EE364a Review Session 7

session outline:

- derivatives and chain rule (Appendix A.4)
- numerical linear algebra (Appendix C)
  - factor and solve method
  - exploiting structure and sparsity

Derivative and gradient

When f is real-valued ($i.e.$, $f : \mathbf{R}^n \to \mathbf{R}$), the derivative $Df(x)$ is a $1 \times n$ matrix, $i.e.$, it is a *row* vector.

- its transpose is called the *gradient* of the function:

$$\nabla f(x) = Df(x)^T,$$

  which is a (column) vector, $i.e.$, in $\mathbf{R}^n$

- its components are the partial derivatives of f:

$$\nabla f(x)_i = \frac{\partial f(x)}{\partial x_i}, \quad i = 1, \ldots, n$$

- the first-order approximation of f at a point x can be expressed as (the affine function of z)

$$f(x) + \nabla f(x)^T (z - x)$$

example: Find the gradient of $g : \mathbf{R}^m \to \mathbf{R}$,

$$g(y) = \log \sum_{i=1}^{m} \exp(y_i).$$

solution.

$$\nabla g(y) = \frac{1}{\sum_{i=1}^{m} \exp y_i} \begin{bmatrix} \exp y_1 \\ \vdots \\ \exp y_m \end{bmatrix}$$

Chain rule

Suppose $f : \mathbf{R}^n \to \mathbf{R}^m$ and $g : \mathbf{R}^m \to \mathbf{R}^p$ are differentiable. Define $h : \mathbf{R}^n \to \mathbf{R}^p$ by $h(x) = g(f(x))$. Then

$$Dh(x) = Dg(f(x))Df(x).$$

- Composition with an affine function:

  Suppose $g : \mathbf{R}^m \to \mathbf{R}^p$ is differentiable, $A \in \mathbf{R}^{m\times n}$, and $b \in \mathbf{R}^m$. Define $h : \mathbf{R}^n \to \mathbf{R}^p$ as $h(x) = g(Ax + b)$.

  The derivative of h is $Dh(x) = Dg(Ax + b)A$.

  When g is real-valued ($i.e.$, $p = 1$),

  $$\nabla h(x) = A^T \nabla g(Ax + b).$$

example A.2: Find the gradient of $h : \mathbf{R}^n \to \mathbf{R}$,

$$h(x) = \log \sum_{i=1}^{m} \exp(a_i^T x + b_i),$$

where $a_1, \ldots, a_m \in \mathbf{R}^n$, and $b_1, \ldots, b_m \in \mathbf{R}$.

Hint: h is the composition of the affine function $Ax + b$, where $A \in \mathbf{R}^{m\times n}$ has rows $a_1^T, \ldots, a_m^T$, and the function $g(y) = \log(\sum_{i=1}^m \exp y_i)$.

solution.

$$\nabla g(y) = \frac{1}{\sum_{i=1}^m \exp y_i} \begin{bmatrix} \exp y_1 \\ \vdots \\ \exp y_m \end{bmatrix}$$

then by the composition formula we have

$$\nabla h(x) = \frac{1}{\mathbf{1}^T z} A^T z$$

where $z_i = \exp(a_i^T x + b_i)$, $i = 1, \ldots, m$

Second derivative

When f is real-valued ($i.e.$, $f : \mathbf{R}^n \to \mathbf{R}$), the second derivative or Hessian matrix $\nabla^2 f(x)$ is a $n \times n$ matrix, with components

$$\nabla^2 f(x)_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}, \qquad i = 1, \ldots n, \quad j = 1, \ldots, n,$$

- the second-order approximation of f, at or near x, is the quadratic function of z defined by

$$\widehat{f}(z) = f(x) + \nabla f(x)^T (z - x) + (1/2)(z - x)^T \nabla^2 f(x)(z - x).$$

Chain rule for second derivative

- Composition with scalar function

  Suppose $f : \mathbf{R}^n \to \mathbf{R}$, $g : \mathbf{R} \to \mathbf{R}$, and $h(x) = g(f(x))$.

  The second derivative of h is

$$\nabla^2 h(x) = g'(f(x))\nabla^2 f(x) + g''(f(x))\nabla f(x)\nabla f(x)^T.$$

- Composition with affine function

  Suppose $g : \mathbf{R}^m \to \mathbf{R}$, $A \in \mathbf{R}^{m\times n}$, and $b \in \mathbf{R}^m$. Define $h : \mathbf{R}^n \to \mathbf{R}$ by $h(x) = g(Ax + b)$.

