G22.3033-004, Spring 2009

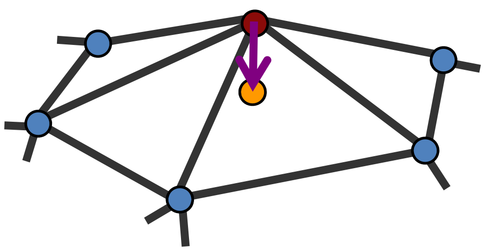
Interactive Shape Modeling

Solvers

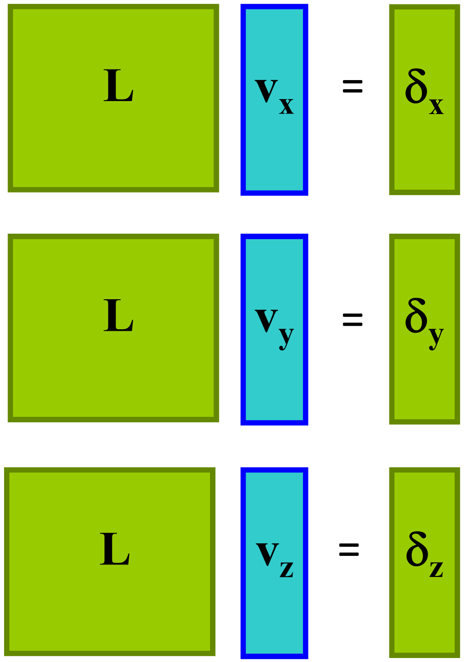
Linear Solvers

Motivation

- Laplace-type systems

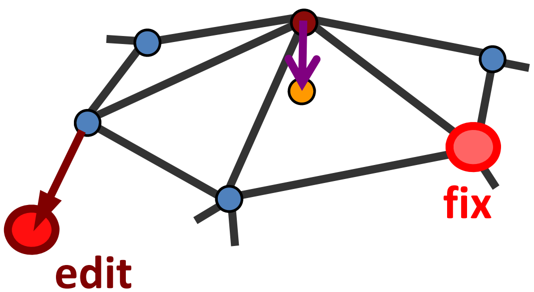


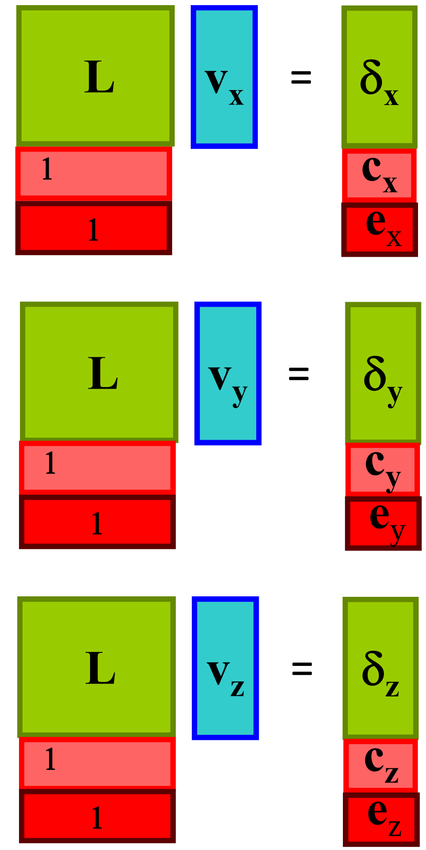
$$\boldsymbol{\delta}_i = \sum_{j \in N(i)} w_{ij} \left(\mathbf{v}_i - \mathbf{v}_j \right)$$



Linear Solvers

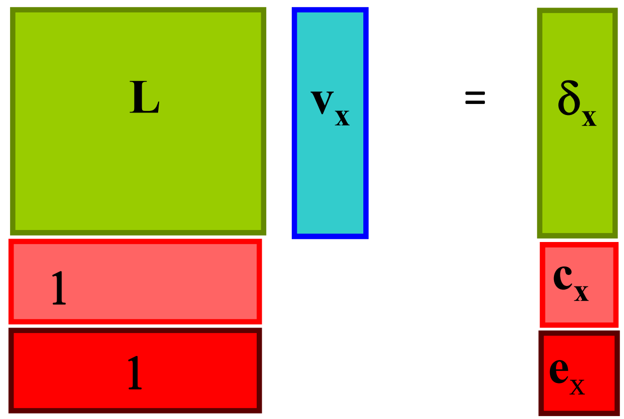
Motivation





Linear Solvers

Motivation

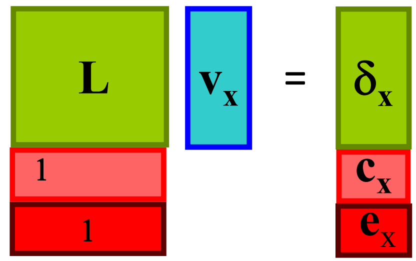


$$\tilde{\mathbf{x}} = \arg\min_{\mathbf{x}} \left(\left\| L\mathbf{x} - \boldsymbol{\delta}_x \right\|^2 + \sum_{s=1}^{k} \left| x_k - c_k \right|^2 \right)$$

… and the same for y and z

Linear Solvers

Motivation



$$\tilde{L}\,x = c$$

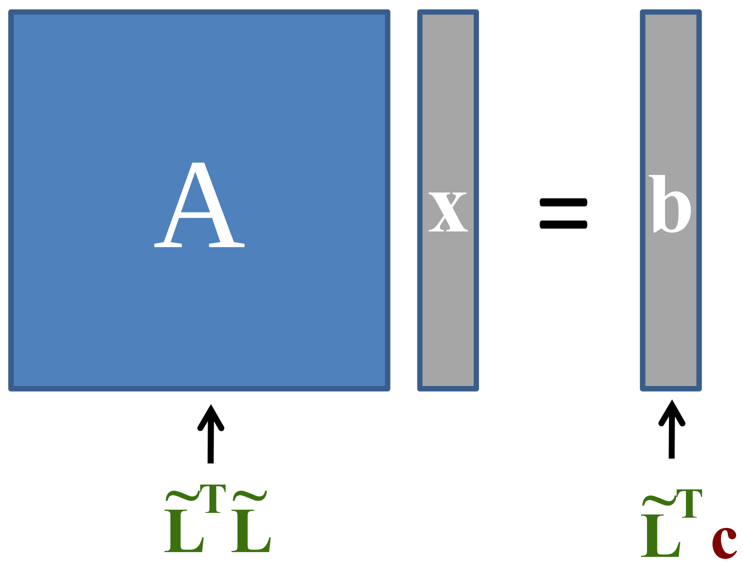
Normal Equations:

$$\tilde{L}^T \tilde{L}\,x = \tilde{L}^T c$$

$$x = (\tilde{L}^T \tilde{L})^{-1}\,\tilde{L}^T c$$

Linear Systems

- Matrix is often fixed, rhs changes



Iterative Solvers

- General approach: try to minimize some energy function $E(\mathbf{x})$
- Linear case: $E(\mathbf{x}) = \|\mathrm{A}\mathbf{x} - \mathbf{b}\|^2$
- Start from a guess $\mathbf{x}_0$
- Iteratively improve: $\mathbf{x}_{i+1} = g(\mathbf{x}_i)$
- Convergence: $E(\mathbf{x})$ sufficiently small

