Lecture 9. Introduction to Numerical Techniques

Ivan Papusha

CDS270–2: Mathematical Methods in Control and System Engineering

May 27, 2015

Logistics

- hw8 (last one) due today.
  - do an easy problem or CYOA
- hw7 solutions posted online
- reading: Davis Ch1–3
- examples today mostly from Tim Davis's book
- Trefethen and Bau, *Numerical Linear Algebra* is also a good resource.

Convex optimization methods

We wish to solve the unconstrained minimization problem

$$\text{minimize} \quad f(x).$$

- gradient descent: start with a guess x_0 for the optimum and update

$$x_{k+1} := x_k - t_k \nabla f(x_k),$$

  where t_k is a small step size.
- generally require step sizes to satisfy

$$\sum_k t_k^2 < \infty, \quad \sum_k t_k = \infty.$$

- convexity of f means algorithm converges to global minimum
- if f is not differentiable, replace $\nabla f(x_k)$ by a *subgradient* $g \in \mathbf{R}^n$, which satisfies

$$f(y) \geq f(x_k) + g^T(y - x_k) \quad \text{for all } y \in \mathbf{R}^n.$$

Interior point methods

The main idea is to solve for $x^\star$ in the optimality condition

$$\nabla f(x^\star) = 0.$$

workhorse technique. in practice, converges in ~ 10 steps.

---

algorithm: Newton's method

given: a starting point $x \in \mathbf{dom}\, f$, tolerance $\epsilon > 0$.
repeat:

1. Compute the Newton step and decrement
   $\Delta x_{\mathrm{nt}} := -\nabla^2 f(x)^{-1} \nabla f(x), \quad \lambda^2 := \nabla f(x)^T \nabla^2 f(x)^{-1} \nabla f(x)$
2. Stopping criterion. **quit** if $\lambda^2/2 \leq \epsilon$.
3. Line search. choose step size t by backtracking line search
4. Update. $x := x + t\Delta x_{\mathrm{nt}}$

Adding constraints: barrier method

To solve the constrained optimization problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \end{array}$$

we use a barrier function $\phi : \mathbf{R}^n \to \mathbf{R}$ that goes to $+\infty$ as x approaches the boundary of the feasible set.

logarithmic barrier.

$$\phi(x) = -\sum_{i=1}^{m} \log(-f_i(x))$$

central path. solve the related unconstrained problem

$$\text{minimize} \quad tf_0(x) + \phi(x).$$

As $t \to \infty$, the optimizing solution $x^\star(t) \to x^\star$.

Barriers for common cones

Barriers ϕ for conic constraints can be defined in terms of the generalized logarithm ψ,

$$\phi(x) = -\sum_{i=1}^{m} \psi_i(-f_i(x)),$$

- Nonnegative orthant $\mathbf{R}^n_+$:

$$\psi(x) = \sum_{i=1}^{n} \log x_i, \quad \nabla\psi(x) = (1/x_1, \ldots, 1/x_n)$$

- Second-order cone $\mathcal{Q}^n = \{(x_0, x_1) \mid \|x_1\|_2 \leq x_0\}$:

$$\psi(x) = \log\left(x_0^2 - \sum_{i=1}^{n-1}(x_1)_i^2\right)$$

- Positive semidefinite cone $\mathbf{S}^n_+$:

$$\psi(X) = \log\det(X), \quad \nabla\psi(X) = X^{-1}$$

On writing linear algebra software

~~don't~~ **try not to**

- `BLAS`: Basic Linear Algebra Subprograms
  - Level 1. norms, dot products, scalar-vector products $O(n)$
  - Level 2. matrix-vector products, rank-one updates $O(n^2)$
  - Level 3. matrix-matrix sums and products $O(n^3)$
- `LAPACK`: Linear Algebra PACKage
  - QR factorization, SVD, eigenvalue problems
- `ATLAS`: Automatically Tuned Linear Algebra Software
- `SuiteSparse`: direct methods for sparse systems
  - sparse matrix factorization, Cholesky decomposition, fill-reducing orderings, GPU methods
  - much of the suite is part of core Matlab

reference implementations often in fortran, many re-implementations exist

Sparse data structure

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix}$$

- highlighted zeros are *structural zeros*
- n_{nz} = number of nonzeros in the matrix
- three ways to store the matrix
  1. 2-dimensional array: $O(mn)$ space, independent of sparsity
  2. triplet format: list of triples (i, j, a_{ij}), $O(3n_{\text{nz}})$ space
  3. compressed column format: $O(2n_{\text{nz}} + n)$ space