  The second derivative of h is

$$\nabla^2 h(x) = A^T \nabla^2 g(Ax + b)A.$$

example A.4: Find the Hessian of $h : \mathbf{R}^n \to \mathbf{R}$,

$$h(x) = \log \sum_{i=1}^{m} \exp(a_i^T x + b_i),$$

where $a_1, \ldots, a_m \in \mathbf{R}^n$, and $b_1, \ldots, b_m \in \mathbf{R}$.

Hint: For $g(y) = \log(\sum_{i=1}^m \exp y_i)$, we have

$$\begin{aligned}
\nabla g(y) &= \frac{1}{\sum_{i=1}^m \exp y_i} \begin{bmatrix} \exp y_1 \\ \vdots \\ \exp y_m \end{bmatrix} \\
\nabla^2 g(y) &= \mathbf{diag}(\nabla g(y)) - \nabla g(y) \nabla g(y)^T.
\end{aligned}$$

solution. using the chain rule for composition with affine function,

$$\nabla^2 h(x) = A^T \left(\frac{1}{\mathbf{1}^T z} \mathbf{diag}(z) - \frac{1}{(\mathbf{1}^T z)^2} z z^T \right) A$$

where $z_i = \exp(a_i^T x + b_i)$, $i = 1, \ldots, m$

Numerical linear algebra

factor-solve method for $Ax = b$

- consider set of n linear equations in n variables, $i.e.$, A is square
- computational cost $f + s$
  - f is flop count of factorization
  - s is flop count of solve step
- for single factorization and k solves, computational cost is $f + ks$

LU factorization

- nonsingular matrix A can be decomposed as $A = PLU$
- $f = (2/3)n^3$ (Gaussian elimination)
- $s = 2n^2$ (forward and back substitution)
- for example, can compute $n \times n$ matrix inverse with cost $f + ns = (8/3)n^3$ (why?)

  solution.

  - write $AX = I$ as $A[x_1 \cdots x_n] = [e_1 \cdots e_n]$
  - then solve $Ax_i = e_i$ for $i = 1, \ldots, n$

Cholesky factorization

- symmetric, positive definite matrix A can be decomposed as $A = LL^T$
- $f = (1/3)n^3$
- $s = 2n^2$
- prob. 9.31a: only factor once every N iterations, but solve every iteration
  - every N steps, computation is $f + s = (1/3)n^3 + 2n^2$ flops
  - all other steps, computation is $s = 2n^2$ flops

Exploiting structure

computational costs for solving $Ax = b$

structure of A	f	s
none	$(2/3)n^3$	$2n^2$
symmetric, positive definite	$(1/3)n^3$	$2n^2$
lower triangular	0	n^2
k-banded ($a_{ij} = 0$ if $\lvert i - j\rvert > k$)	$4nk^2$	$6nk$
block diag with m blocks	$(2/3)n^3/m^2$	$2n^2/m$
DFT (using FFT to solve)	0	$5n \log n$

Block elimination

solve

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

- first equation $A_{11}x_1 + A_{12}x_2 = b_1$ gives us

$$x_1 = A_{11}^{-1}(b_1 - A_{12}x_2)$$

- second equation is then

$$(A_{22} - A_{21}A_{11}^{-1}A_{12})x_2 = b_2 - A_{21}A_{11}^{-1}b_1.$$

- speedup if A_{11} and $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ are easy to invert

example: Solve the set of equations

$$\begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix} x = \begin{bmatrix} b \\ c \end{bmatrix}$$

where $A \in \mathbf{R}^{n\times n}$, $B \in \mathbf{R}^{n\times n}$, $b \in \mathbf{R}^n$, $c \in \mathbf{R}^n$, and matrices A and B are nonsingular

- flop count of brute-force method?

  solution. $(2/3)(2n)^3 = (16/3)n^3$

- how can we exploit structure?

  solution.

  - partition $x = (x_1, x_2)$
  - $x_1 = A^{-1}b$, $x_2 = B^{-1}c$
  - flop count: $2(2/3)n^3 = (4/3)n^3$

example: Solve the set of equations

$$\begin{bmatrix} I & B \\ C & I \end{bmatrix} x = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

where $B \in \mathbf{R}^{m\times n}$, $C \in \mathbf{R}^{n\times m}$, and $m \gg n$; also assume that the whole matrix is nonsingular

- flop count of brute-force method? **solution.** $(2/3)(m+n)^3$
- how can we exploit structure?

  solution.

