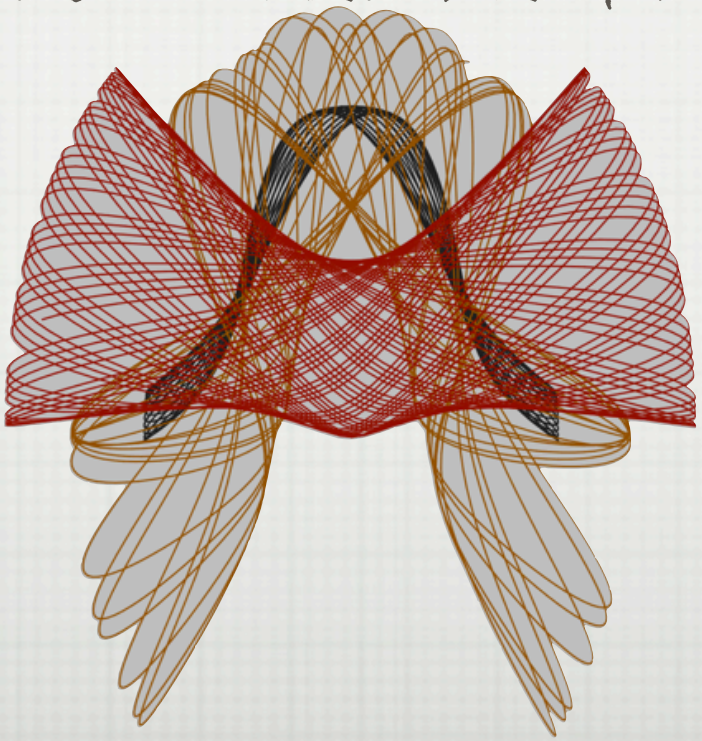
PHY-4810
COMPUTATIONAL PHYSICS

LECTURE 13: MATRIX ALGEBRA



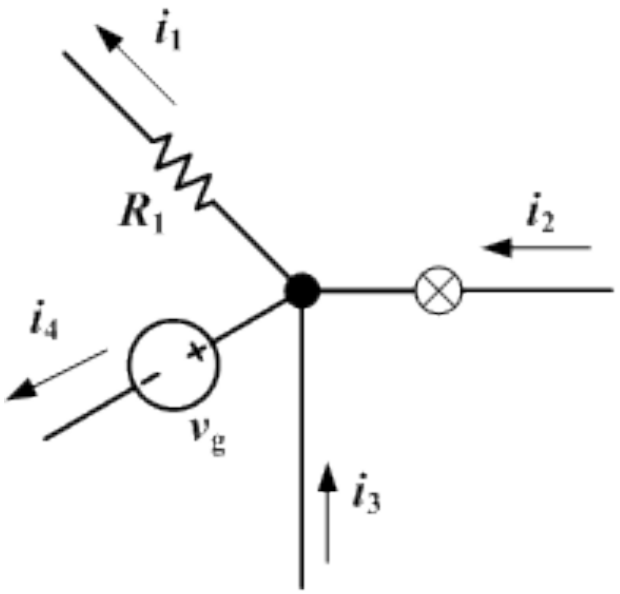
BEFORE WE START...

- **This is not the easiest material covered so far...**
  - **I would say it is maybe the hardest...**
- **Lots of materials...**
  - **(hopefully) some new...**
- **Focus on "using" rather than "deriving"...**
  - **Main ideas behind the derivation will be provided though...**
- **Bear with me...**

MOTIVATION: KIRCHHOFF'S LAWS

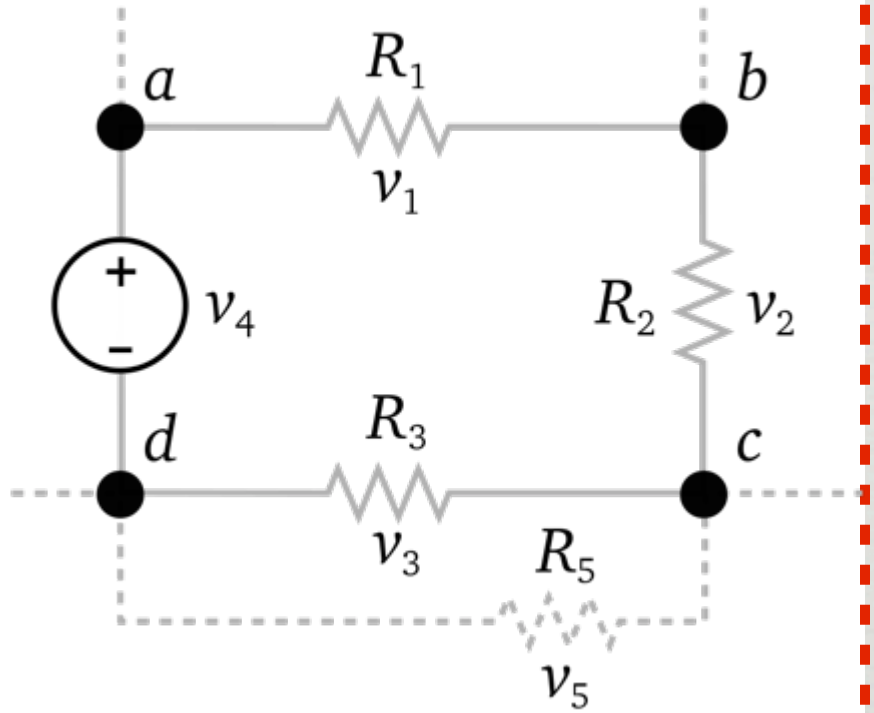
Gustav Robert Kirchhoff

CURRENT LAW



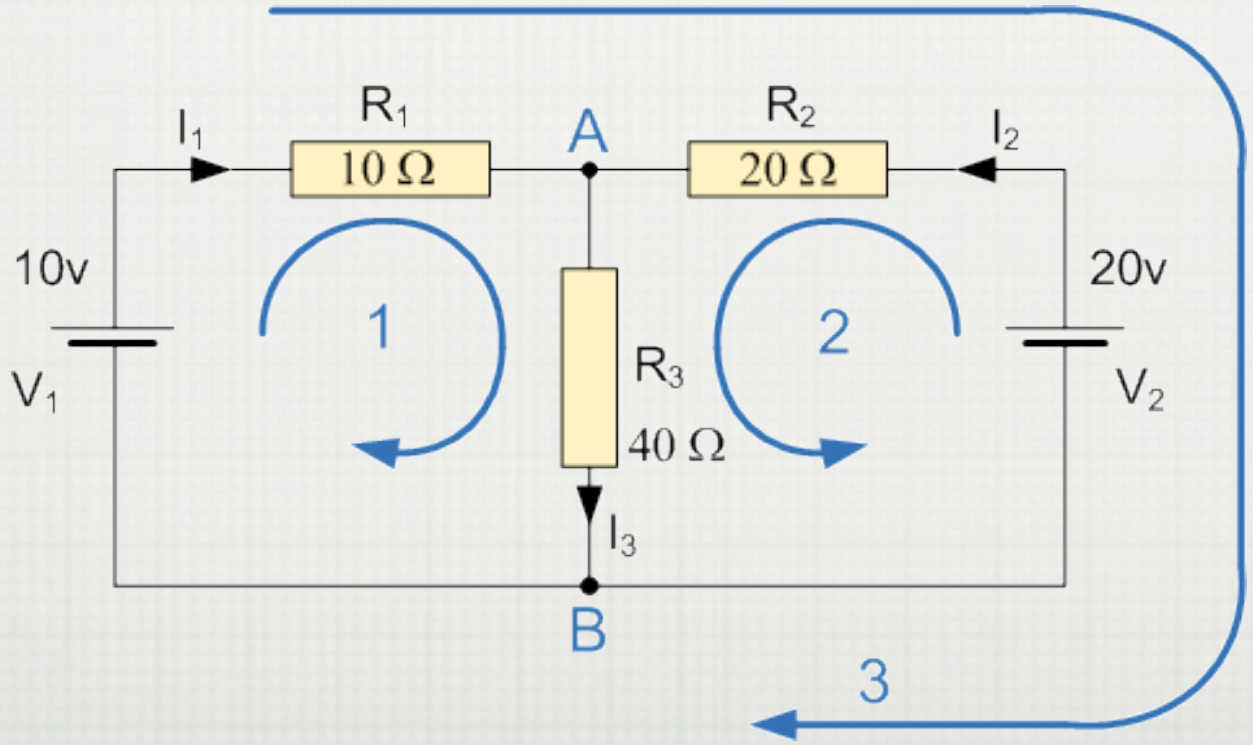
$$\sum_{k=1}^{n} I_k = 0$$

VOLTAGE LAW

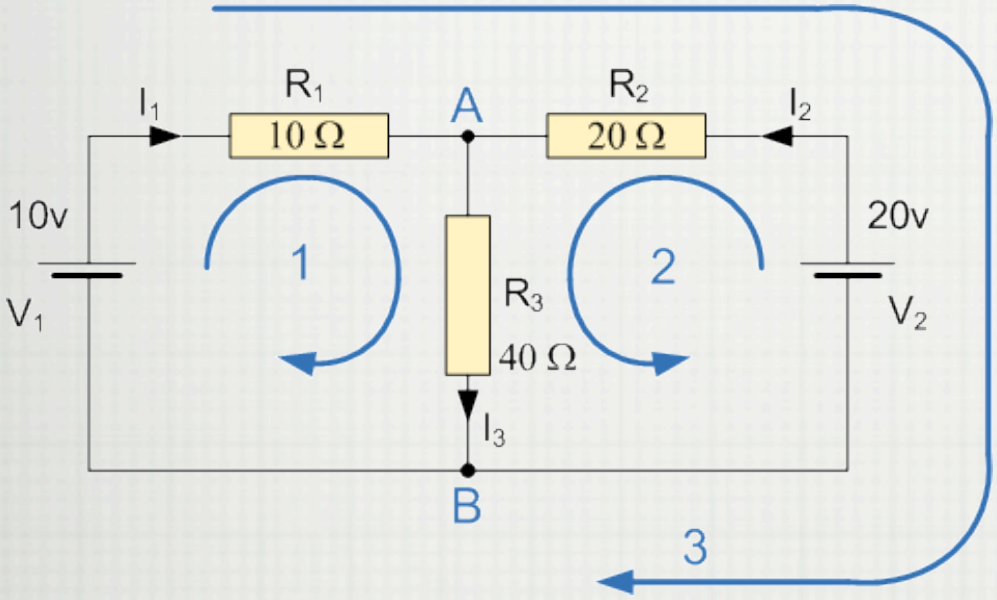


$$\sum_{k=1}^{n} V_k = 0$$

EXAMPLE



EXPLICITLY



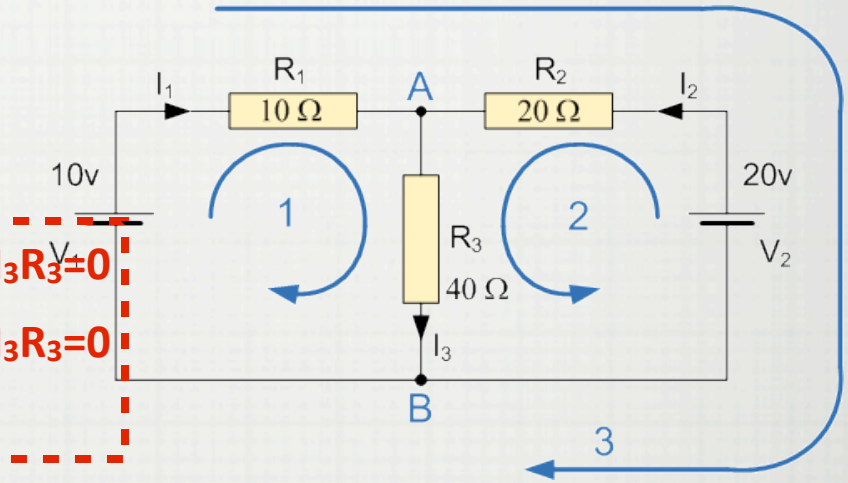
- **NODE A**
  - $I_1+I_2=I_3$

- **Loop 1:**
  - Drop at the battery: $+V_1$
  - Drop at R_1: $-I_1R_1$
  - Drop at R_3: $-I_3R_3$
  - $V_1-I_1R_1-I_3R_3=0$
- **Loop 2:**
  - Drop at the battery: $+V_2$
  - Drop at R_2: $-I_2R_2$
  - Drop at R_3: $-I_3R_3$
  - $V_2-I_2R_2-I_3R_3=0$