Descent Search

General algorithm

- Input: initial guess $\mathbf{x}_0 \in R^n$
- Step 0: set $i = 0$
- Step 1: if $E(\mathbf{x}) < \varepsilon$ stop,
  else compute ***search direction*** $\mathbf{h}_i \in \mathrm{R}^n$
- Step 2: compute the ***step size*** λ_i

  $$\lambda_i \in \arg\min_{\lambda \geq 0} E(\mathbf{x}_i + \lambda \cdot \mathbf{h}_i)$$ ← Line search
- Step 3: set $\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda_i \mathbf{h}_i$, goto Step 1

Descent Search

Quadratic energy (linear problem)

- Input: initial guess $\mathbf{x}_0 \in R^n$
- Step 0: set $i = 0$
- Step 1: if $\|A\mathbf{x} - \mathbf{b}\|^2 < \varepsilon$ stop,
  else compute ***search direction*** $\mathbf{h}_i \in \mathrm{R}^n$
- Step 2: compute the ***step size*** λ_i

$$\lambda_i \in \arg\min_{\lambda \geq 0} \|A(\mathbf{x}_i + \lambda \cdot \mathbf{h}_i) - \mathbf{b}\|$$

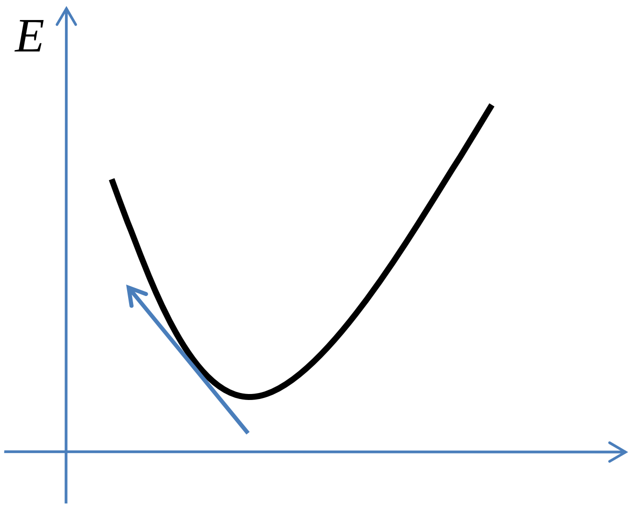
Line search

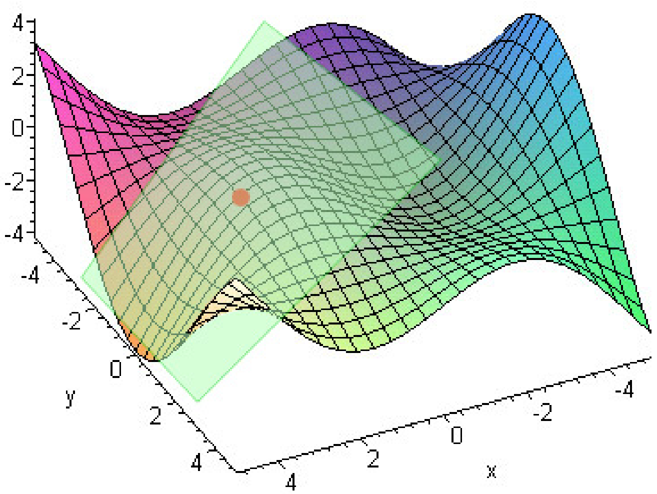
- Step 3: set $\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda_i \mathbf{h}_i$, goto Step 1

Search Direction $\mathbf{h}_i$

Steepest descent

- Gradient is the direction in which the function grows the fastest



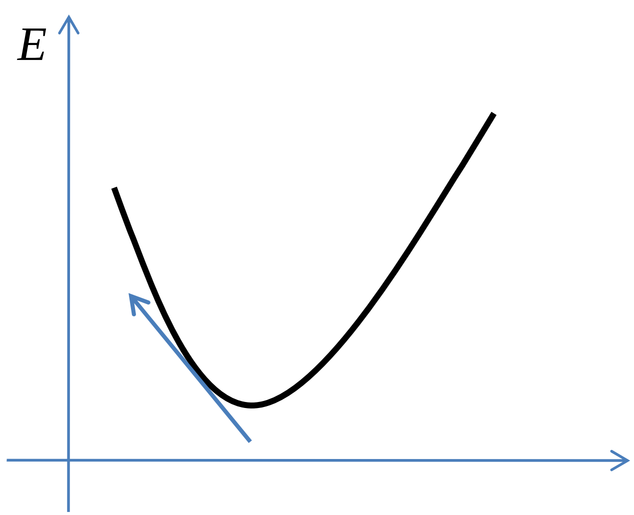


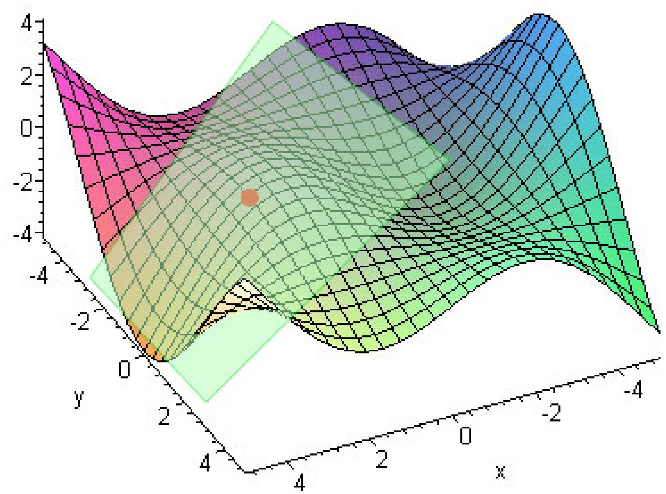
$$\mathbf{h}_i = -\nabla E(\mathbf{x}_i) \,/\, \|\nabla E(\mathbf{x}_i)\|$$