  – use block elimination to get equations

  $$(I - CB)x_2 = b_2 - Cb_1 \quad \text{and} \quad x_1 = b_1 - Bx_2$$

  – flop count: forming $I - CB$ costs $2mn^2$, $b_2 - Cb_1$ is $2mn$, solving for x_2 is $(2/3)n^3$, and computing x_1 costs $2mn$; overall complexity is $2mn^2$

Solving almost separable linear equations

Consider the following system of $2n + m$ equations

$$\begin{aligned} Ax + By &= c \\ Dx + Ey + Fz &= g \\ Hy + Jz &= k \end{aligned}$$

where $A, J \in \mathbf{R}^{n\times n}$, $B, H \in \mathbf{R}^{n\times m}$, $D, F \in \mathbf{R}^{m\times n}$, $E \in \mathbf{R}^{m\times m}$, $c, k \in \mathbf{R}^{n}$, $g \in \mathbf{R}^{m}$ and $n > m$

- need to solve the following system

$$\begin{bmatrix} A & B & 0 \\ D & E & F \\ 0 & H & J \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} c \\ g \\ k \end{bmatrix}$$

- naive way: treat as dense

- can take advantage of the structure by first reordering the equations and variables

$$\begin{bmatrix} A & 0 & B \\ 0 & J & H \\ D & F & E \end{bmatrix} \begin{bmatrix} x \\ z \\ y \end{bmatrix} = \begin{bmatrix} c \\ k \\ g \end{bmatrix}$$

  the system now looks like an “arrow” system, which we can efficiently solve by block elimination.

- since

$$\begin{bmatrix} A & 0 \\ 0 & J \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix} + \begin{bmatrix} B \\ H \end{bmatrix} y = \begin{bmatrix} c \\ k \end{bmatrix}$$

then

$$\begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} A^{-1}c \\ J^{-1}k \end{bmatrix} - \begin{bmatrix} A^{-1}B \\ J^{-1}H \end{bmatrix} y$$

- we know that

$$\begin{bmatrix} D & F \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix} + Ey = g$$

- then plugging into the expression derived above

$$\begin{bmatrix} D & F \end{bmatrix} \left(\begin{bmatrix} A^{-1}c \\ J^{-1}k \end{bmatrix} - \begin{bmatrix} A^{-1}B \\ J^{-1}H \end{bmatrix} y \right) + Ey = g$$

- therefore

$$(E - DA^{-1}B - FJ^{-1}H)y = g - DA^{-1}c - FJ^{-1}k$$

We can therefore solve the system of equations efficiently by taking advantage of structure in the following way

- form

$$M = A^{-1}B, \qquad n = A^{-1}c,$$
$$P = J^{-1}H, \qquad q = J^{-1}k.$$

- compute $r = g - Dn - Fq$.

- compute $S = E - DM - FP$.

- find

$$y = S^{-1}r, \qquad x = n - My, \qquad z = q - Py.$$

Using sparsity in Matlab

- construct using `sparse`, `spalloc`, `speye`, `spdiags`, `spones`
- visualize and analyze using `spy`, `nnz`
- also have `sprand`, `sprandn`, `eigs`, `svds`
- be careful not to accidentally make sparse matrices dense

using sparsity in additional problem 2

- the (sparse) tridiagonal matrix $\Delta \in \mathbf{R}^{n \times n}$

$$\Delta = \begin{bmatrix} 1 & -1 & 0 & \cdots & 0 & 0 & 0 \\ -1 & 2 & -1 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 2 & -1 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 2 & -1 \\ 0 & 0 & 0 & \cdots & 0 & -1 & 1 \end{bmatrix}.$$

  can be built in Matlab as follows:

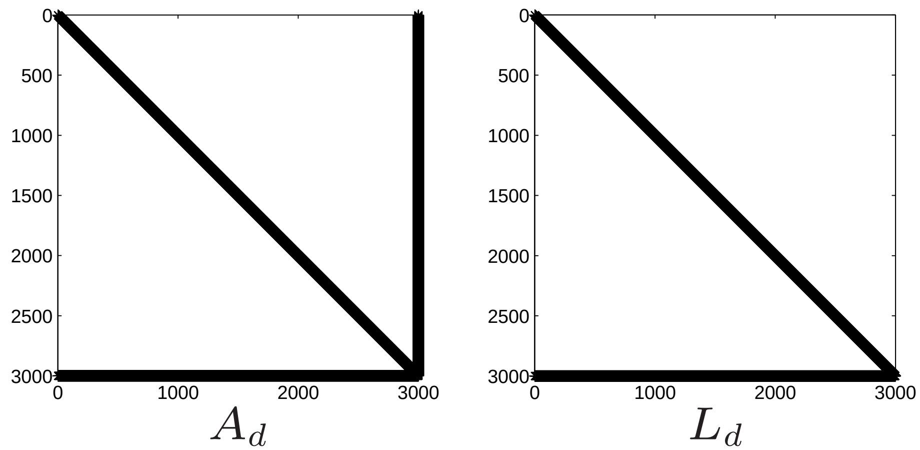
```
e = ones(n,1);
D = spdiags([-e 2*e -e],[-1 0 1], n,n);
D(1,1) = 1; D(n,n) = 1;
```