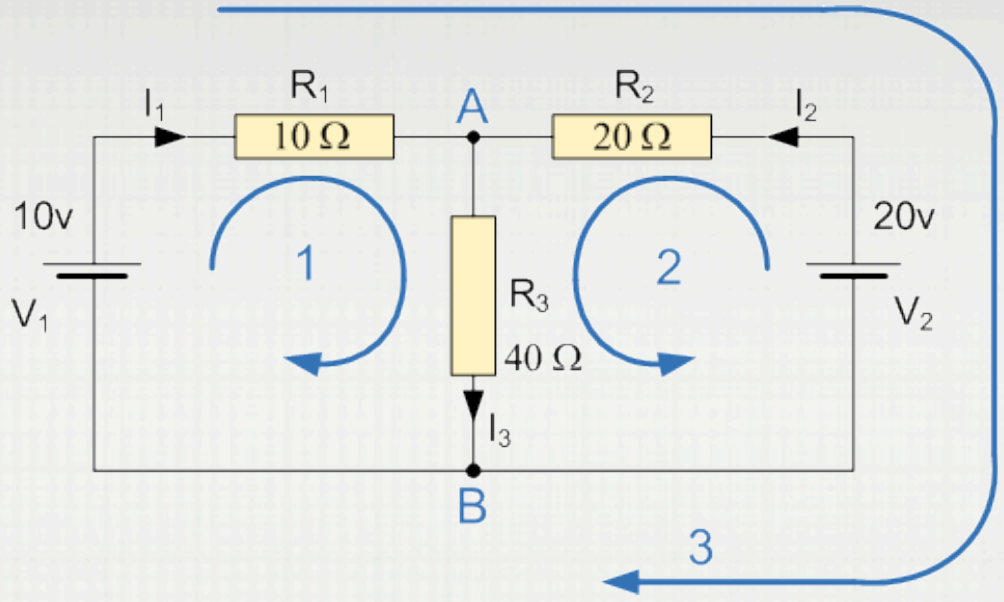
MATRIX FORM

$$\begin{pmatrix} R_1 & 0 & R_3 \\ 0 & R_2 & R_3 \\ 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} V_1 \\ V_2 \\ 0 \end{pmatrix}$$

- $V_1 - I_1R_1 - I_3R_3 = 0$
- $V_2 - I_2R_2 - I_3R_3 = 0$
- $I_1 + I_2 = I_3$



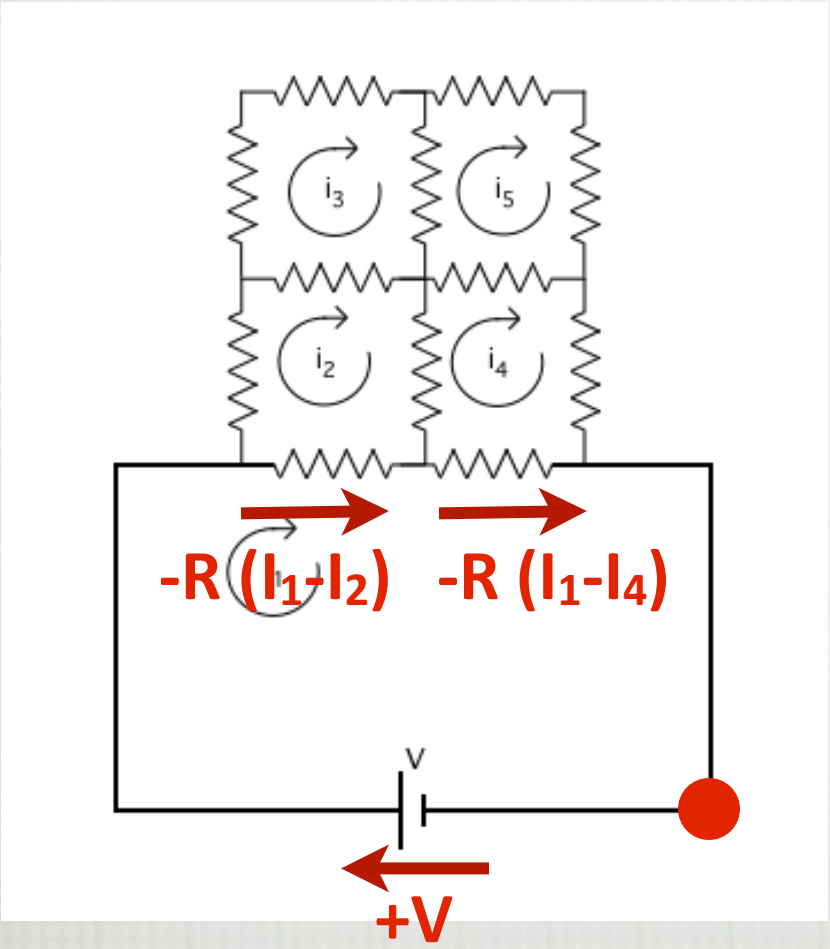
$$\begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} R_1 & 0 & R_3 \\ 0 & R_2 & R_3 \\ 1 & 1 & -1 \end{pmatrix}^{-1} \begin{pmatrix} V_1 \\ V_2 \\ 0 \end{pmatrix}$$



$$\begin{pmatrix} 1 & 0 & 4 \\ 0 & 2 & 4 \\ 1 & 1 & -1 \end{pmatrix}^{-1} = \frac{-1}{14} \begin{pmatrix} -6 & 4 & -8 \\ 4 & -5 & -4 \\ -2 & -1 & -2 \end{pmatrix} \qquad \begin{pmatrix} V_1 \\ V_2 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}$$

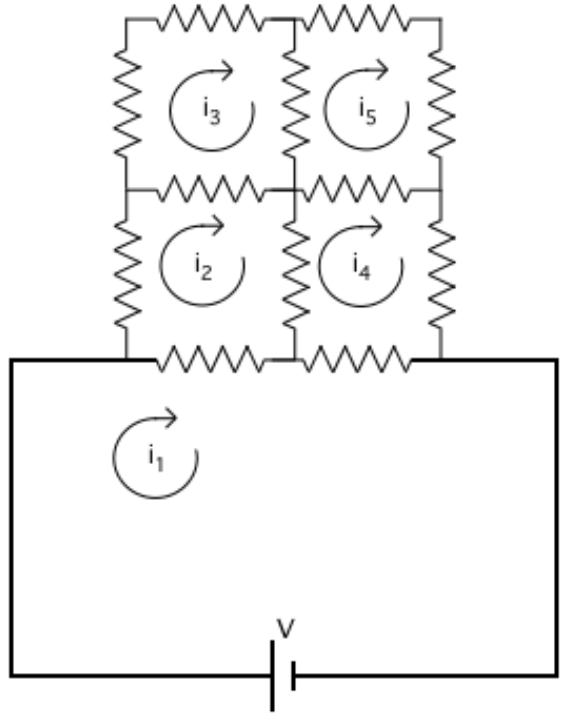
SOLUTION: $$\begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} -1/7 \\ 3/7 \\ 2/7 \end{pmatrix}$$

MORE COMPLICATED PROBLEMS



- Loop 1: $V-R(i_1-i_2) - R(i_1-i_4) =0$
- Loop 2: $0= Ri_2 + R(i_2-i_3) + R(i_2-i_4) + R(i_2-i_1)$
- Loop 3: $0= 2\,Ri_3 + R(i_3-i_5) + R(i_3-i_2)$
- Loop 4: $0= Ri_4 + R(i_4-i_1) + R(i_4-i_2) + R(i_4-i_5)$
- Loop 5: $0= 2Ri_5 + R(i_5-i_4) + R(i_5-i_3)$

- Loop 1: $V=2i_1-i_2-i_4$
- Loop 2: $0= -i_1+4i_2-i_3-i_4$
- Loop 3: $0= 4i_3 -i_5-i_2$
- Loop 4: $0= 4i_4-i_1-i_2-i_5$
- Loop 5: $0= 4i_5-i_4-i_3$