Search Direction $\mathbf{h}_i$

Steepest descent

- Gradient is the direction in which the function grows the fastest



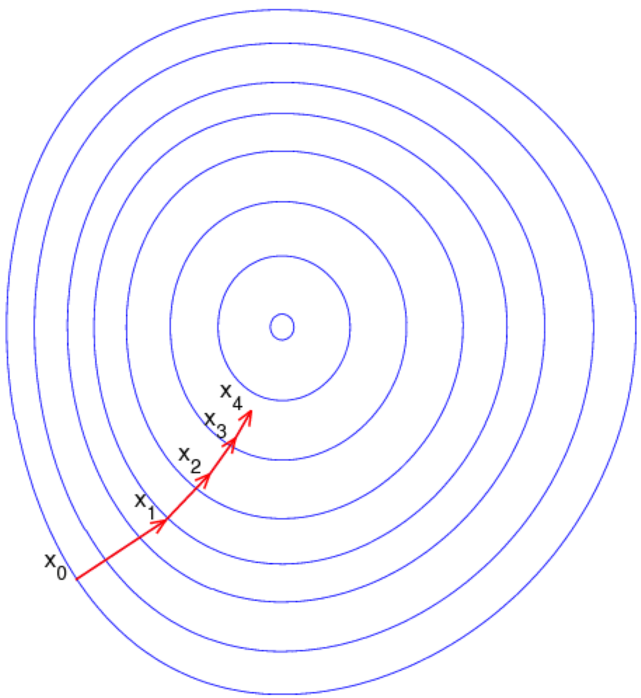


$$\nabla E(\mathbf{x}_i) = 2(A^T A\mathbf{x}_i - A^T \mathbf{b})$$

Search Direction $\mathbf{h}_i$

Steepest descent

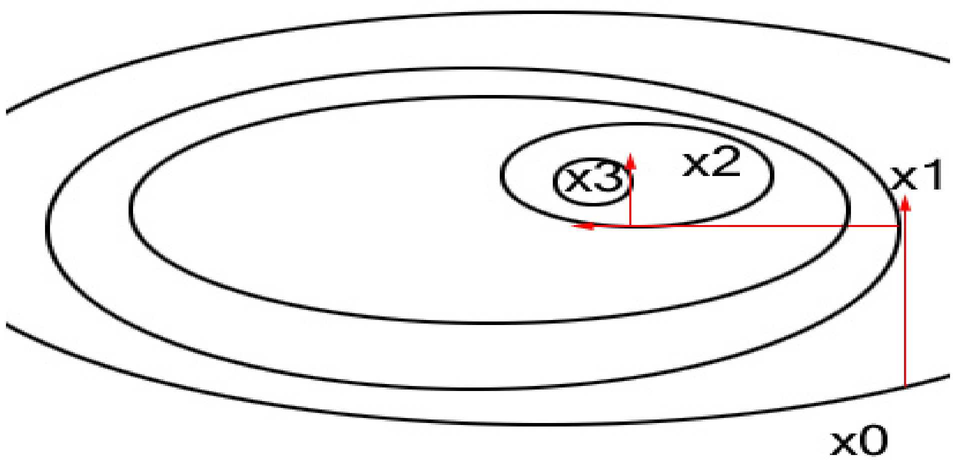
- Gradient is the direction in which the function grows the fastest



Search Direction $\mathbf{h}_i$

Steepest descent

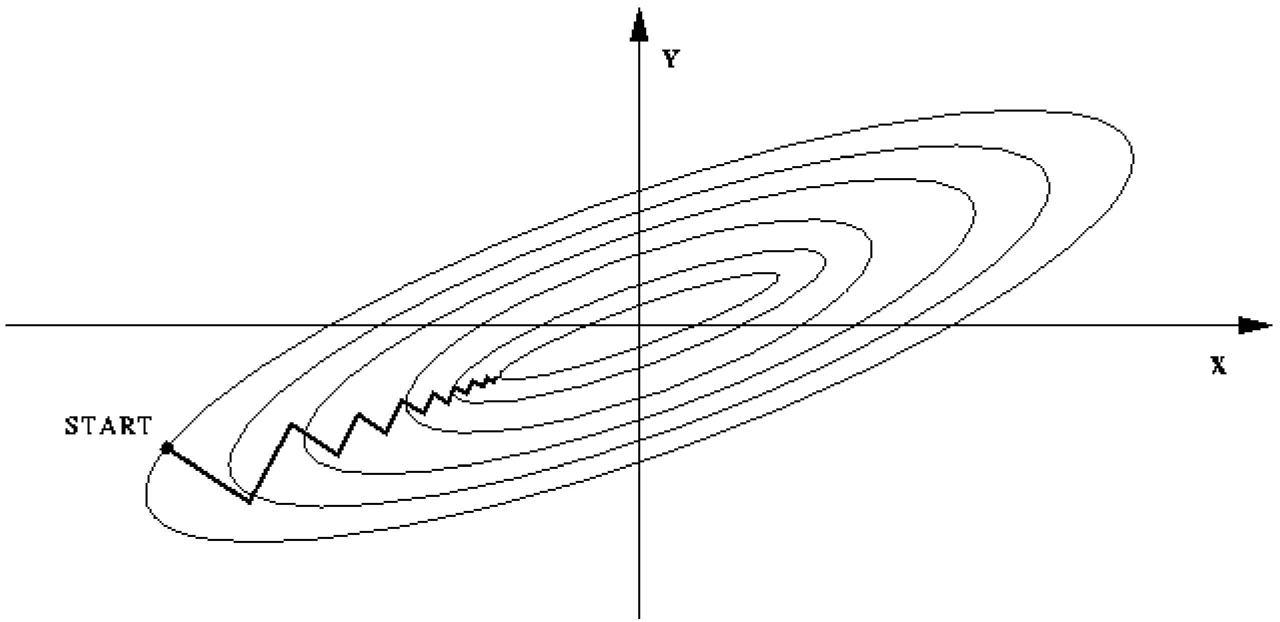
- Unlucky case: we pick the same direction many times



Search Direction $\mathbf{h}_i$

Steepest descent

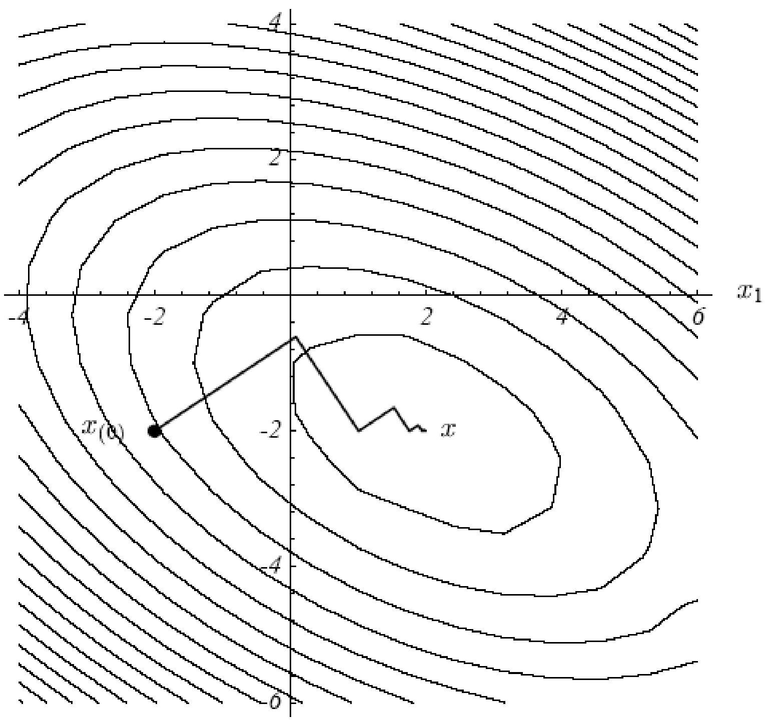
- Unlucky case: we pick the same direction many times