Triplet format

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix}$$

(zero-based indexing in C code)

```
int i []    = { 2,   1,   3,   0,   1,   3,   3,   1,   0,   2   };
int j []    = { 2,   0,   3,   2,   1,   0,   1,   3,   0,   1   };
double x [] = { 3.0, 3.1, 1.0, 3.2, 2.9, 3.5, 0.4, 0.9, 4.5, 1.7 };
```

- multiple entries interpretation: if (i, j, α) and (i, j, β) both appear in the data structure, then the value of a_{ij} is the sum $\alpha + \beta$.
- conceptually useful when first forming a matrix
- not particularly efficient structure for linear algebra operations

Compressed column format

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix}$$

(zero-based indexing in C code)

```
int p []    = { 0,              3,              6,        8,        10 };
int i []    = { 0,   1,   3,   1,   2,   3,   0,   2,   1,   3   };
double x [] = { 4.5, 3.1, 3.5, 2.9, 1.7, 0.4, 3.2, 3.0, 0.9, 1.0 };
```

- `p` is an array of column pointers
- row indices of column `j` are stored in `i[p[j]]` through `i[p[j+1]-1]`
- first entry `p[0]` is always zero, last entry `p[n]` is the number of nonzero entries in the matrix
- access to a column is simple, access to a row (or tranposing) is difficult

Sparse data structure

```
/* matrix in compressed column or triplet form */
typedef struct cs_sparse {
  int nzmax; /* max number of entries */
  int m;     /* number of rows */
  int n;     /* number of columns */
  int *p;    /* col pointers (size n+1) or col indices (size nzmax) */
  int *i;    /* row indices, size nzmax */
  double *x; /* numerical values, size nzmax */
  int nz;    /* number of entries in triplet matrix */
             /* or -1 for compressed column */
} cs;
```

- handles both compressed column and triplet format
- p holds either column pointers (compressed column form) or column indices (triplet form)
- memory management (reallocating *x and resetting nzmax) is up to the algorithm
  - some algorithms need a temporary workspace
  - control systems running in-loop may need real time guarantees

Printing a compressed column matrix by columns

```
int cs_print (const cs *A, ...) {
  int p, j, m, n, nzmax, nz, *Ap, *Ai;
  double *Ax;
  if (!A) { printf ("(null)\n"); return (0); }
  n = A->m; n = A->n; Ap = A->p; Ai = A->i; Ax = A->x;
  nzmax = A->nzmax; nz = A->nz;
  ...
  if (nz < 0) { /* if matrix is compressed-column */
    for (j = 0; j < n; j++) {
      printf("col %d : locations %d to %d\n", j, Ap[j], Ap[j+1]-1);
      for (p = Ap[j]; p < Ap[j+1]; p++) {
        printf("     %d : %g\n", Ai[p], Ax ? Ax[p] : 1);
      }
    }
  }
  ... /* handle triplet case here */
  return (1);
}
```

Matrix–vector multiplication

The compressed column format allows for efficient computation of matrix–vector updates, *e.g.*,

$$y := Ax + y$$

guiding principle. exploit structural nonzeros to avoid needless work

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} := \begin{bmatrix} | & | & & | \\ A_{*1} & A_{*2} & \cdots & A_{*n} \\ | & | & & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

---

algorithm: gaxpy "general A times x plus y"

for $j = 0$ to $n - 1$ **do**
  for each i for which $a_{ij} \neq 0$ **do**
    $y_i := y_i + a_{ij} x_j$

Matrix–vector multiplication

---

algorithm: gaxpy "general A times x plus y"

for $j = 0$ to $n - 1$ **do**
  for each i for which $a_{ij} \neq 0$ **do**
    $y_i := y_i + a_{ij} x_j$

```
int cs_gaxpy (const cs *A, const double *x, double *y) {
  int p, i, j, n, *Ap, *Ai;
  double *Ax;
  if (!CS_CSC(A) || !x || !y) return (0);  /* check inputs */
  n = A->n; Ap = A->p; Ai = A->i; Ax = A->x;
  for (j = 0; j < n; j++) {                /* for each column */
    for (p = Ap[j]; p < Ap[j+1]; p++) {    /* for nonzero row */
      y[Ai[p]] += Ax[p] * x[j];
    }
  }
  return (1); /* success */
}
```