CURRENTS=

i_1= 1.6

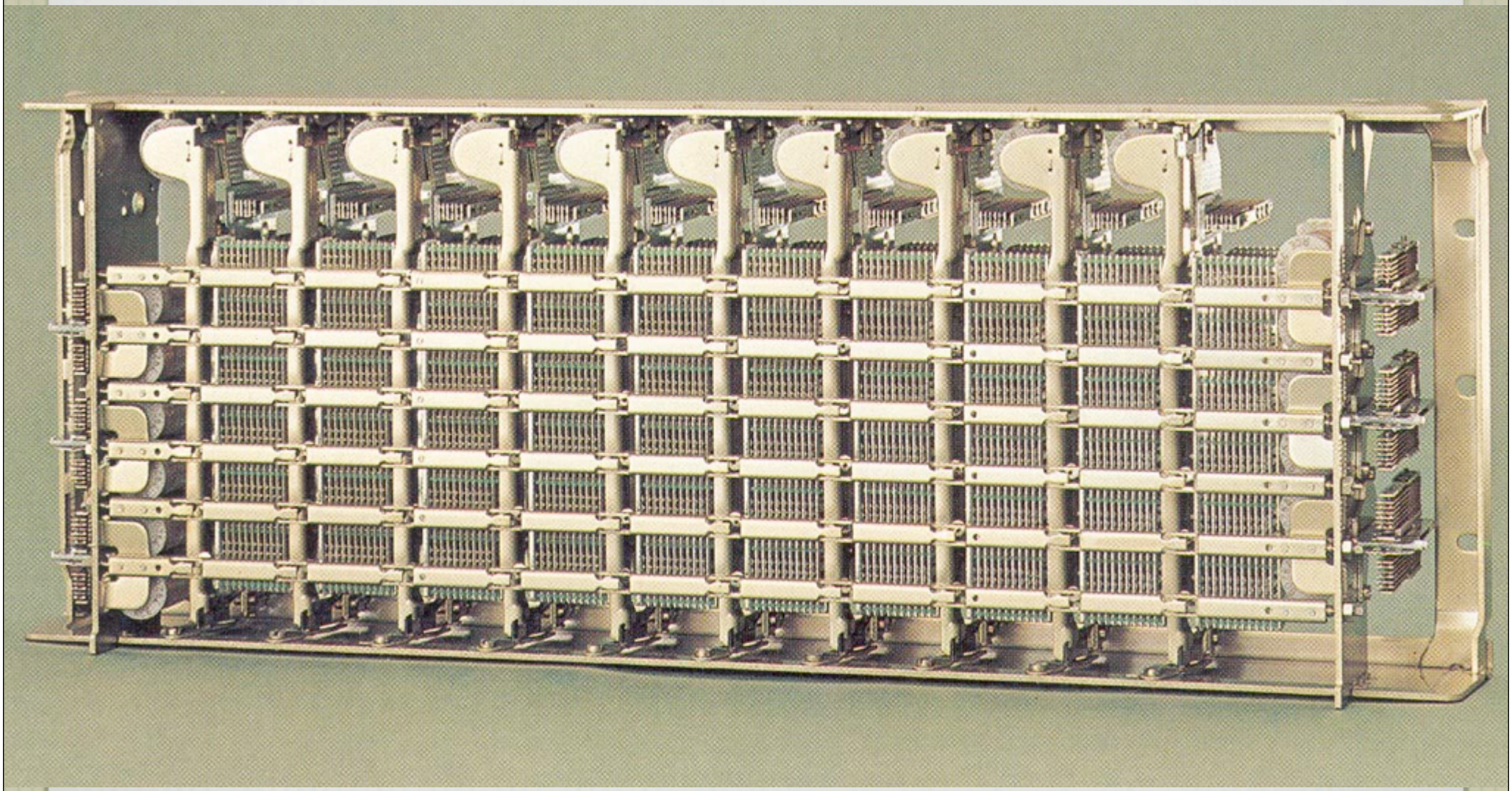
i_2=0.6

i_3=0.2

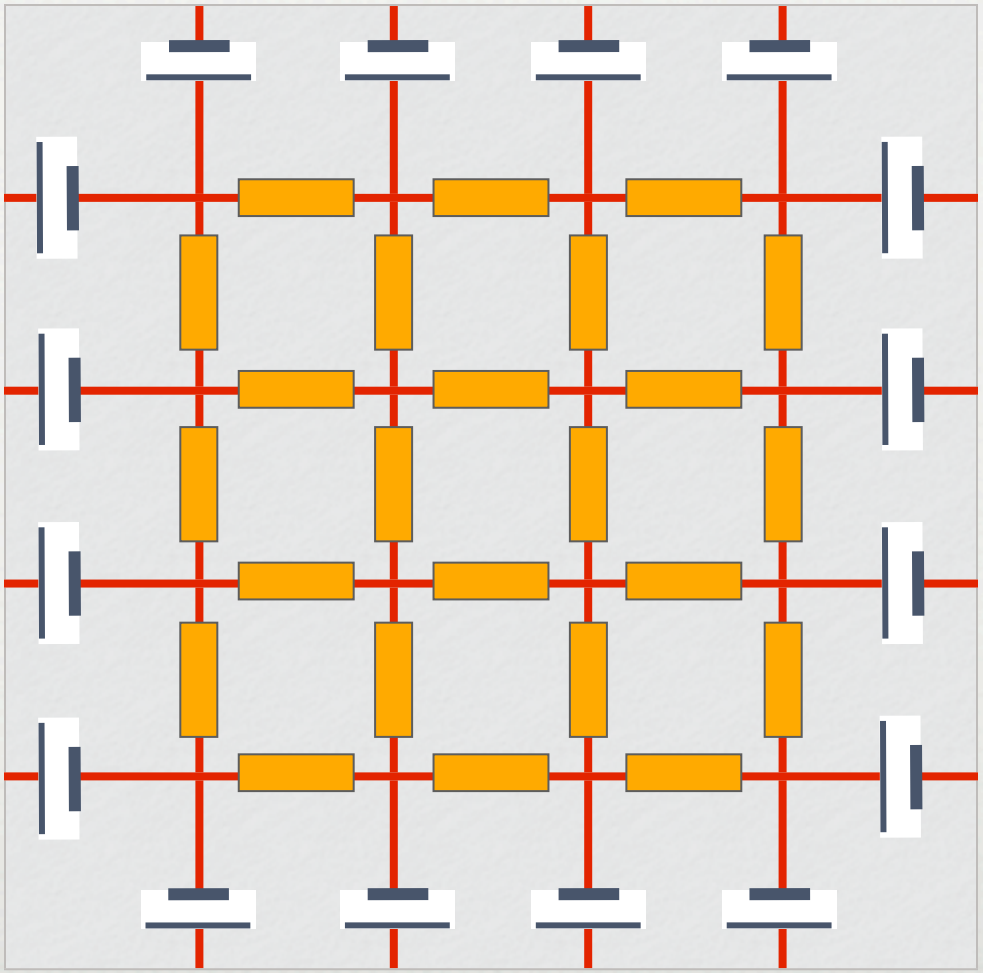
i_4=0.6

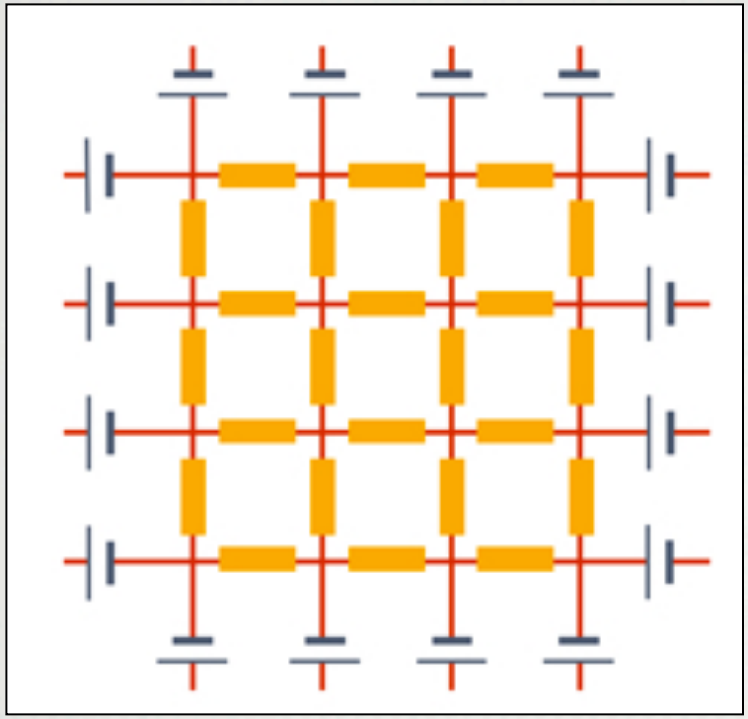
i_5=0.2

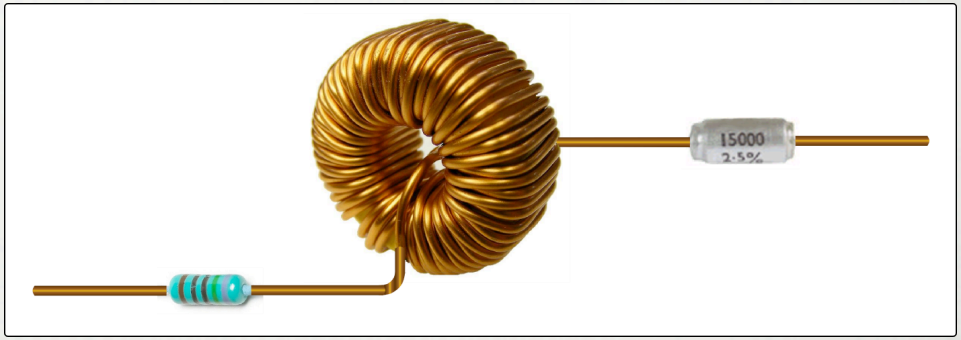
$$\begin{pmatrix} 2 & -1 & 0 & -1 & 0 \\ -1 & 4 & -1 & -1 & 0 \\ 0 & -1 & 4 & 0 & -1 \\ -1 & -1 & 0 & 4 & -1 \\ 0 & 0 & -1 & -1 & 4 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \\ i_5 \end{pmatrix} = \begin{pmatrix} V \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$



OUR OBJECTIVE

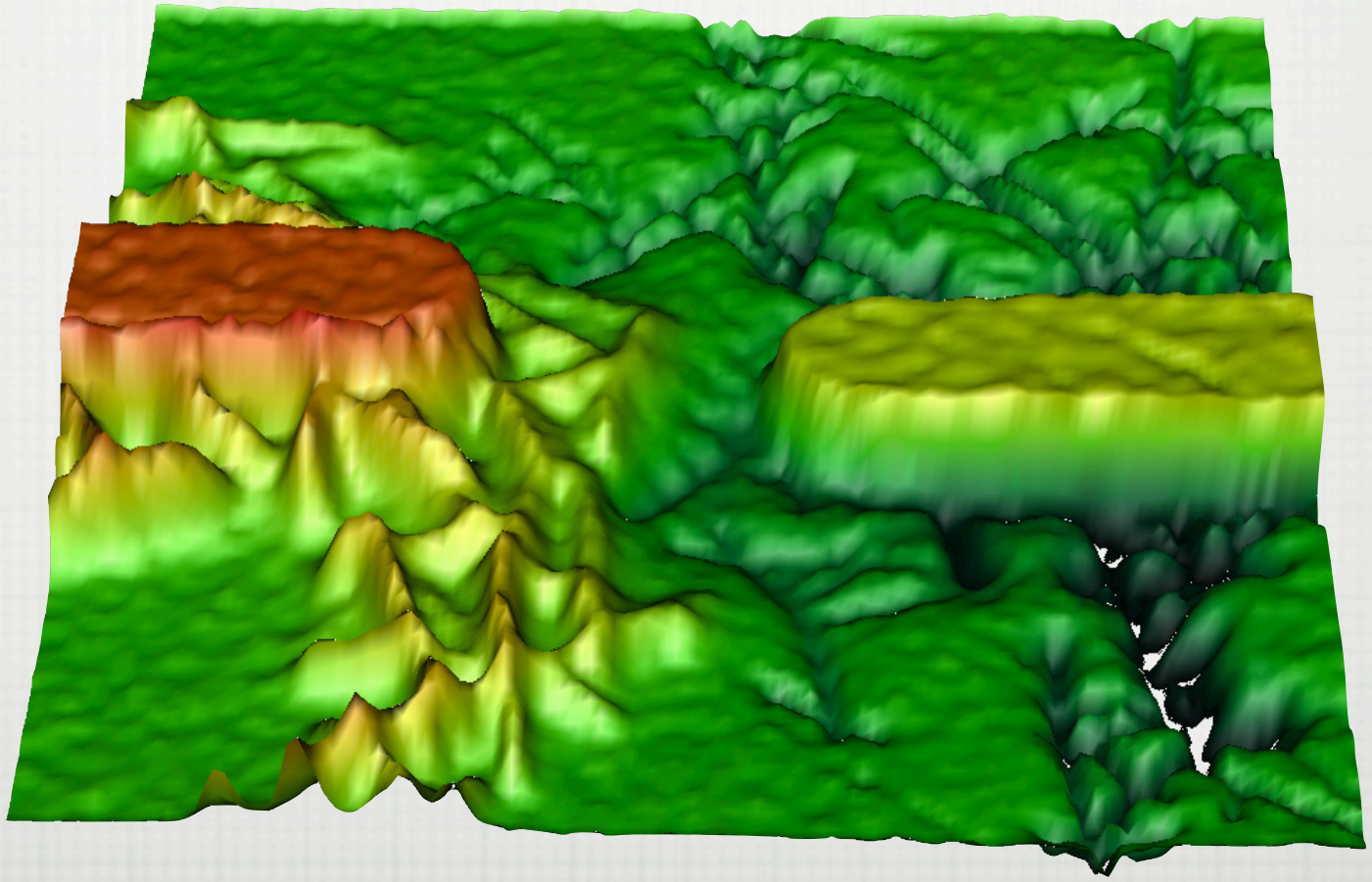






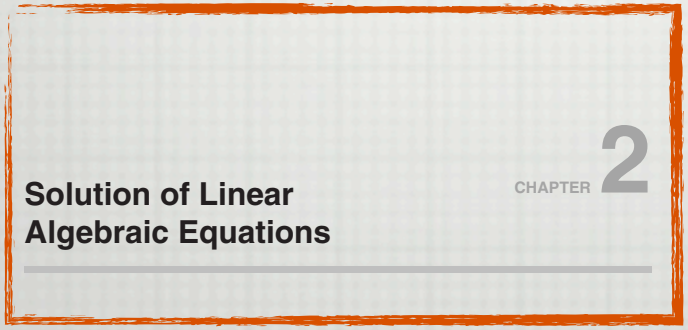
- can be any type of “impedance”, z
  - (explicitly dependent on V)
  - (capacitive)
  - (inductive)
  - (resistive)
  - ...
- number of resistors in excess of thousands!