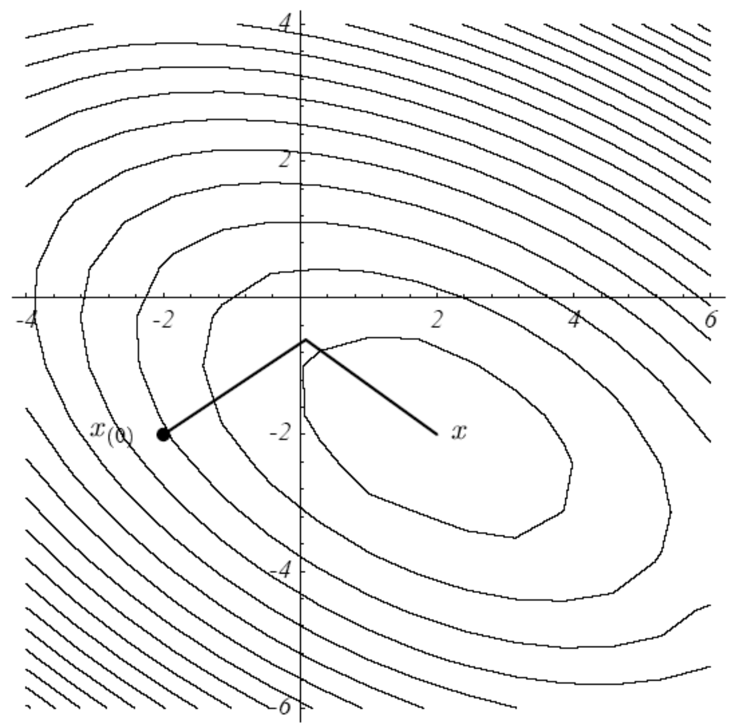


Search Direction $\mathbf{h}_i$

Conjugate gradient

- Choose n linearly independent directions
- $\Rightarrow$ Converge in n steps





Search Direction $\mathbf{h}_i$

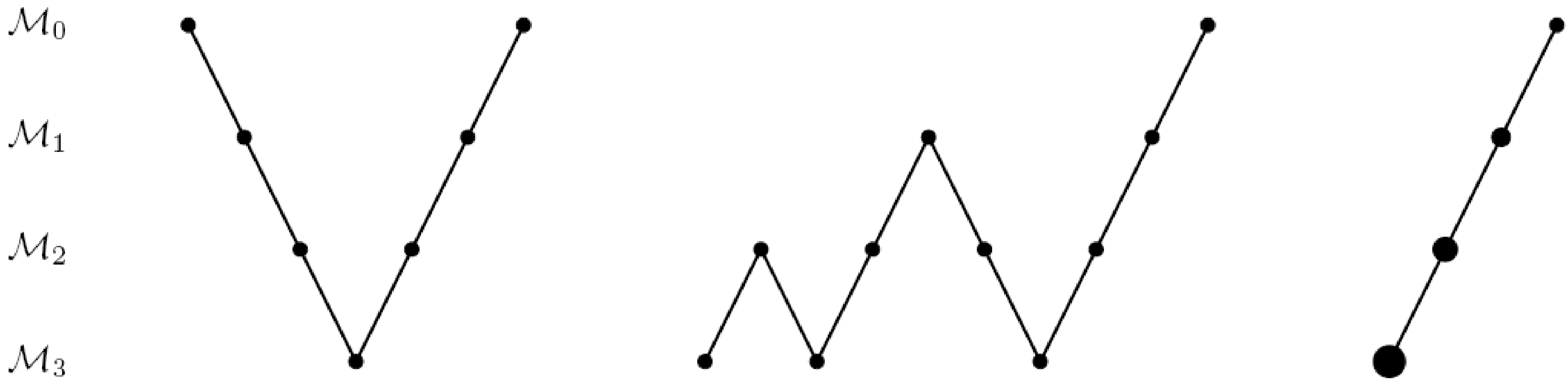
Conjugate gradient

- The directions $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_n$ are chosen to be mutually "conjugate", i.e., orthogonal w.r.t. the inner product defined by A

$$\left\langle \mathbf{A}\mathbf{h}_i, \mathbf{h}_j \right\rangle = \mathbf{h}_j^{\mathrm{T}} \mathbf{A}\mathbf{h}_i = 0$$

Multigrid Solvers

- Coarsen the matrix and the rhs
- Solve on the coarse level, then interpolate to the finer level
- On meshes: geometric multigrid, i.e. coarsen the mesh by edge collapse operations



Iterative Solvers

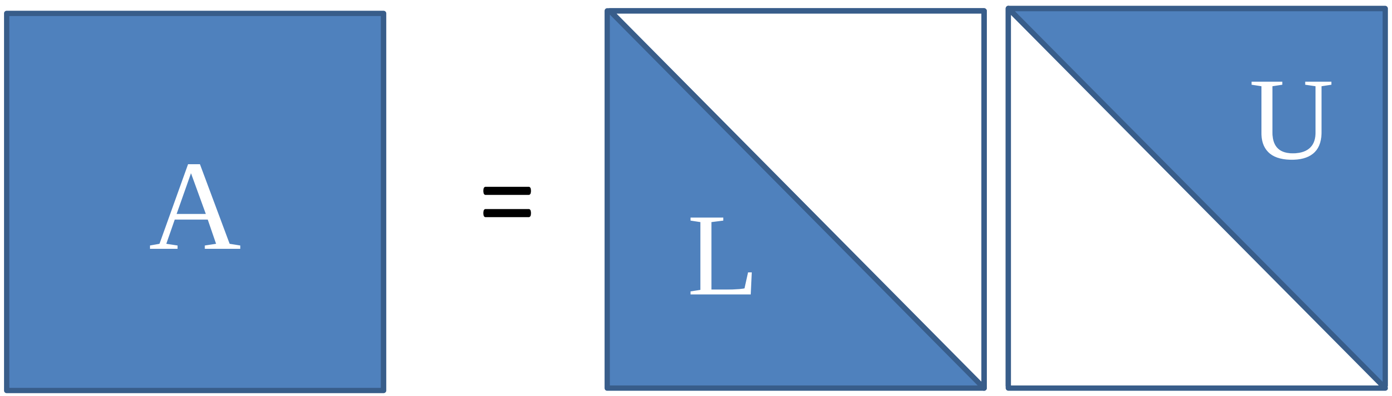
Discussion

- Efficient in memory
  - Only store the matrix A

- Not much gain when the rhs changes
  - Still need to iterate to find the solution, even though A is the same
- Too slow for interactive applications
- Problem-dependent parameters