Solving triangular systems

We wish to solve $Lx = b$ for $x \in \mathbf{R}^n$, where L is a lower triangular $n \times n$ matrix. Partition L and x into blocks,

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix},$$

which leads to two equations

$$\begin{aligned} l_{11}x_1 &= b_1 \\ l_{21}x_1 + L_{22}x_2 &= b_2. \end{aligned}$$

recursive substructure:

1. the first block (scalar) is $x_1 = (b_1/l_{11})$
2. we can solve for $x_2 \in \mathbf{R}^{n-1}$ by solving the $(n-1) \times (n-1)$ triangular system
$$L_{22}x_2 = b_2 - l_{21}x_1.$$

Solving triangular systems

We can unwind the tail recursion to obtain the triangular solve algorithm

---

algorithm: `lsolve` "lower triangular solve"

$x := b$
for $j = 0$ to $n - 1$ **do**
  $x_j := x_j / l_{jj}$
  for each $i > j$ for which $l_{ij} \neq 0$ **do**
    $x_i := x_i - l_{ij} x_j$

- since b_1 and b_2 are only used once, x can overwrite b in the implementation
- saves the need to create a temporary workspace

Solving triangular systems

```
int cs_solve (const cs *L, double *x) {
  int p, j, n, *Lp, *Li;
  double *Lx;
  n = L->n; Lp = L->p; Li = L->i; Lx = L->x;
  if (!CS_CSC(L) || !x) return (0); /* check inputs */
  for (j = 0; j < n; j++) {
    x[j] /= Lx[Lp[j]];
    for (p = Lp[j]+1; p < Lp[j+1]; p++) {
      x[Li[p]] -= Lx[p] * x[j];
    }
  }
  return (1);
}
```

- accesses L columnwise
- assumes the diagonal entries of L are nonzero and present
- input `x` initially contains the righthandside b, and is overwritten with the solution.

Solving general systems

Now we wish to solve the system $Ax = b$ where A is a general matrix.

- lots of ways to do this
- generally all these involve some sort of factorization of A
  - A unstructured: LU, LDU, QR, SVD
  - A positive semidefinite: Cholesky (LL^T, LDL^T), diagonalization (generally half the effort of the unstructured algorithm)
  - permuted versions of these (more later)
- most of the work in interior point solvers is spent in this step

Example: LU decomposition

fact. Every matrix $A \in \mathbf{R}^{n \times n}$ can be written as

$$A = LU,$$

where L is lower triangular and U is upper triangular.

We can solve $Ax = b$ by splitting into three steps:

1. *factorization.* find L and U such that $A = LU$.
2. *forward solve.* solve the lower triangular system

$$Ly = b.$$

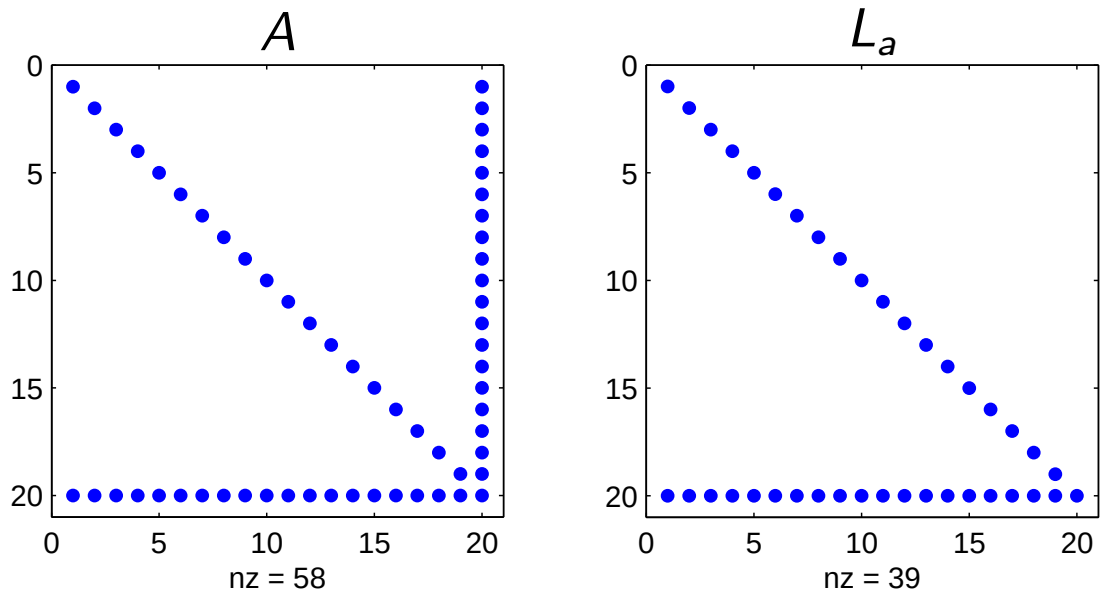
3. *backward solve.* solve the upper triangular system