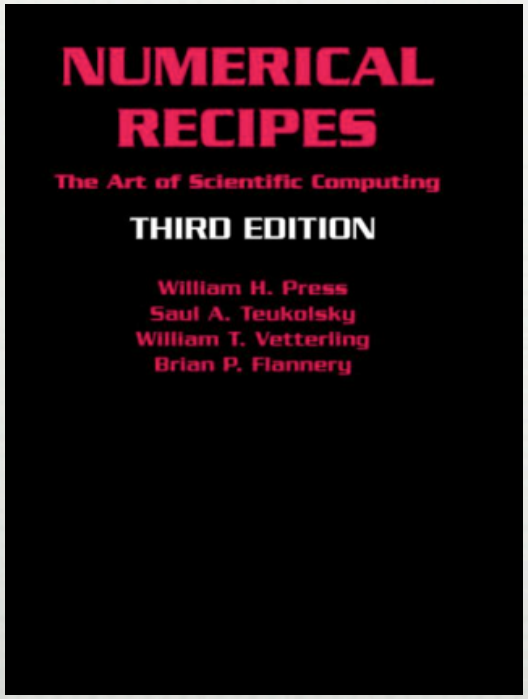
EQUIVALENT CIRCUIT MODEL

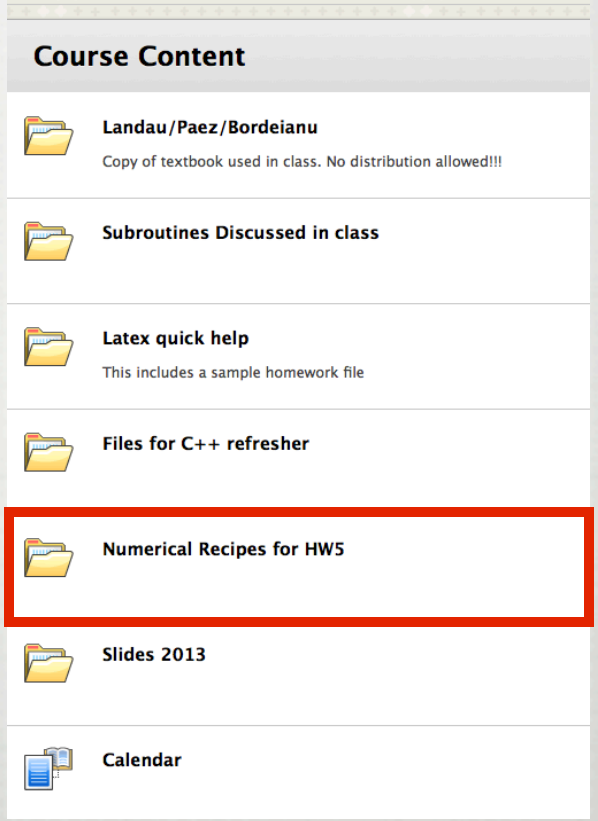


MAIN REFERENCE FOR THIS LECTURE

- ☐ **Numerical Recipes (chapter 2 is loaded on LMS)**
- ☐ **Chapter 2; Libraries used: gaussj; cholesky, LU, sparse (on LMS)**







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

$$x = A^{-1} b$$

TYPICAL FORM OF A SYSTEM OF LINEAR EQUATIONS

$$
\begin{aligned}
a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \cdots + a_{0,N-1}x_{N-1} &= b_0 \\
a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \cdots + a_{1,N-1}x_{N-1} &= b_1 \\
a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + \cdots + a_{2,N-1}x_{N-1} &= b_2 \\
\cdots \qquad\qquad\qquad\qquad\qquad & \cdots \\
a_{M-1,0}x_0 + a_{M-1,1}x_1 + \cdots + a_{M-1,N-1}x_{N-1} &= b_{M-1}
\end{aligned}
$$

MATRIX NOTATION

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,N-1} \\ a_{10} & a_{11} & \cdots & a_{1,N-1} \\ & \cdots & & \\ a_{M-1,0} & a_{M-1,1} & \cdots & a_{M-1,N-1} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ \cdots \\ b_{M-1} \end{bmatrix}$$

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

SINGULAR VERSUS NON-SINGULAR

- Singular system: (determinant=0)
  - "Row degeneracy" (i.e. one row is a linear combination of other rows)
  - equivalent to an "Under-determined system"
- Numerically:
  - Rows could be degenerate within roundoff-error!
  - Accumulated roundoff errors during solution yield singular system, especially if N is large
- Algorithms seek to
  - Avoid accumulating round-off errors
  - Accelerate calculations by taking advantage of form of matrix
  - Accelerate calculations by taking advantage of theorems of linear algebra
- Good news: it is easy to check solution!

SIZE ROUTINELY SOLVABLE

- **Dense matrices: a few thousands**
- **Sparse matrices: a few millions**
- **Sometime much care has to be taken, especially for close-to singular problems (i.e. singular value decomposition)**

MAIN ISSUES, IN A NUTSHELL

- **Accuracy**
- **Speed**
- **Memory Requirement**

BY-PRODUCTS OF SOLVING Ax=B

- **Matrix inverse**
- **Determinant**
- **Underdetermined systems (fewer equations than unknowns)**
  - **-> singular value decomposition**
- **Overdetermined systems (more equations than unknowns)**
  - **-> linear least-square problem**

THERE ARE MANY TOPICS OUT THERE…

- **But we can't investigate them all in one lecture**
- **My goal is to convince you that depending on the problem you need to carefully choose your linear solver.**
- **A clever choice can yield ORDERS of magnitude in improvements (accuracy, time, or most importantly memory consumption)**

EXAMPLE OF SOFTWARE: LAPACK

		SINGLE PRECISION		DOUBLE PRECISION	
TYPE	**DESCRIPTION**	**REAL**	**COMPLEX**	**REAL**	**COMPLEX**
General	Solves a general system of linear equations AX=B.	SGESV	CGESV	DGESV	ZGESV
	Solves a general system of linear equations AX=B with iterative refinement.	–	–	DSGESV	ZCGESV
	Computes an LU factorization of a general matrix, using partial pivoting with row interchanges.	SGETRF	CGETRF	DGETRF	ZGETRF
	Computes the inverse of a general matrix, using the LU factorization.	SGETRI	CGETRI	DGETRI	ZGETRI
	Solves a general system of linear equations AX=B, $A^TX=B$, or $A^HX=B$, using the LU factorization.	SGETRS	CGETRS	DGETRS	ZGETRS
Positive Definite	Solves a symmetric positive definite system of linear equations AX=B.	SPOSV	CPOSV	DPOSV	ZPOSV
	Computes the Cholesky factorization of a symmetric positive definite matrix.	SPOTRF	CPOTRF	DPOTRF	ZPOTRF
	Solves a symmetric positive definite system of linear equations AX=B, using the Cholesky factorization.	SPOTRS	CPOTRS	DPOTRS	ZPOTRS
Triangular	Solves a triangular system of linear equations AX=B, $A^TX=B$, or $A^HX=B$.	STRTRS	CTRTRS	DTRTRS	ZTRTRS
	Computes the inverse of a triangular matrix.	STRTRI	CTRTRI	DTRTRI	ZTRTRI
General Banded	Computes an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges.	SGBTRF	CGBTRF	DGBTRF	ZGBTRF
Positive Definite Banded	Computes the Cholesky factorization of a real symmetric positive definite band matrix A.	SPBTRF	CPBTRF	DPBTRF	ZPBTRF

LAPACK: SVD

		SINGLE PRECISION		DOUBLE PRECISION	
TYPE	**DESCRIPTION**	**REAL**	**COMPLEX**	**REAL**	**COMPLEX**
General	Computes the singular value decomposition (SVD) of a general rectangular matrix.	SGESVD	CGESVD	DGESVD	ZGESVD
	Reduces a general rectangular matrix to real bidiagonal form by an orthogonal transformation.	SGEBRD	CGEBRD	DGEBRD	ZGEBRD
	Generates the orthogonal transformation matrices from a reduction to bidiagonal form.	SORGBR	CUNGBR	DORGBR	ZORGBR
Bidiagonal	Computes the singular value decomposition (SVD) of a real bidiagonal matrix, using the bidiagonal QR algorithm.	SBDSQR	CBDSQR	DBDSQR	ZBDSQR

SURVEY OF NUMERICAL TECHNIQUES

- ☐ **Gauss-Jordan Elimination**
- ☐ **LU decomposition**
- ☐ **Tridiagonal and band-diagonal systems**
- ☐ **Iterative Improvement of a solution to linear equations**
- ☐ **Singular Value Decomposition**
- ☐ **Sparse Linear Systems**
- ☐ **Cholesky Decomposition**
- ☐ **QR Decomposition**

SURVEY OF NUMERICAL TECHNIQUES

- ☐ **Gauss-Jordan Elimination**
- ☐ **LU decomposition**
- ☐ **Tridiagonal and band-diagonal systems**
- ☐ **Iterative Improvement of a solution to linear equations**
- ☐ **Singular Value Decomposition**
- ☐ **Sparse Linear Systems**
- ☐ **Cholesky Decomposition**
- ☐ **QR Decomposition**

GAUSS-JORDAN ELIMINATION

- Use same technique as the one you se when you calculate A^{-1} manually
- Up to 3-5 times slower than "more clever" methods (unless you need A^{-1} explicitly)
- Solid method
- Uses pivoting
  - idea #1 : changing row order of A (and corresponding b) does not change the problem
  - idea #2: replacing a row of A by a combination of other row and itself (and corresponding b) does not change the problem
  - idea #3: changing two columns of A (and corresponding order of x) does not change the problem
- see gaussj.h for C++ implementation from Numerical Recipes

PIVOTING

- No pivoting: only operation 2 is used
- Pivoting means we change order of rows and columns
- In fact no pivoting means ... roundoff errors will yield unstable algorithm
- Note that you can have partial and full pivoting

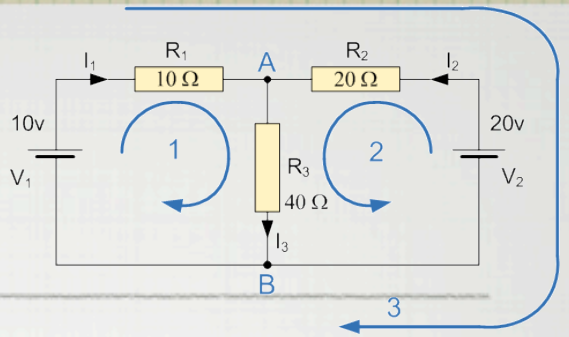
GJ: REFERENCE CARD (gausj.h)

`void gaussj(MatDoub_IO &a, MatDoub_IO &b)`

Linear equation solution by Gauss-Jordan elimination, equation (2.1.1) above. The input matrix is `a[0..n-1][0..n-1]`. `b[0..n-1][0..m-1]` is input containing the `m` right-hand side vectors. On output, `a` is replaced by its matrix inverse, and `b` is replaced by the corresponding set of solution vectors.