- the sparse identity matrix can be built using `speye`

Sparse Cholesky factorization with permutations

consider factorizing downward arrow matrix $A_d = L_d L_d^T$, with $n = 3000$

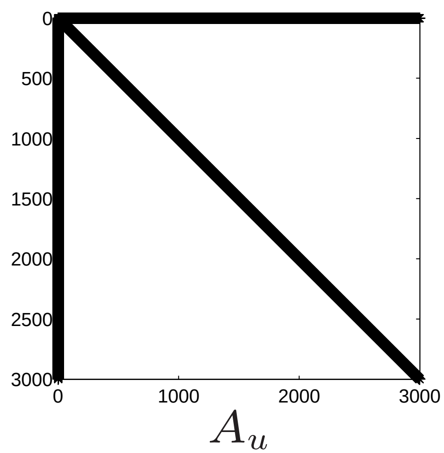
- `nnz(A_d)=8998`
- call using `L_d=chol(A_d,'lower')`
  (use `L_d=chol(A_d)'` in older versions of Matlab)

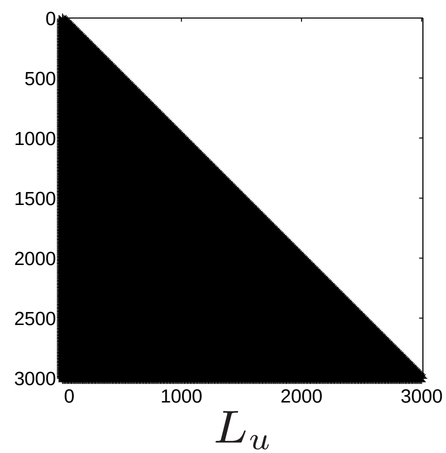


- `nnz(L_d)=5999`; factorization takes `tf_d=0.0022` seconds
- to solve $A_d x = b$, call `x=L_d'\(L_d\b)`, which takes `ts_d=0.0020` seconds to run

now look at factorizing upward arrow matrix $A_u = L_u L_u^T$

- again, `nnz(A_u)=8998`
- call using `L_u=chol(A_u,'lower')`

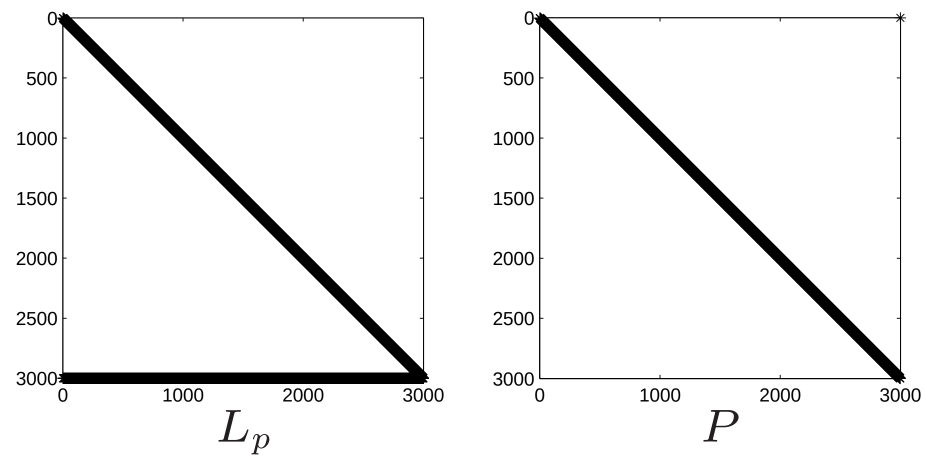




- `nnz(L_u)=4501500`, and takes `tf_u=3.7288` seconds to compute
- calling `x=L_u'\(L_u\b)` no longer efficient; takes `ts_u=0.5673` seconds to run

instead factorize A_u with permutations, $A_u = PL_pL_p^TP^T$

- call using `[L_p,pp,P]=chol(A_u,'lower')`



- P is Toeplitz matrix with 1's on the sub-diagonal and in upper right corner, $i.e.$, $P_{1,n} = 1$, $P_{k+1,k} = 1$ for $k = 1, \ldots, n$, all other entries 0
- factorization only takes `tf_p=0.0028` seconds to compute
- solve $A_u x = b$ using `x=P'\(L_p'\(L_p\(P\b)));` solve takes `ts_p=0.0042` seconds