)` for $\mathbf{A}$ of size $m \times n$, $m \geq n$

- `qr(A,0)` returns the compact matrix $\mathbf{Q}$ (although the full is computed).
  Algorithm: Householder transformations.

- See also `orth`, which only computes the compact $\mathbf{Q}$.

- A nice feature of the QR decomposition: in general, we don't need to
  "pivot" to preserve numerical stability.

  This makes QR a nice alternative to LU for solving linear systems.
  Although the operations count is twice as big, the data handling is simpler.

- QR can get the basis for the range of a full-rank matrix $\mathbf{A}$ (the first n
  columns of $\mathbf{Q}$) and the null-space of $\mathbf{A}^*$ (the last $m - n$ columns of $\mathbf{Q}$).

- QR can be used to solve linear least squares problems.

Solving linear least squares problems

$$\min_{\mathbf{x}} \| \mathbf{Ax} - \mathbf{b} \|$$

The MATLAB backslash command, x = A \ b, which solves a linear system when **A** is square and nonsingular, solves the problem in the least squares sense when **A** has more rows than columns.

Case Study: Data Fitting

Problem: Fit a model to data in order to reduce the effects of noise in the measurements.

Is a straight line a good fit to the data (t_i, f_i), $i = 1, \ldots, 10$?

Let

$$\mathbf{A} = \begin{bmatrix} 1 & t_1 \\ . & . \\ . & . \\ . & . \\ 1 & t_{10} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} f_1 \\ . \\ . \\ . \\ f_{10} \end{bmatrix}$$

```
sigma=.05
t = [1:10];
ve = ...    % the values of f
plot(t,ve,'g*')
hold on
for i=1:10,
    plot([t(i),t(i)],[ve(i)+2*sigma,ve(i)-2*sigma])
end
axis([0 11 -.2 1.2])
a = [ones(10,1),t'];
coef = a \ ve';
plot(t,a*coef,'m')
```

Limitations of QR

If the columns of **A** are linearly dependent, or are close to being linearly dependent, then the QR decomposition does not behave well.

The Rank-Revealing QR Decomposition
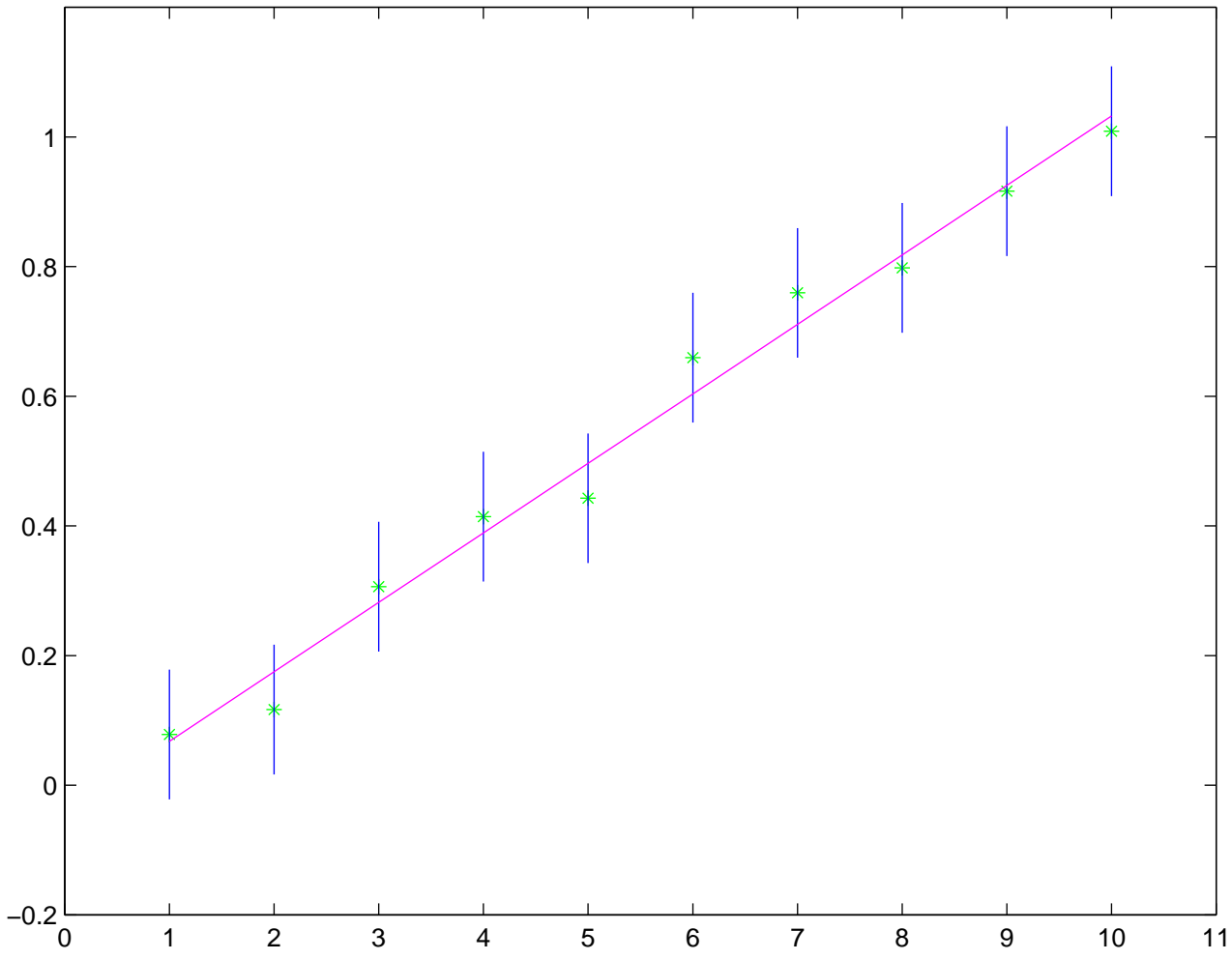
See Section 5.4.