- **Usage:**
  - **MatDoub_IO A(10,10);**
  - **MatDoub_IO b(10,1);**
  - **gaussj(a,b);**
  - **solution is now in b, A^{-1} is in A**

EXAMPLE




```
#include <iostream>
#include <fstream>
#include <cmath>
#include "nr3.h"
#include "gaussj.h"
using namespace std;
int main() {
    int size=3;
    MatDoub_IO A(size,size);
    MatDoub_IO b(size,1);

    //Define LHS
    //first row
    A[0][0]=1;
    A[0][1]=0;
    A[0][2]=4;

    //second row
    A[1][0]=0;
    A[1][1]=2;
    A[1][2]=4;

    //third row
    A[2][0]=1;
    A[2][1]=1;
    A[2][2]=-1;

    //Define RHS
    b[0][0]=1;
    b[1][0]=2;
    b[2][0]=0;

    //perform Gauss-Jordan
    gaussj(A,b);

    cout << "Solution is: \n";
    for (int i=0;i<3;i++){
        cout << i << ":\t" << b[i][0] << endl;
    }
    cout<<"Finished"<<endl;
    return 0;
}
```

$$\begin{pmatrix} 1 & 0 & 4 \\ 0 & 2 & 4 \\ 1 & 1 & -1 \end{pmatrix}^{-1} = \frac{-1}{14} \begin{pmatrix} -6 & 4 & -8 \\ 4 & -5 & -4 \\ -2 & -1 & -2 \end{pmatrix}$$

```
bash-3.2$ ./a.out
Solution is:
0:	-0.142857
1:	0.428571
2:	0.285714
Finished
```

$$\begin{pmatrix} V_1 \\ V_2 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} -1/7 \\ 3/7 \\ 2/7 \end{pmatrix}$$

LU DECOMPOSITION

- **We try to decompose A as :**

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A}$$

- **With:**

$$\begin{bmatrix} \alpha_{00} & 0 & 0 & 0 \\ \alpha_{10} & \alpha_{11} & 0 & 0 \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & 0 \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \cdot \begin{bmatrix} \beta_{00} & \beta_{01} & \beta_{02} & \beta_{03} \\ 0 & \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & 0 & \beta_{33} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

ADVANTAGE OF LU DECOMPOSITION?

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

We have now two problems to solve:

1. finding y such that:

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$$

Simple Forward substitution!

2. finding x such that:

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$$

Simple Backward substitution!

LU: WHAT'S THE ADVANTAGE?

- **We end up with two trivial problems!**
- **(1) y can be found by forward substitution**

$$y_0 = \frac{b_0}{\alpha_{00}}$$

$$y_i = \frac{1}{\alpha_{ii}}\left[b_i - \sum_{j=0}^{i-1} \alpha_{ij}\, y_j\right] \qquad i = 1, 2, \ldots, N-1$$

- **(2) x can be found by backward substitution**

$$x_{N-1} = \frac{y_{N-1}}{\beta_{N-1,N-1}}$$

$$x_i = \frac{1}{\beta_{ii}}\left[y_i - \sum_{j=i+1}^{N-1} \beta_{ij}\, x_j\right] \qquad i = N-2, N-3, \ldots, 0$$

LU DECOMPOSITION

$$
\begin{aligned}
i < j: &\quad \alpha_{i0}\beta_{0j} + \alpha_{i1}\beta_{1j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \\
i = j: &\quad \alpha_{i0}\beta_{0j} + \alpha_{i1}\beta_{1j} + \cdots + \alpha_{ii}\beta_{jj} = a_{ij} \\
i > j: &\quad \alpha_{i0}\beta_{0j} + \alpha_{i1}\beta_{1j} + \cdots + \alpha_{ij}\beta_{jj} = a_{ij}
\end{aligned}
$$

- Set $\alpha_{ii} = 1, i = 0, \ldots, N-1$ (equation 2.3.11).
- For each $j = 0, 1, 2, \ldots, N-1$ do these two procedures: First, for $i = 0, 1, \ldots, j$, use (2.3.8), (2.3.9), and (2.3.11) to solve for β_{ij}, namely

$$\beta_{ij} = a_{ij} - \sum_{k=0}^{i-1} \alpha_{ik}\beta_{kj} \tag{2.3.12}$$

(When $i = 0$ in 2.3.12 the summation term is taken to mean zero.) Second, for $i = j+1, j+2, \ldots, N-1$ use (2.3.10) to solve for α_{ij}, namely

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=0}^{j-1} \alpha_{ik}\beta_{kj} \right) \tag{2.3.13}$$

Be sure to do both procedures before going on to the next j.

- **Crout's algorithm..... beyond the scope of this course**

LU: REFERENCE CARD (LUDCMP.H)

struct LUdcmp
Object for solving linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ using LU decomposition, and related functions.

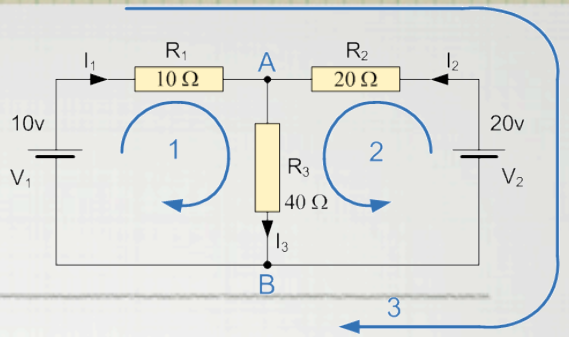
```
{
    Int n;
    MatDoub lu;                                          Stores the decomposition.
    VecInt indx;                                         Stores the permutation.
    Doub d;                                              Used by det.
    LUdcmp(MatDoub_I &a);                                Constructor. Argument is the matrix A.
    void solve(VecDoub_I &b, VecDoub_O &x);              Solve for a single right-hand side.
    void solve(MatDoub_I &b, MatDoub_O &x);              Solve for multiple right-hand sides.
    void inverse(MatDoub_O &ainv);                       Calculate matrix inverse A^-1.
    Doub det();                                          Return determinant of A.
    void mprove(VecDoub_I &b, VecDoub_IO &x);                    Discussed in §2.5.
    MatDoub_I &aref;                                     Used only by mprove.
};
```

LU: IN PRACTICE

```
const Int n = ...
MatDoub a(n,n);
VecDoub b(n),x(n);
...
LUdcmp alu(a);
alu.solve(b,x);
```

- can also compute inverse, and determinant
- if GJ ~ N^3; LU ~ $N^3/3$

EXAMPLE




```
#include <iostream>
#include <fstream>
#include <cmath>
#include "nr3.h"
#include "ludcmp.h"

using namespace std;

int main() {
    int size=3;

    MatDoub_IO A(size,size);
    VecDoub b(size), x(size);
    //Define LHS
    //first row
    A[0][0]=1;
    A[0][1]=0;
    A[0][2]=4;

    //second row
    A[1][0]=0;
    A[1][1]=2;
    A[1][2]=4;

    //third row
    A[2][0]=1;
    A[2][1]=1;
    A[2][2]=-1;

    //Define RHS
    b[0]=1;
    b[1]=2;
    b[2]=0;

    //perform LU Decomposition
    LUdcmp alu(A);

    //solve problem
    alu.solve(b,x);

    cout << "LU Solution is: \n";
    for (int i=0;i<3;i++){
        cout << i << ":\t" << x[i] << endl;
    }
    cout<<"Finished"<<endl;
    return 0;
}
```

$$\begin{pmatrix} 1 & 0 & 4 \\ 0 & 2 & 4 \\ 1 & 1 & -1 \end{pmatrix}^{-1} = \frac{-1}{14} \begin{pmatrix} -6 & 4 & -8 \\ 4 & -5 & -4 \\ -2 & -1 & -2 \end{pmatrix}$$

bash-3.2$ ./a.out
LU Solution is:
0: -0.142857
1: 0.428571
2: 0.285714
Finished

$$\begin{pmatrix} V_1 \\ V_2 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} -1/7 \\ 3/7 \\ 2/7 \end{pmatrix}$$

CHOLESKY DECOMPOSITION

- If a square matrix A happens to be symmetric and positive-definite, then it has a special, more efficient, triangular decomposition.

- Instead of seeking arbitrary lower and upper triangular factors L and U, Cholesky de- composition constructs a lower triangular matrix L whose transpose L^T can itself serve as the upper triangular part. In other words we have

$$\mathbf{L} \cdot \mathbf{L}^T = \mathbf{A}$$

- This factorization is sometimes referred to as "taking the square root" of the matrix A, though, because of the transpose, it is not literally that. The components of L^T are of course related to those of L by

$$L_{ij}^T = L_{ji}$$

- **SPEEDUP: A FACTOR OF 2 COMPARED TO LU DECOMPOSITION**

$$L_{ii} = \left(a_{ii} - \sum_{k=0}^{i-1} L_{ik}^2 \right)^{1/2}$$

$$L_{ji} = \frac{1}{L_{ii}} \left(a_{ij} - \sum_{k=0}^{i-1} L_{ik} L_{jk} \right) \qquad j = i+1, i+2, \ldots, N-1$$

REFERENCE CARD: CHOLESKY.H

`struct Cholesky{`

Object for Cholesky decomposition of a matrix $\mathbf{A}$, and related functions.

`Int n;`

`MatDoub el;` Stores the decomposition.

`Cholesky(MatDoub_I &a) : n(a.nrows()), el(a) {`

Constructor. Given a positive-definite symmetric matrix `a[0..n-1][0..n-1]`, construct and store its Cholesky decomposition, $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T$.