$$Ux = y.$$

Sparse Cholesky factorization

Consider factoring downward arrow matrix $A = L_a L_a^T$

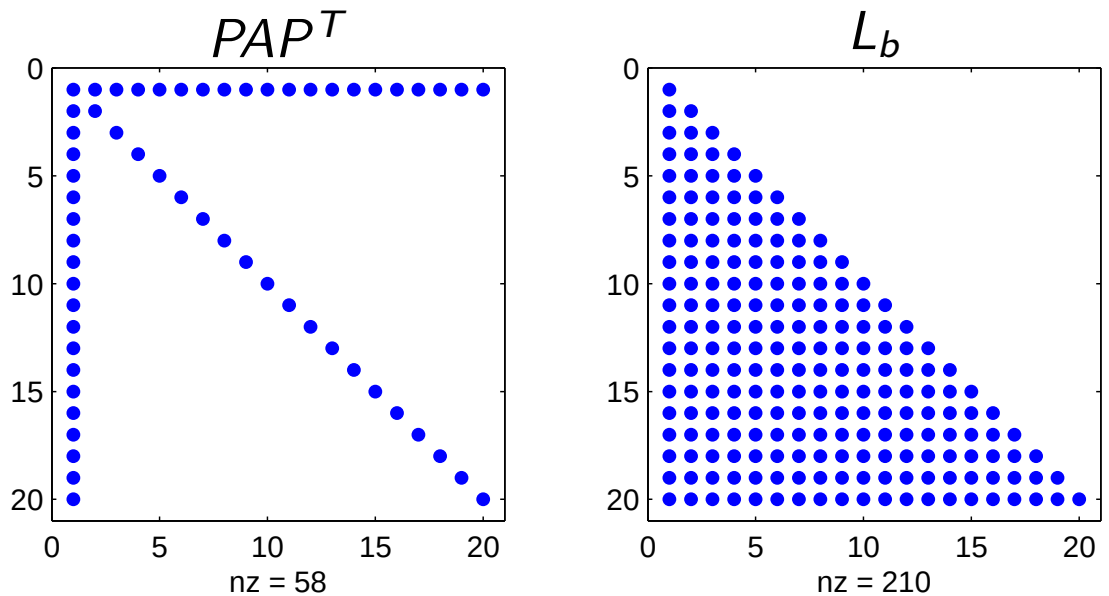
- pick $A = A^T \in \mathbf{R}^{20\times 20}$, `nnz(A)`$=58$
- call using `L_a=chol(A,'lower')`



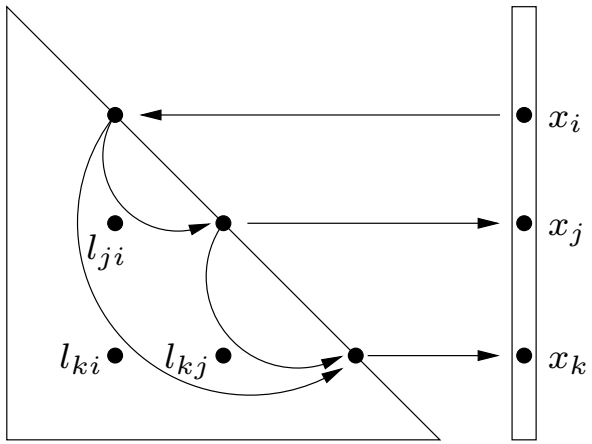
Sparse Cholesky factorization with permutation