Matrix Factorization

LU decomposition

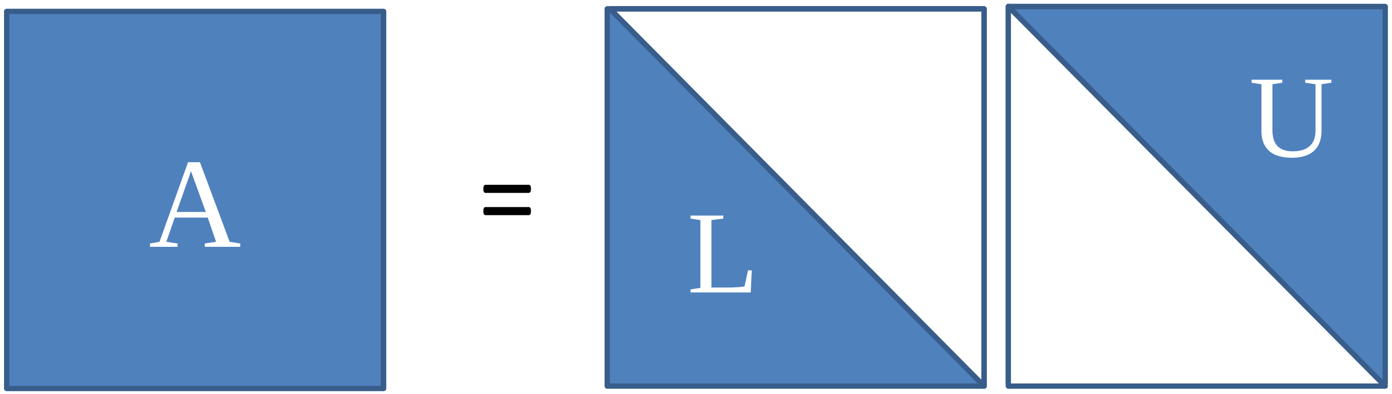


$$A\mathbf{x} = \mathbf{b}$$

$$LU\mathbf{x} = \mathbf{b}$$

Matrix Factorization

LU decomposition

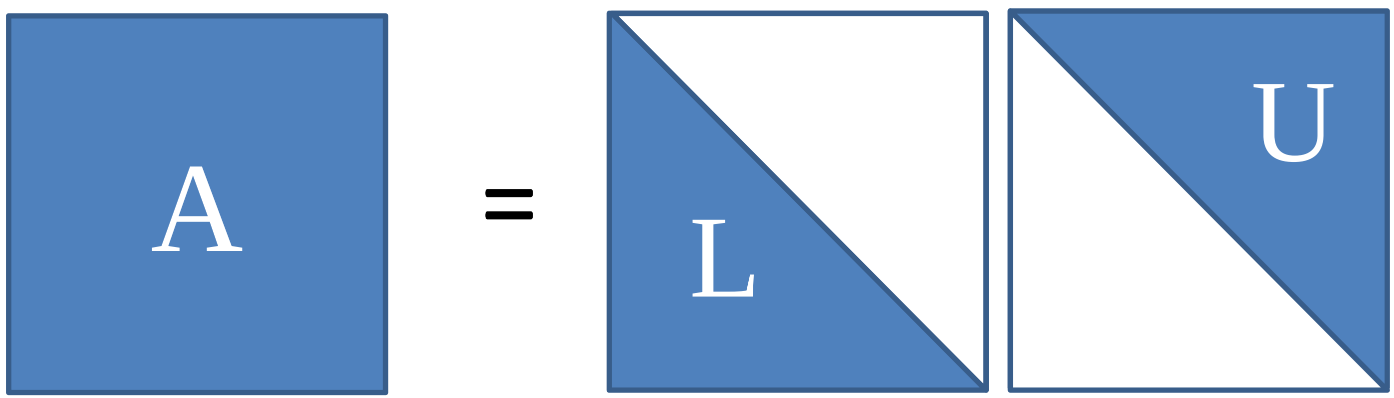


$$A\mathbf{x} = \mathbf{b}$$

$$L(U\mathbf{x}) = \mathbf{b}$$

Matrix Factorization

LU decomposition



$$\mathrm{A}\mathbf{x} = \mathbf{b}$$
$$\mathrm{L}(\mathrm{U}\mathbf{x}) = \mathbf{b}$$

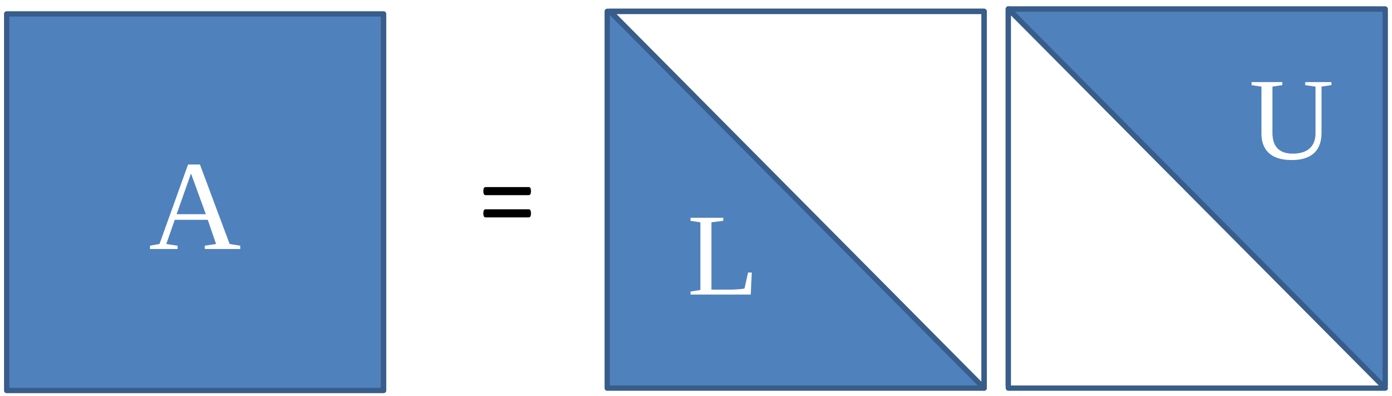
$$\Rightarrow$$

$$\left.\begin{aligned}\mathrm{L}\mathbf{y} &= \mathbf{b}\\ \mathrm{U}\mathbf{x} &= \mathbf{y}\end{aligned}\right\}$$