The Rank-Revealing QR Decomposition (RR-QR)

Definition: The RR-QR decomposition of an $m \times n$ matrix **A** is defined by
AP = **QR** where **P** is a permutation matrix chosen so that the leading principal
submatrix of **R** of dimension $p \times p$ is well conditioned and the other diagonal
block (of dimension $(n - p) \times (n - p)$) is small. The numerical rank of **A** is p.

Note: We'll make use of this in solving systems of nonlinear equations.

How to compute RR-QR

for $i = 1 : n$,

> Among columns $i : n$ of the current **A** matrix, move the column with
> largest norm to the ith column.
> Perform Givens rotations to put zeros below row i in column i of **A**.

end

In MATLAB, `[Q,R,P] = qr(A,0)` produces a rank-revealing QR decomposition.

Cost of RR-QR

The number of multiplications is the same as for the Gram-Schmidt QR
algorithm or the Givens QR algorithm, whichever is used, but the pivoting
introduces extra overhead.

Uses of RR-QR

- determining whether a matrix has full rank

- a "poor-man's" principal component analysis (PCA), useful in contexts
  such as information retrieval.

Limitations of RR-QR

It can be fooled. Use SVD (higher cost) if reliability is essential.

Eigendecomposition

See Section 5.5.

Eigendecomposition

Definition: The eigendecomposition of a matrix $\mathbf{A}$ of dimension $n \times n$ is $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1}$ where $\mathbf{\Lambda}$ is a diagonal matrix with entries λ_i the eigenvalues. The columns of $\mathbf{U}$ are the right eigenvectors:

$$\mathbf{A}\mathbf{u}_i = \lambda_i \mathbf{u}_i \,.$$

The decomposition is guaranteed to exist if

- $\mathbf{A}$ is real symmetric or complex Hermitian, or
- the eigenvalues of $\mathbf{A}$ are distinct.

Otherwise, the decomposition may fail to exist, although it will exist for a nearby matrix.

How to compute the eigendecomposition

The basic algorithm is simple, but the refinements that make it work really well are not. We'll just focus on the basics. When you do an eigendecomposition, make sure you choose high-quality software to compute it.

The idea:

Step 1: Reduce the matrix $\mathbf{A}$ to compact form, so that it is easy to manipulate. Find a unitary matrix $\mathbf{V}$ so that

$$\mathbf{V}^*\mathbf{A}\mathbf{V} = \mathbf{H}$$

where $\mathbf{H}$ is

- tridiagonal if $\mathbf{A}$ is Hermitian (or real symmetric)
- upper Hessenberg otherwise.