Example:

```
#include “cholesky.h”;

MatDoub lap(4,4); //declaration
VECDOUB B[4], X[4];

Cholesky Laplace (lap); //class constructor with lap
Laplace.solve(b,x); //compute inverse
```

TRIDIAGONAL AND BAND DIAGONAL

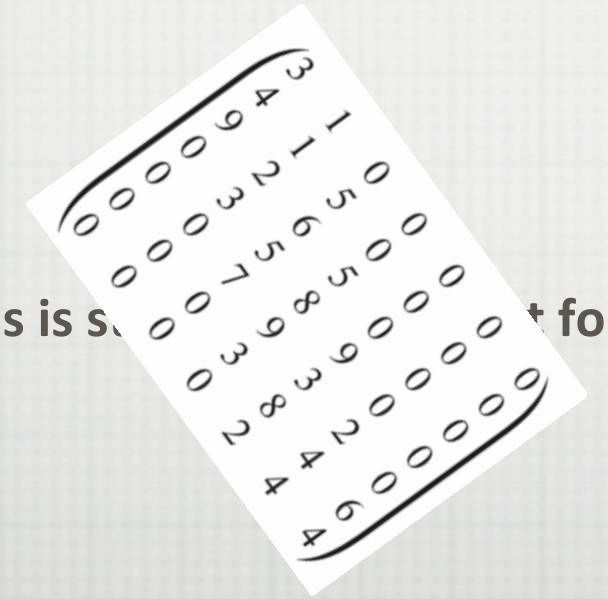
$$\begin{bmatrix} b_0 & c_0 & 0 & \cdots & & & \\ a_1 & b_1 & c_1 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-2} & b_{N-2} & c_{N-2} \\ & & & \cdots & 0 & a_{N-1} & b_{N-1} \end{bmatrix} \cdot \begin{bmatrix} u_0 \\ u_1 \\ \cdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ \cdots \\ r_{N-2} \\ r_{N-1} \end{bmatrix}$$

- **only store vectors a, b, c, r!**

```
void tridag(VecDoub_I &a, VecDoub_I &b, VecDoub_I &c, VecDoub_I &r, VecDoub_O &u)
Solves for a vector u[0..n-1] the tridiagonal linear set given by equation (2.4.1). a[0..n-1],
b[0..n-1], c[0..n-1], and r[0..n-1] are input vectors and are not modified.
```

BAND-DIAGONAL

$$\begin{pmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 5 & 0 & 0 & 0 & 0 \\ 9 & 2 & 6 & 5 & 0 & 0 & 0 \\ 0 & 3 & 5 & 8 & 9 & 0 & 0 \\ 0 & 0 & 7 & 9 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 8 & 4 & 6 \\ 0 & 0 & 0 & 0 & 2 & 4 & 4 \end{pmatrix}$$



- This is s[illegible] form

$$\begin{pmatrix} x & x & 3 & 1 \\ x & 4 & 1 & 5 \\ 9 & 2 & 6 & 5 \\ 3 & 5 & 8 & 9 \\ 7 & 9 & 3 & 2 \\ 3 & 8 & 4 & 6 \\ 2 & 4 & 4 & x \end{pmatrix}$$

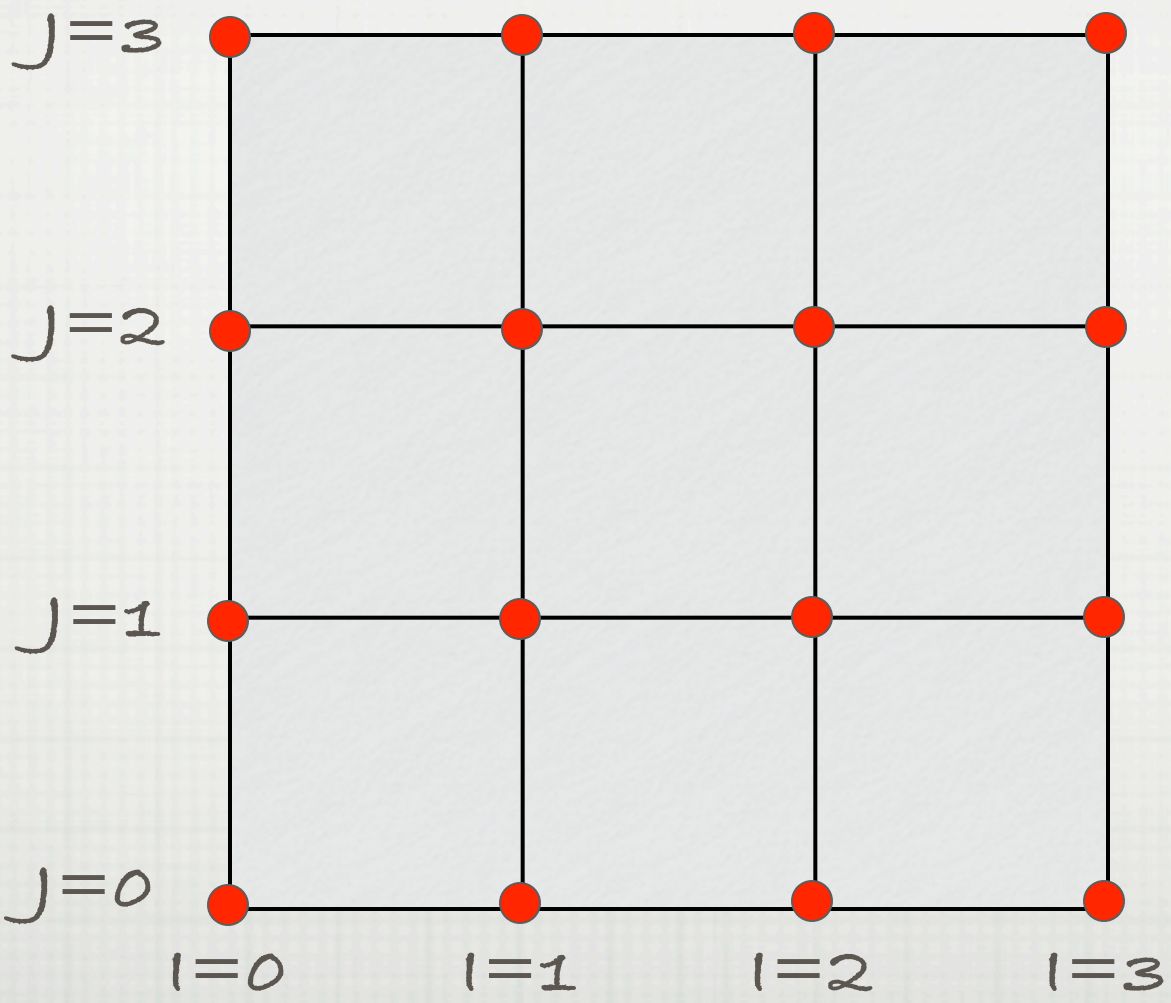
EXPLICIT EXAMPLE: BACK TO LAPLACE

LAPLACE REVISITED (LECTURE 10)

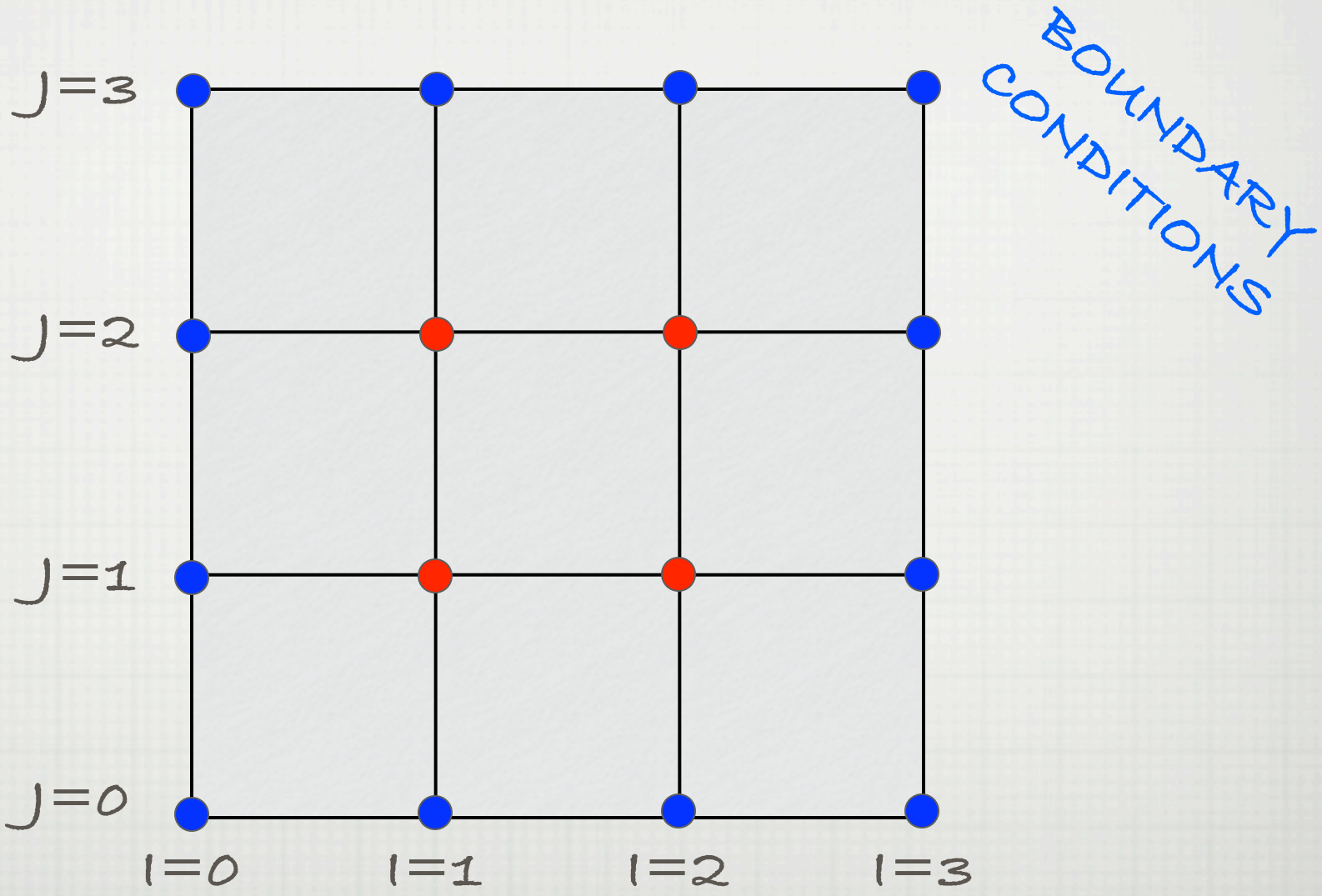
$$\frac{\partial^2 U(x,y)}{\partial x^2} + \frac{\partial^2 U(x,y)}{\partial y^2} = \begin{cases} 0 & \text{Laplace's equation} \\ -4\pi\rho(\mathbf{x}) & \text{Poisson's equation} \end{cases}$$