This is backsubstitution. If L, U are sparse it is very fast. The hard work is computing L and U

Matrix Factorization

LU decomposition



$$\mathrm{A}\mathbf{x} = \mathbf{b}$$
$$\mathrm{L}(\mathrm{U}\mathbf{x}) = \mathbf{b}$$

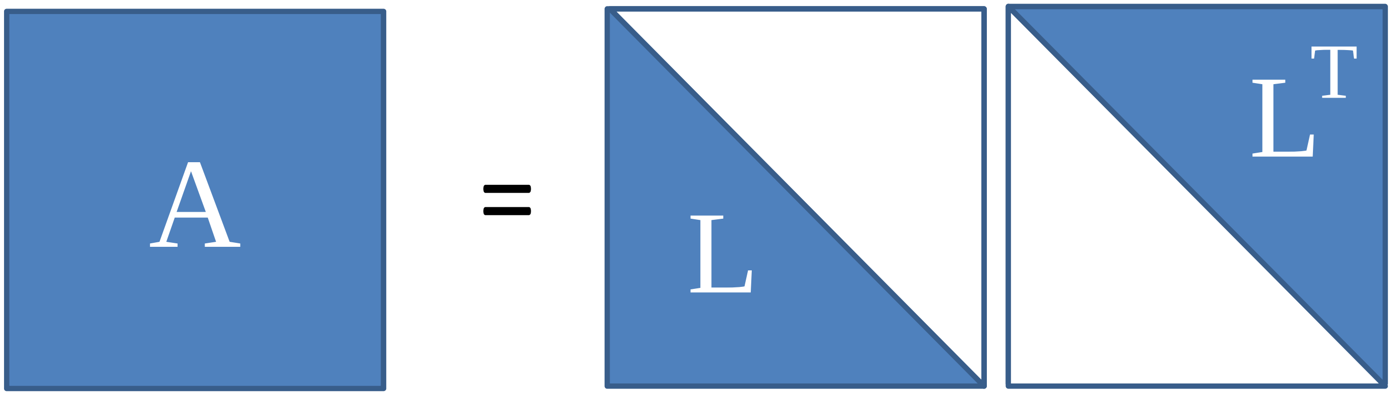
$\Rightarrow$

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

This is backsubstitution. If L, U are sparse it is very fast. The hard work is computing L and U

Matrix Factorization

Cholesky decomposition



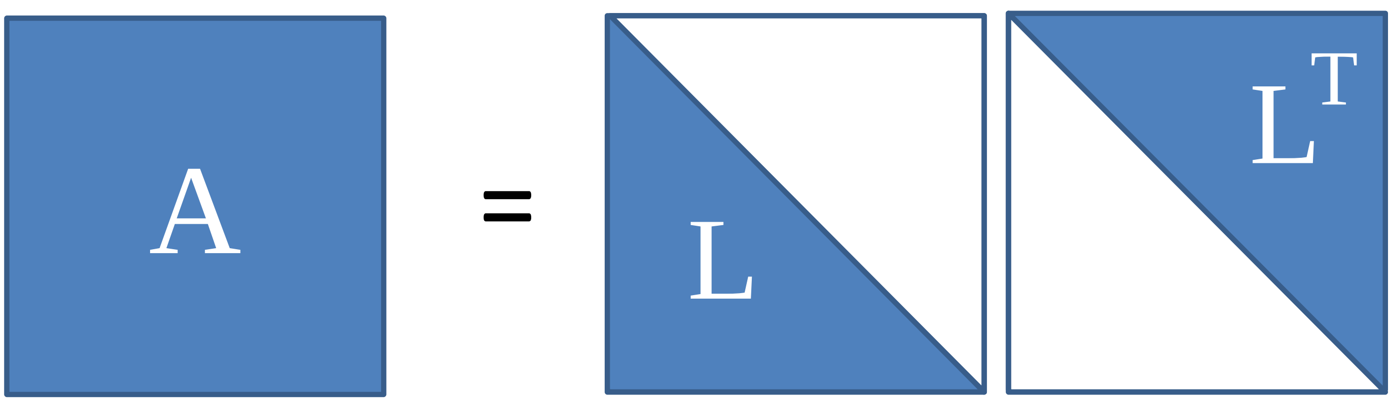
Cholesky factor exists if A is positive definite. It is even better than LU because we save memory.

Cholesky Decomposition

$$A = LL^T$$

- A is symmetric positive definite (PSD):

  $\forall\ \mathbf{x} \neq 0,\ \langle A\mathbf{x}, \mathbf{x} \rangle > 0 \quad \Leftrightarrow \quad$ all A's eigenvalues > 0