This can be done in $O(n^3)$ operations.

Note that we have done a similarity transformation, so if we find an eigendecomposition of $\mathbf{H}$ as

$$\mathbf{H} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1}$$

then we have the eigendecomposition

$$\mathbf{A} = (\mathbf{V}\mathbf{U})\mathbf{\Lambda}(\mathbf{V}\mathbf{U})^{-1}$$

for $\mathbf{A}$.

Step 2: Find the eigendecomposition of $\mathbf{H}$ by QR iteration:

- Form $\mathbf{H} = \mathbf{QR}$.

- Replace $\mathbf{H}$ by $\mathbf{RQ}$.

Note that since $\mathbf{Q}^*\mathbf{Q} = \mathbf{I}$ and $\mathbf{H} = \mathbf{QR}$,

$$\mathbf{RQ} = (\mathbf{Q}^*\mathbf{Q})\mathbf{RQ} = \mathbf{Q}^*\mathbf{HQ}$$

so the new $\mathbf{H}$ has the same eigenvalues as the old one, and if we have an eigendecomposition of $\mathbf{RQ}$, then we have an eigendecomposition of $\mathbf{H}$.

The cost of this step is $O(n^2)$.

We repeat Step 2 many times (about $5n$, typically), and often some subdiagonal elements of $\mathbf{H}$ converge to zero. Once that happens, we can read some eigenvalues off the diagonal.

In order to ensure convergence of all subdiagonal elements and make it faster, there are many refinements to the algorithm, mostly involving shifting the matrix (by subtracting a multiple of the identity matrix) in order to emphasize one eigenvalue for the iteration.

In MATLAB: [U,Lambda] = eig(A)

Cost of eigendecomposition

The cost of Step 1 is $O(n^3)$, and the cost of $O(n)$ iterations of Step 2 is $O(n^3)$, so the total cost is $O(n^3)$.

Uses of eigendecomposition

These are many, but as a case-study, consider stability analysis.

Stability analysis

http://www.edmunds.com/ownership/safety/articles/45992/article.html

Consider the linear relations

$$\begin{aligned}
\mathbf{x}_{n+1} &= \mathbf{Ax}_n + \mathbf{Bu}_n \\
\mathbf{v}_n &= \mathbf{Cx}_n + \mathbf{Du}_n
\end{aligned}$$

where

- **A**, **B**, **C**, and **D** are matrices,
- $\mathbf{x}_n$ is the state vector of a system at time n
- $\mathbf{u}_n$ defines the controls used on the system at time n
- $\mathbf{v}_n$ is some observation of the system at time n.

Control theory is a complicated area, but the first thing to know about it is the importance of stability of the system. Without stability, small changes in the controls create arbitrarily large changes in **x**.

For stability, it is sufficient that $\|\mathbf{A}\| < 1$, where the norm is defined to be the square root of the largest eigenvalue of $\mathbf{A}^*\mathbf{A}$.

For instance, we might want to design a system with

$$\mathbf{A} = \begin{bmatrix} .5 & .4 & a \\ a & .3 & .4 \\ .3 & .3 & .3 \end{bmatrix}$$

where a is a parameter to be determined.

Then we could determine the range of values of a for which $\|\mathbf{A}\| < 1$.

Note: Other matrix stability problems involve determining whether all of the eigenvalues of a matrix lie in the left half-plane.

Limitations of the eigendecomposition

In general, the eigenvalues with large magnitude are computed more accurately than those of small magnitude.

Why: Due to rounding error, a stable algorithm will compute the exact eigendecomposition for some matrix close to the given matrix **A**. If **A** has an eigenvalue close to zero, the relative error in its computed value may be quite large.

The SVD

See Section 5.6.

The SVD

Definition: Every matrix **A** of dimensions $m \times n$ $(m \geq n)$ can be decomposed as

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^*$$

where

- **U** has dimension $m \times m$ and $\mathbf{U}^*\mathbf{U} = \mathbf{I}$,

- $\mathbf{\Sigma}$ has dimension $m \times n$, the only nonzeros are on the main diagonal, and they are nonnegative real numbers $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n$,

- **V** has dimension $n \times n$ and $\mathbf{V}^*\mathbf{V} = \mathbf{I}$.