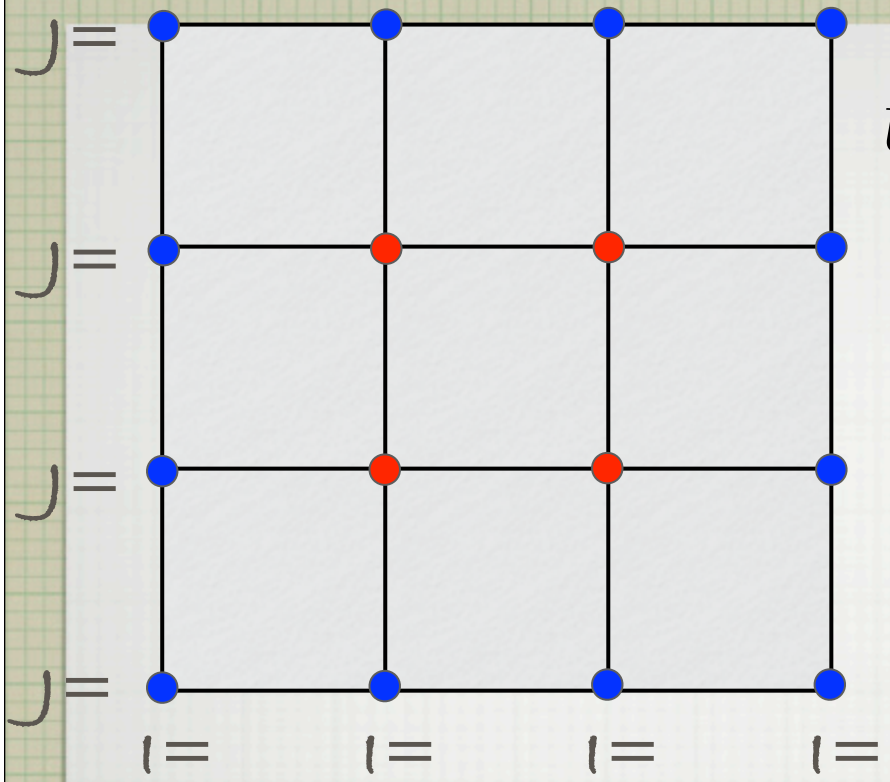
$$U_{i,j} = \tfrac{1}{4}\left[U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}\right] + \pi\rho(i\Delta, j\Delta)\Delta^2$$

$$U_{i,j} = \tfrac{1}{4}\left[U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}\right] + \pi\rho(i\Delta, j\Delta)\Delta^2$$



$$U_{i,j} = \tfrac{1}{4}\left[U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}\right] + \pi\rho(i\Delta, j\Delta)\Delta^2$$





$$U_{i,j} = \tfrac{1}{4}\left[U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}\right]$$

$$U_{11} = 1/4(U_{10} + U_{12} + U_{01} + U_{21})$$
$$U_{21} = 1/4(U_{20} + U_{22} + U_{31} + U_{11})$$
$$U_{12} = 1/4(U_{11} + U_{13} + U_{02} + U_{22})$$
$$U_{22} = 1/4(U_{21} + U_{23} + U_{12} + U_{32})$$

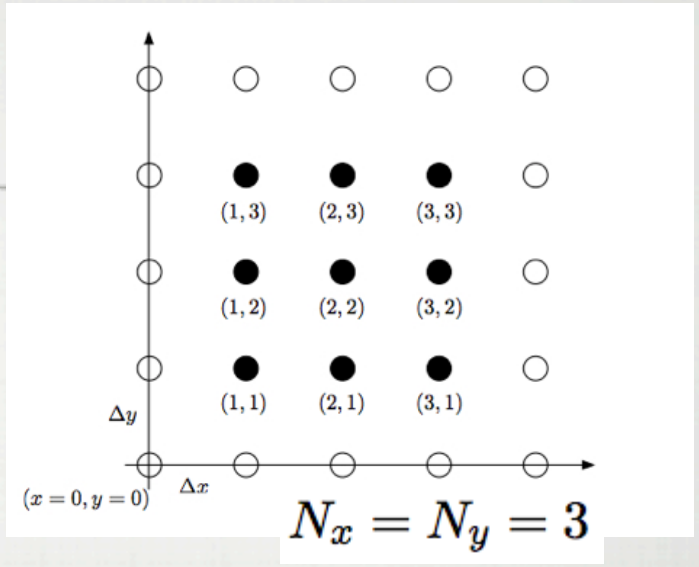
$$
\begin{aligned}
U_{11} &= 1/4(U_{10} + U_{12} + U_{01} + U_{21}) & 4U_{11} - U_{12} - U_{21} &= U_{10} + U_{01} \\
U_{21} &= 1/4(U_{20} + U_{22} + U_{31} + U_{11}) & -U_{11} + 4U_{21} - U_{22} &= U_{20} + U_{31} \\
U_{12} &= 1/4(U_{11} + U_{13} + U_{02} + U_{22}) & -U_{11} + 4U_{12} - U_{22} &= U_{13} + U_{02} \\
U_{22} &= 1/4(U_{21} + U_{23} + U_{12} + U_{32}) & -U_{12} + 4U_{22} - U_{21} &= U_{23} + U_{32}
\end{aligned}
$$

$$
\begin{pmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{pmatrix}
\begin{pmatrix} U_{11} \\ U_{21} \\ U_{12} \\ U_{22} \end{pmatrix}
=
\begin{pmatrix} U_{10} + U_{01} \\ U_{20} + U_{31} \\ U_{13} + U_{02} \\ U_{23} + U_{32} \end{pmatrix}
$$

$$
\begin{pmatrix} U_{11} \\ U_{21} \\ U_{12} \\ U_{22} \end{pmatrix}
=
\begin{pmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{pmatrix}^{-1}
\begin{pmatrix} U_{10} + U_{01} \\ U_{20} + U_{31} \\ U_{13} + U_{02} \\ U_{23} + U_{32} \end{pmatrix}
$$

$$\begin{pmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{pmatrix}^{-1} = \begin{pmatrix} 0.292 & 0.0833 & 0.0833 & 0.0417 \\ 0.0833 & 0.292 & 0.0417 & 0.0833 \\ 0.0833 & 0.0417 & 0.292 & 0.0833 \\ 0.0417 & 0.0833 & 0.0833 & 0.292 \end{pmatrix}$$

ONE WAY TO BUILD THE MATRIX



$$N_x = N_y = 3$$

$$\partial^2 x \equiv \begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}$$

$$\partial^2 y \equiv \begin{pmatrix} -2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -2 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -2 \end{pmatrix}$$

$$\frac{\partial^2 f(x_n, y_m)}{\partial x^2} \sim \frac{f(x_n - \Delta x, y_m) - 2\, f(x_n, y_m) + f(x_n + \Delta x, y_m)}{\Delta x^2}$$

PUTTING IT ALL TOGETHER (LHS)

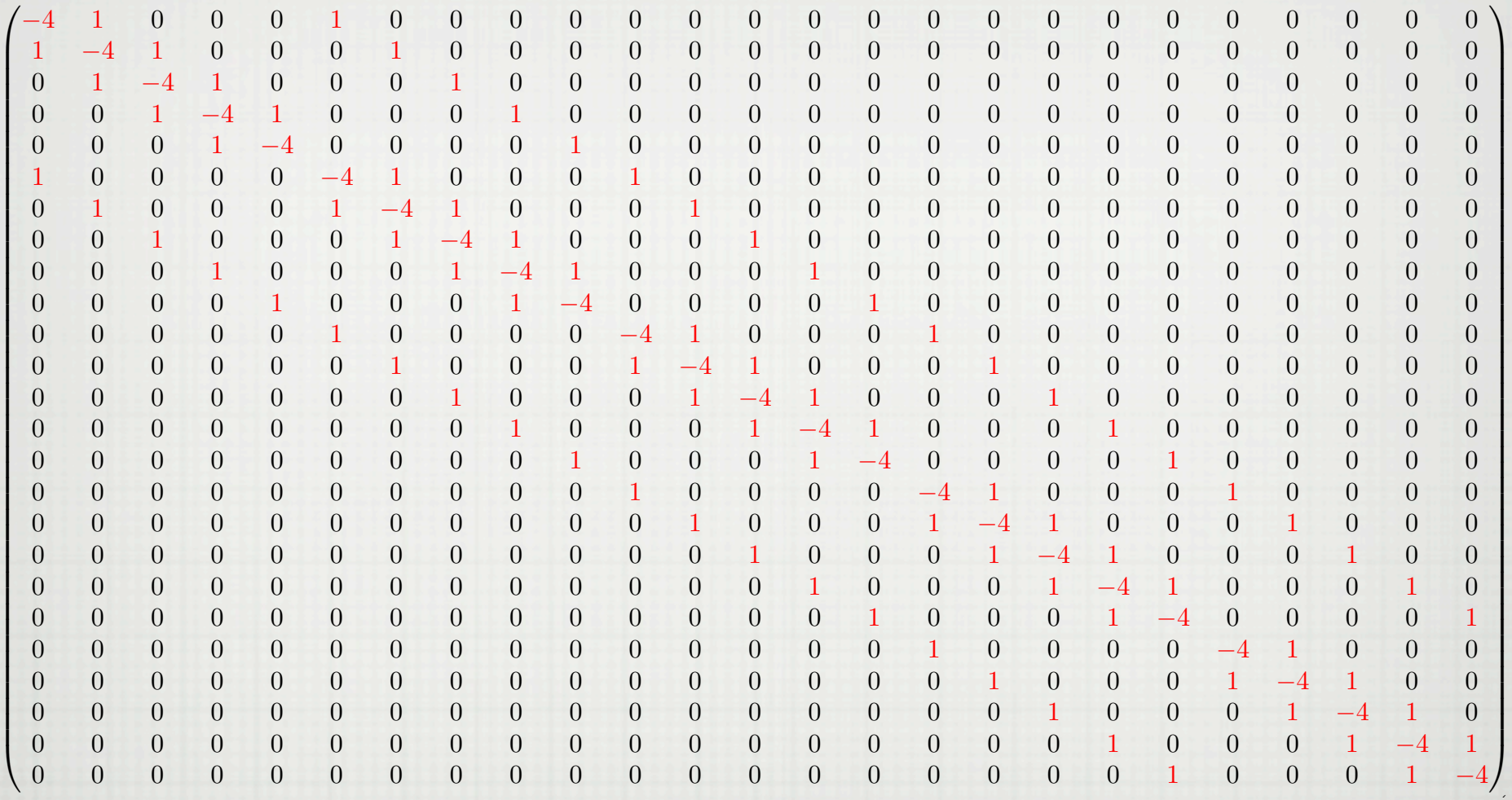
$$\nabla^2 \equiv \begin{pmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{pmatrix}$$

$$\nabla^2 \equiv \begin{pmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{pmatrix}$$

$$\begin{pmatrix} -4 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & -4 \end{pmatrix}$$

n_y x (n_x x n_x) blocks along the diagonal

blocks are connected by a diagonal of “1”