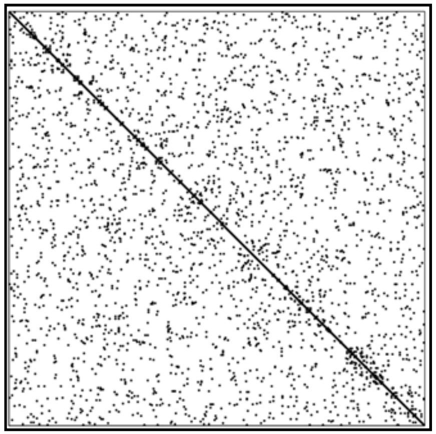


Dense Cholesky Factorization

$A = LL^T$

500×500 matrix

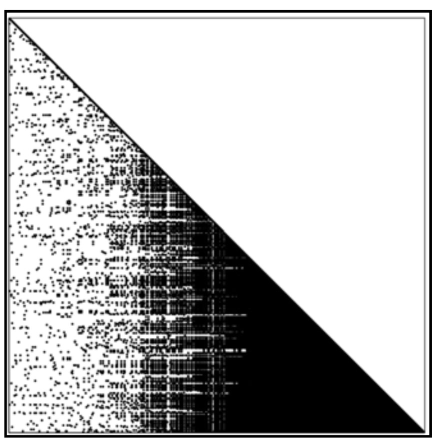
3500 nonzeros



Cholesky Factorization

L

36k nonzeros

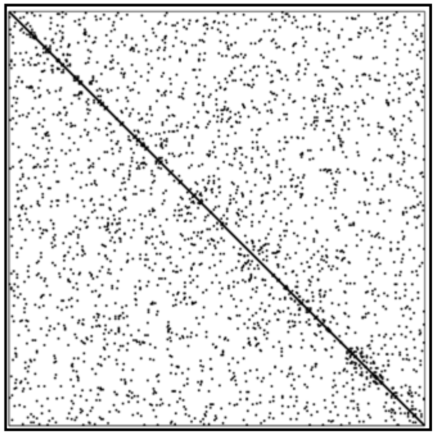


Sparse Cholesky Factorization

$A = LL^T$

500×500 matrix

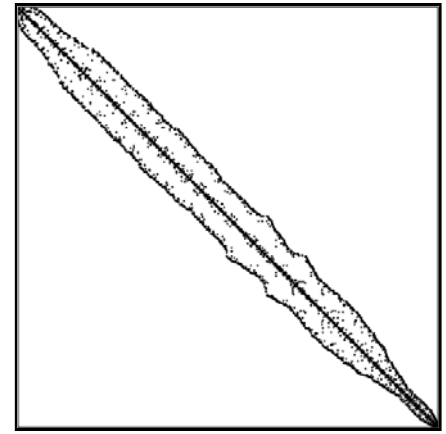
3500 nonzeros



Reordering

PAP^T

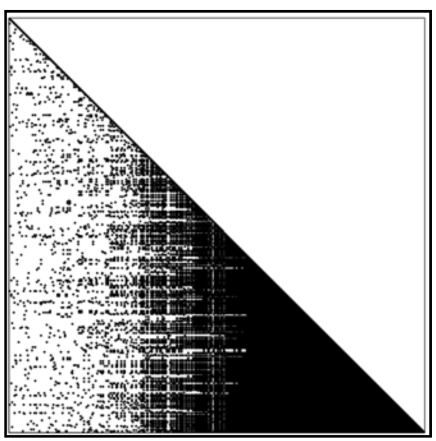
reverse Cuthill-McKee algorithm



Cholesky Factorization

L

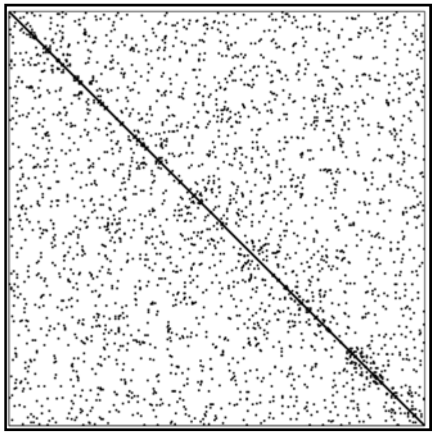
36k nonzeros



Sparse Cholesky Factorization

$A = LL^T$

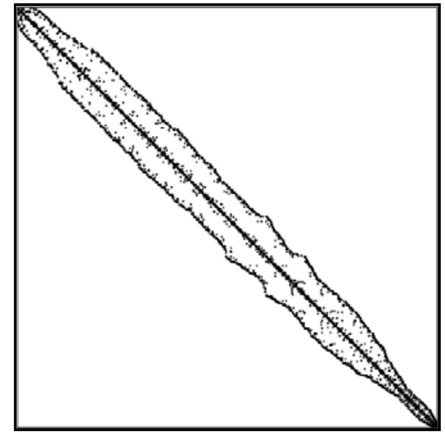
500×500 matrix
3500 nonzeros



Reordering

PAP^T

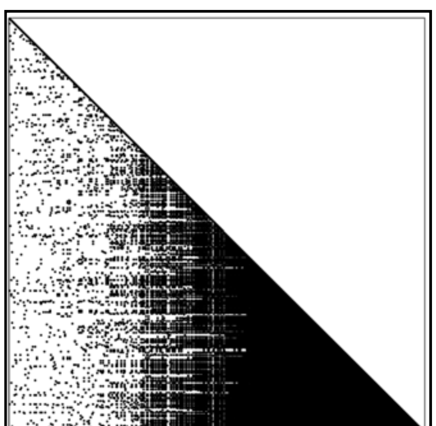
reverse Cuthill-McKee algorithm



Cholesky Factorization

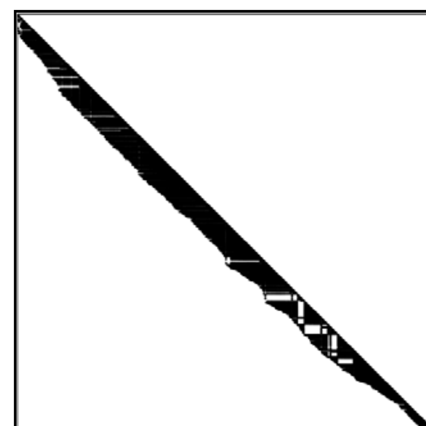
L

36k nonzeros



L

14k nonzeros

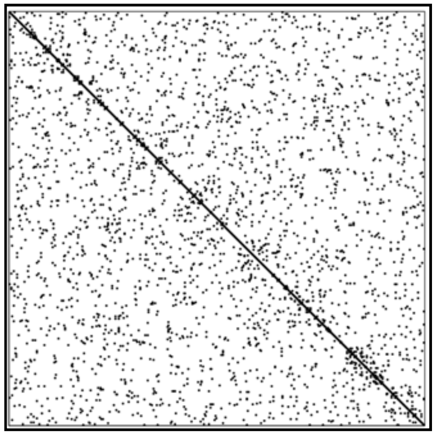


Sparse Cholesky Factorization

$A = LL^T$

500×500 matrix

3500 nonzeros

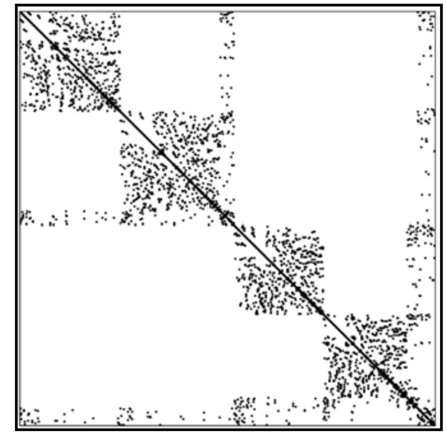


Reordering

PAP^T

nested dissection

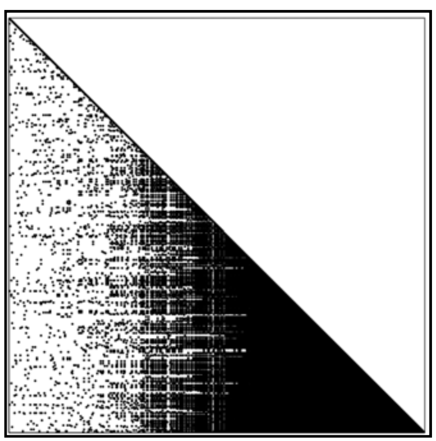
(parallelizable)