now permute the entries of A to PAP^T and rewrite $L_b L_b^T = PAP^T$

- `nnz(P*A*P')`=58
- call using `L_b=chol(P*A*P','lower')`



Sparsity rule for Cholesky decomposition



Thm. 4.2: $a_{ik} \neq 0$ implies $l_{ki} \neq 0$

Thm. 4.3: $l_{ji} \neq 0$ and $l_{ki} \neq 0$ implies $l_{kj} \neq 0$

Thus l_{ki} redundant for $\mathcal{X} = \text{Reach}_L(i)$

Figure 4.1. *Pruning the directed graph G_L yields the elimination tree $\mathcal{T}$*

(From Davis, Ch 4)

Sparsity rule for Cholesky decomposition

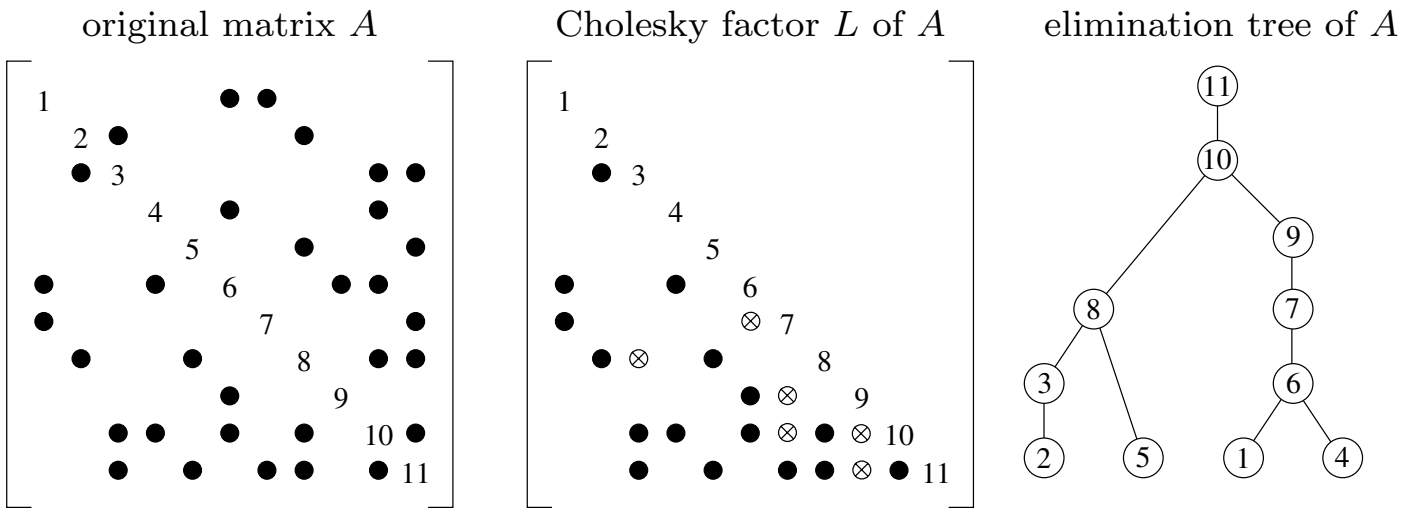


Figure 4.2. *Example matrix A, factor L, and elimination tree*

(From Davis, Ch 4)

Fill-reducing orderings

Well developed theory for sparse operations see *e.g.*, [Davis 2006]

- instead of solving $Ax = b$, solve new system

$$\underbrace{(PAP^T)}_{=LL^T}(Px) = Pb, \quad P = \text{permutation matrix}$$

- then do two triangular solves for y and $\tilde{x}$

$$Ly = Pb, \quad L^T\tilde{x} = y \quad \rightarrow \quad x = P^T\tilde{x}$$

- finding optimal fill-reducing ordering is NP-complete
- minimum degree heuristic: at each step of Gaussian elimination permute rows and columns to minimize the number of off-diagonal nonzeros in pivot row and column
- lots of heuristics (AMD, COLAMD, METIS)

Thanks!