Some useful relations

If $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^*$, then

$$\begin{aligned}
\mathbf{A}^*\mathbf{A} &= (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^*)^*\mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \\
&= \mathbf{V}\mathbf{\Sigma}^*\mathbf{U}^*\mathbf{U}\mathbf{\Sigma}\mathbf{V}^* = \mathbf{V}\mathbf{\Sigma}^*\mathbf{\Sigma}\mathbf{V}^*.
\end{aligned}$$

So we have a decomposition of $\mathbf{A}^*\mathbf{A}$ as a unitary matrix times a diagonal matrix times the conjugate transpose of the unitary matrix, and the diagonal matrix has entries σ_i^2.

Therefore,

- The singular values σ_i of **A** are the square roots of the eigenvalues of $\mathbf{A}^*\mathbf{A}$.

- The columns of **V** are the right singular vectors of **A** and the eigenvectors of $\mathbf{A}^*\mathbf{A}$.

- By forming $\mathbf{A}\mathbf{A}^*$, we would see that the columns of **U**, which are the left singular vectors of **A**, are the eigenvectors of $\mathbf{A}\mathbf{A}^*$.

How to compute the SVD

The algorithm is a variant on algorithms for computing eigendecompositions.

We won't give details, but note that it is rather complicated, so you would want to use a high-quality existing code rather than writing your own.

In MATLAB: `[U,S,V] = svd(A)`

Cost of the SVD

The cost is $O(mn^2)$ when $m \geq n$. The constant is of order 10.

Uses of the SVD

- solving ill-conditioned least squares problems

- solving discretized ill-posed problems

Case Study: Solving Integral Equations

Image deblurring.

See Chapter 6.

Limitations of SVD

Expensive but quite reliable!

- It is more expensive than our other options, but it is the right algorithm to use when one of them is inadequate.
- It can be used to solve almost all of the problems we have discussed!

So if you are stranded on a desert island and allowed only one piece of linear algebra software, which should you choose?

Updating Decompositions

See Chapter 7.

Updating decompositions

Sometimes our matrix changes in a rather simple way, and we want to reconsider our problem.

Do we need to recompute our decomposition or can we just update it at less cost?

Some example applications

- Suppose we are solving a system of linear inequalities

$$\mathbf{Ax} \geq \mathbf{b}$$

  with $\mathbf{A}$ of dimension $m \times n$ $(m \geq n)$ and we think that the first n of them should be active:

$$\mathbf{a}_i^T \mathbf{x} = b_i, \;\; i = 1, \ldots, n$$

  But we discover that the solution to these equations violates the kth inequality $(k > n)$ and we want to add it to our current system of equations and delete the 5th equation. Can we solve our new linear system easily?

  This problem routinely arises in minimization problems when we have linear inequality constraints, for example, in linear programming problems.

- Suppose that we are solving a linear least squares problem

$$\mathbf{Ax} \approx \mathbf{b}$$

  and we get some new measurements. This adds rows to $\mathbf{A}$ and $\mathbf{b}$. Can we solve our new least squares problem easily?

- Suppose we have computed the eigenvalues and eigenvectors of $\mathbf{A}$, and then $\mathbf{A}$ is changed by addition of a rank-1 matrix

$$\hat{\mathbf{A}} = \mathbf{A} + \mathbf{cr}^T .$$

  What are the eigenvalues of $\hat{\mathbf{A}}$?

Each of these problems (and similar ones) can be solved cheaply by updating (changing) the matrix decomposition of $\mathbf{A}$.

We can also solve the problem by making use of the original decomposition, without explicitly forming the update, and we'll illustrate this technique.

Linear systems with matrix updates

Suppose we have factored

$$\mathbf{PA} = \mathbf{LU}$$

and now we want to solve the linear system

$$(\mathbf{A} + \mathbf{cr}^T)\mathbf{x} = \mathbf{b} .$$

The Sherman-Morrison-Woodbury Formula

$$(\mathbf{A} - \mathbf{ZV}^T)^{-1} = \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{Z}(\mathbf{I} - \mathbf{V}^T\mathbf{A}^{-1}\mathbf{Z})^{-1}\mathbf{V}^T\mathbf{A}^{-1}$$

Unquiz: Show how to use the LU decomposition and the Sherman-Morrison-Woodbury formula to solve the linear system in $O(n^2)$ operations without forming any matrix inverses. []

Updating a QR decomposition

Suppose we have factored

$$\mathbf{A} = \mathbf{QR} .$$

For definiteness, we'll let $\mathbf{A}$ have dimensions 5×3.