Cholesky Factorization

L

36k nonzeros

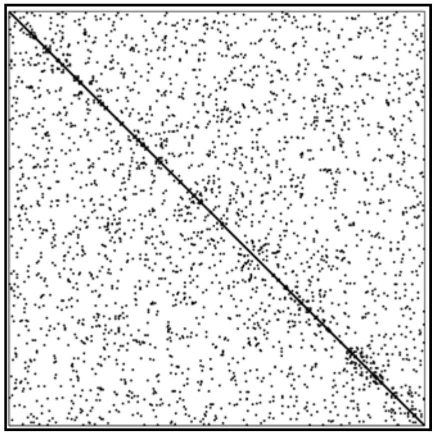


Sparse Cholesky Factorization

$A = LL^T$

500×500 matrix

3500 nonzeros

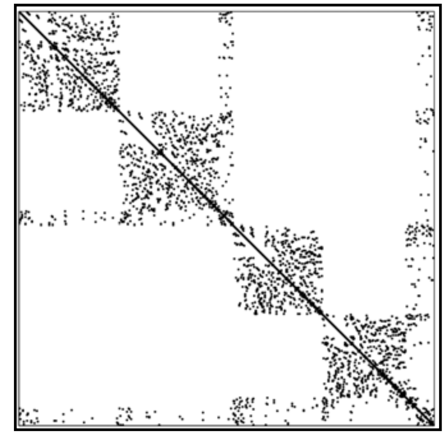


Reordering

PAP^T

nested dissection

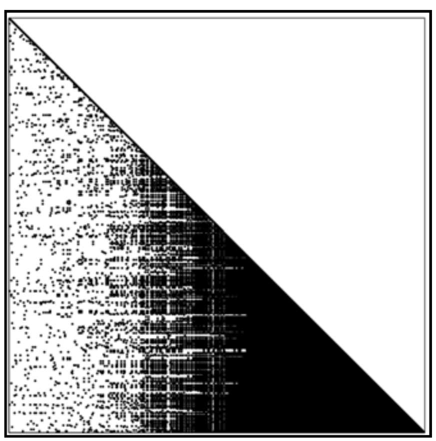
(parallelizable)



Cholesky Factorization

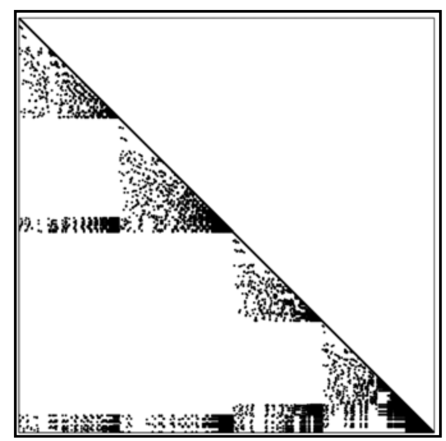
L

36k nonzeros



L

7k nonzeros



Direct Solvers

Discussion

- Highly accurate
  - Manipulate matrix structure
  - No iterations, everything is closed-form
- Easy to use
  - Off-the-shelf library, no parameters
- If A stays fixed, changing rhs ($\mathbf{b}$) is cheap
  - Just need to back-substitute (factor precomputed)

Direct Solvers

Discussion

- High memory cost
  - Need to store the factor, which is typically denser than the matrix A
- If the matrix A changes, need to re-compute the factor (expensive)

TAUCS tutorial

- TAUCS: a library of sparse linear solvers
  - Has both iterative and direct solvers
  - Direct (Cholesky and LU) use reordering and are very fast

- I provide a wrapper for TAUCS on the final project homepage

TAUCS tutorial

- Basic operations:
  - Define a sparse matrix structure
  - Fill the matrix with its nonzero values (i, j, v)
  - Factor $A^T A$
  - Provide an rhs and solve

TAUCS tutorial

- Basic operations:
  - Define a sparse matrix structure

```
InitTaucsInterface();

int idA;
idA = CreateMatrix(4, 3);
```

#rows #cols

TAUCS tutorial

- Basic operations:
  - Fill the matrix A with its nonzero values (i, j, v)

```
SetMatrixEntry(idA, i, j, v);
```

TAUCS tutorial

- Basic operations:
  - Fill the matrix A with its nonzero values (i, j, v)

```
SetMatrixEntry(idA, i, j, v);
```

matrix ID, obtained in CreateMatrix

TAUCS tutorial

- Basic operations:
  - Fill the matrix A with its nonzero values (i, j, v)

```
SetMatrixEntry(idA, i, j, v);
```

row index i, column index j,
zero-based

TAUCS tutorial

- Basic operations:
  - Fill the matrix A with its nonzero values (i, j, v)

```
SetMatrixEntry(idA, i, j, v);
```

value of matrix entry ij
for instance, $-w_{ij}$

TAUCS tutorial

- Basic operations:
  - Factor the matrix A^TA

```
FactorATA(idA);
```

TAUCS tutorial

- Basic operations:
  - Provide an rhs and solve

```
taucsType b[4] = {3, 4, 5, 6};
taucsType x[3];

SolveATA(idA, b, x, 1);
```

TAUCS tutorial

- Basic operations:
  - Provide an rhs and solve

```
taucsType b[4] = {3, 4, 5, 6};
taucsType x[3];

SolveATA(idA, b, x, 1);
```

typedef for `double`

TAUCS tutorial

- Basic operations:
  - Provide an rhs and solve

```
taucsType b[4] = {3, 4, 5, 6};
taucsType x[3];

SolveATA(idA, b, x, 1);
```

ID of the A matrix

TAUCS tutorial

- Basic operations:
  - Provide an rhs and solve

```
taucsType b[4] = {3, 4, 5, 6};
taucsType x[3];

SolveATA(idA, b, x, 1);
```

rhs for the LS system Ax = b

TAUCS tutorial

- Basic operations:
  - Provide an rhs and solve

```
taucsType b[4] = {3, 4, 5, 6};
taucsType x[3];

SolveATA(idA, b, x, 1);
```

array for the solution

TAUCS tutorial

- Basic operations:
  - Provide an rhs and solve

A is 4x3

```
taucsType b[4] = {3, 4, 5, 6};
taucsType x[3];

SolveATA(idA, b, x, 1);
```

number of rhs's

TAUCS tutorial

- Basic operations:
  - Provide an rhs and solve

A is 4x3

```
taucsType b2[8] = {3, 4, 5, 6, 7, 8, 9, 10};
taucsType xy[6];

SolveATA(idA, b2, xy, 2);
```

number of rhs's

TAUCS tutorial

- If the matrix A is square a priori, no need to solve the LS system
- Then just use `FactorA()`and `SolveA()`

Further Reading

- **Efficient Linear System Solvers for Mesh Processing**

  Mario Botsch, David Bommes, Leif Kobbelt

  Invited paper at IMA Mathematics of Surfaces XI, Lecture Notes in Computer Science, Vol 3604, 2005, pp. 62-83.