We'll consider two kinds of changes that are common:

- adding a row (In least squares, this means that new data comes in.)

- deleting a column (In least squares, this means that we decided to reduce the number of parameters in the model.)

Adding a row

Our decomposition now looks like

$$\mathbf{A} = \left[\begin{array}{cc} \mathbf{Q} & \mathbf{0} \\ \mathbf{0} & 1 \end{array} \right] \left[\begin{array}{ccc} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ a_{61} & a_{62} & a_{63} \end{array} \right].$$

In order to complete our decomposition, we need to reduce the as to zeros. We can do this using n Givens rotations, much cheaper than recomputing the entire decomposition.

Deleting a column from a QR decomposition

As an example, if we delete column 1 from A, we can write the decomposition as

$$\hat{\mathbf{A}} = \mathbf{Q} \left[\begin{array}{cc} r_{12} & r_{13} \\ r_{22} & r_{23} \\ 0 & r_{33} \\ 0 & 0 \\ 0 & 0 \end{array} \right]$$

The resulting **R** is almost upper triangular; we just need rotations to reduce the elements labeled r_{22} and r_{33} to zero.

In general, we need $n - k$ rotations, when column k is deleted.

The point of updating

It may seem silly to worry so much about whether to update or recompute; computers are fast, and if we make one change to the matrix, it really doesn't matter which we do.

But when we need to do the task over and over again, in a loop that solves a more complicated problem, it is essential to use appropriate updating techniques, at a cost of $O(n^2)$ instead of recomputing at a cost of $O(n^3)$.

Note that updating can be unstable, though, so it is important to use stable and trusted algorithms.

See Section 5.7.

Some tasks to avoid

- matrix inverse

  We can solve
  $$\mathbf{Ax} = \mathbf{b}$$
  by multiplying both sides of the equation by $\mathbf{A}^{-1}$:
  $$\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}.$$

  Therefore, we can solve linear systems by multiplying the right-hand side $\mathbf{b}$ by $\mathbf{A}^{-1}$.

  This is a BAD idea. It is more expensive than the LU decomposition and it generally computes an answer that has larger error.

  Whenever you see a matrix inverse in a formula, think "LU decomposition".

  Never compute a matrix inverse unless you really want to look at the entries in it. Otherwise, find a decomposition that does the job. Your answer will generally be more accurate and more inexpensive than if you used the inverse matrix.

- Jordan canonical form

Software

Software

For computing matrix decompositions and solving matrix problems in FORTRAN or C, look for LAPACK software (more than 20 million downloads!). For Java, see `http://math.nist.gov/javanumerics/`.

- numerically stable algorithms.

- uniform interface, making use easy.

- row or column oriented implementation, appropriate for the matrix storage scheme used by the language.

- built on BLAS and thus efficient.

(Well, at least efficient when n is large (100 or more). The overhead for small n is rather big.)

Decomposition	Cost	Use
LU	$n^3/3$	- solving linear systems - computing determinants
QR	$mn^2 - 1/3n^3$	- solving well-conditioned linear least squares problems - representing the range or null-space of a matrix
rank-revealing QR	$mn^2 - 1/3n^3$	- solving ill-conditioned linear least squares problems - representing the range or null-space of a matrix

Decomposition	Cost	Use
SVD	$O(mn^2)$	- solving ill-conditioned linear least squares problems - solving discretizations of ill-posed problems - representing the range or null-space of a matrix
eigendecomposition	$O(n^3)$	- determining eigenvalues or eigenvectors of a matrix - determining invariant subspaces - determining stability of a control system - determining convergence

One important omission:

We haven't discussed sparse matrices, those with mostly zero entries.

These are very important in applications such as solution of partial differential equations and in economic modeling.

When you need to work with a large sparse matrix, it is essential to take advantage of the zeros, in order to make the work and the storage manageable.

For more about sparse matrices, see Unit VII.