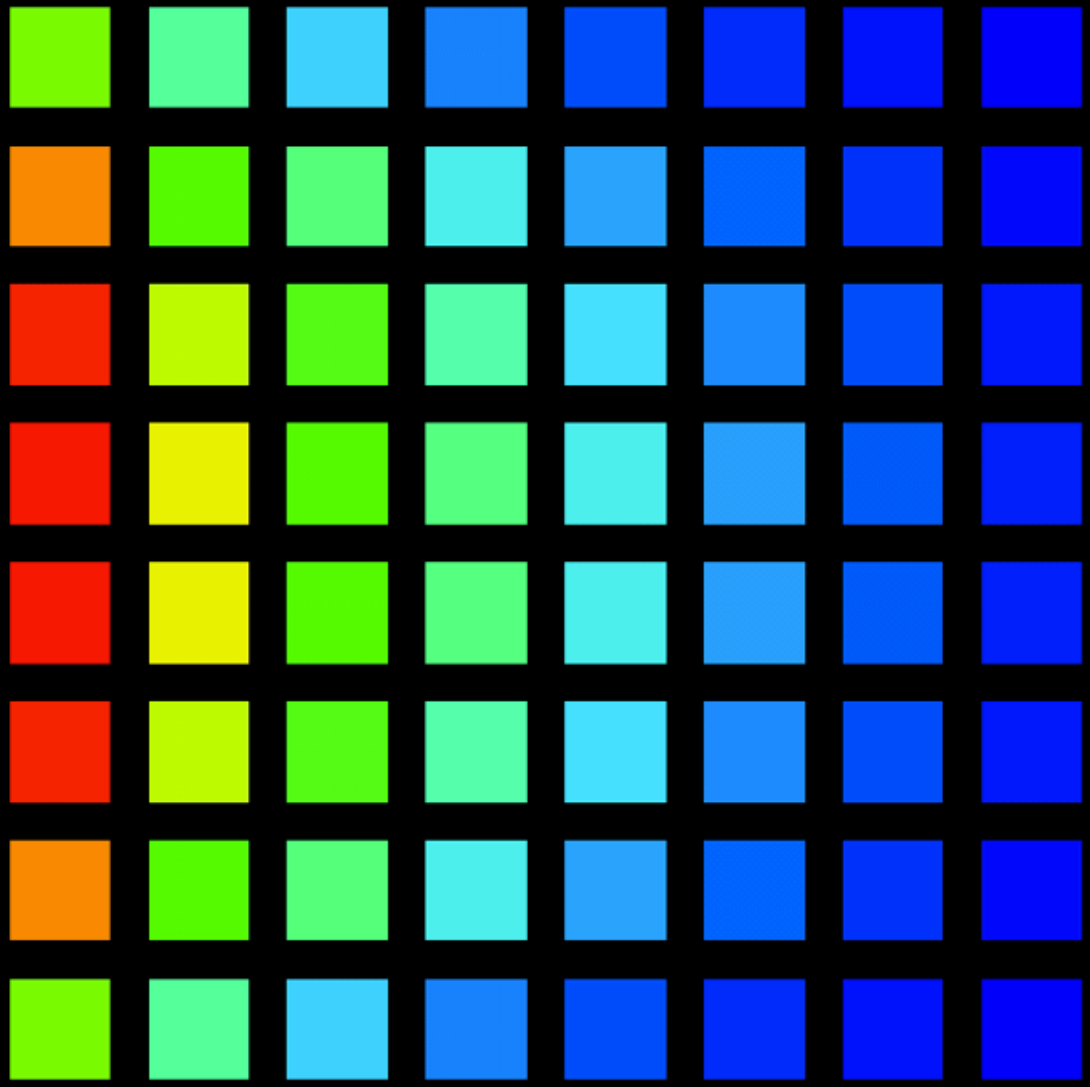
$$\begin{pmatrix}
-4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4
\end{pmatrix}$$

5X5 => 25X25

How previous page was written

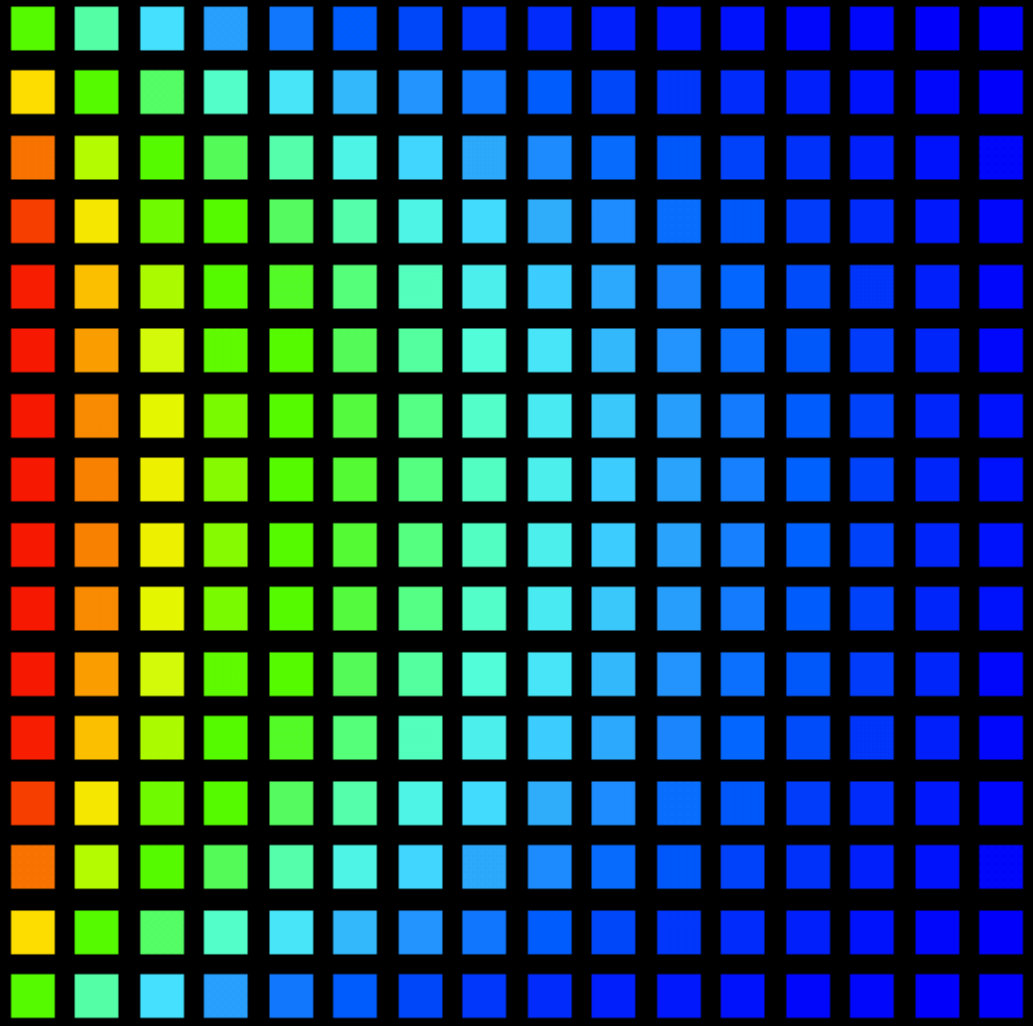
```
cout << endl<<"\n\nBuilidng a Laplacian:"<<endl;
for(int i=0;i<nx*ny;i++){
    for(int j=0;j<nx*ny;j++){
        if(fabs(pot[i][j])>0){
            cout << "{\\color{<red} " << pot[i][j] << " }";
        }
        else{
            cout <<  pot[i][j]  ;

        }   if(j<nx*ny-1) {
                    cout << " & ";
                }
                else {
                    cout << " ";
                }
        }
        if(i<nx*ny-1) cout << " \\" << "\\" <<endl;

}
cout <<endl;
}
```

8 X 8



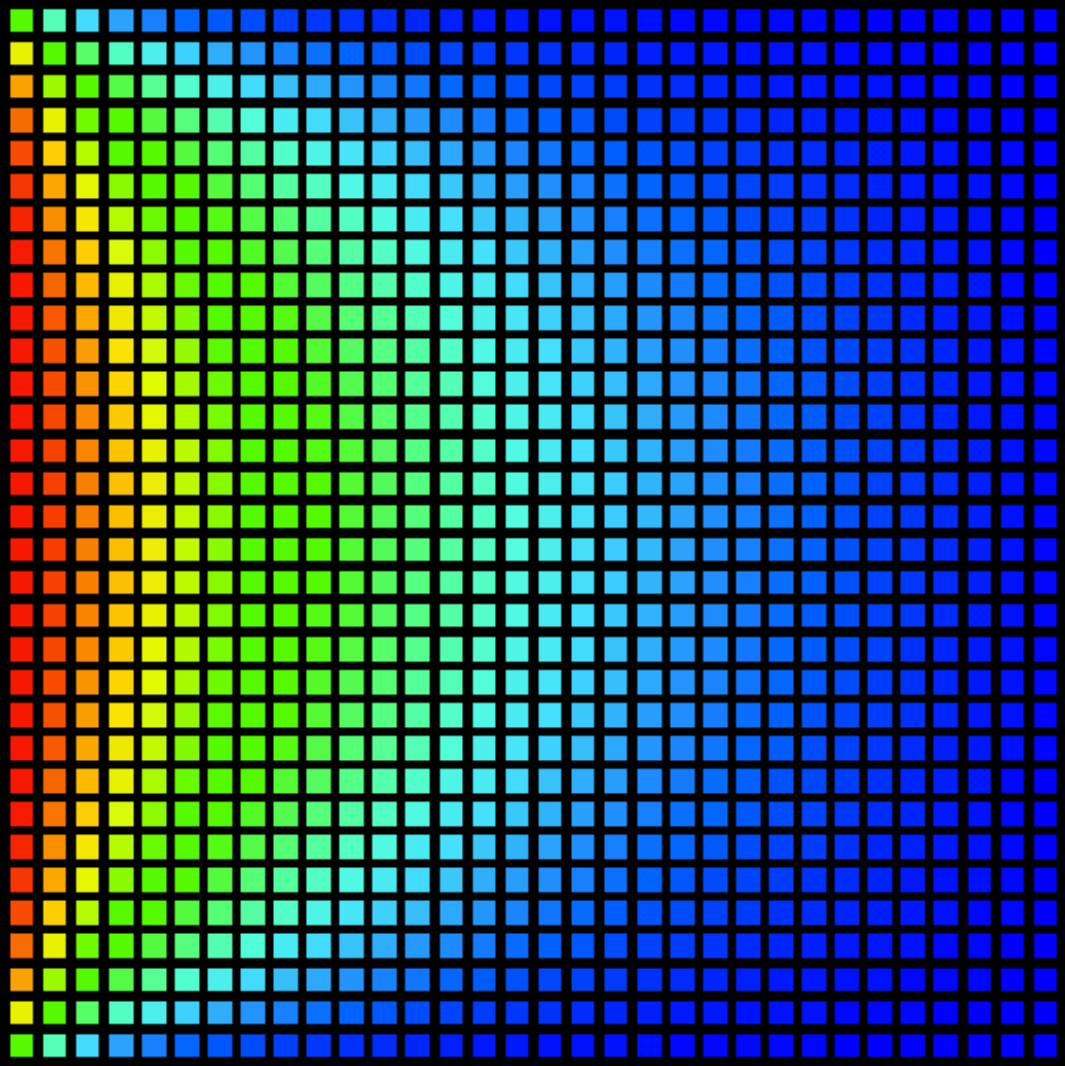
t= 2ms (LU), 4ms(sparse), 5ms(GJ)

16 X 16



t=66ms (LU), 19ms(sp), 251ms(GJ)

32X32



t=3966ms (LU), 167ms(sp), 17347 ms(GJ)

SUMMARY

- Solving Ax=b is a difficult task to perform numerically, especially for very large systems
- Choosing the "right solver" for a given problem with specific symmetry property is key
- Many modern research codes rely heavily on the use of optimal solvers
- In fact, the time used to run such solvers is usually the basis for supercomputer ranking

ADVANCED TOPICS

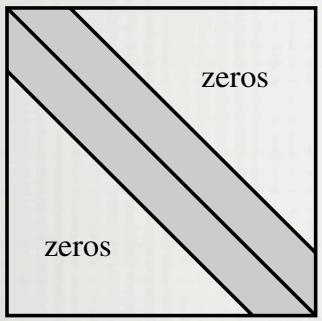
ADVANCED TOPIC

SINGULAR VALUE DECOMPOSITION

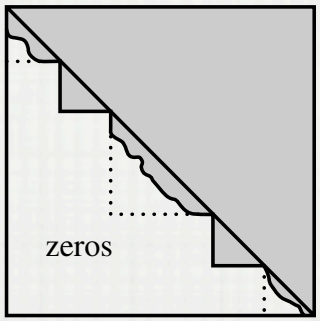
- When problems are too close to singularity
  - SVD will DIAGNOSE the problem and even provide a SOLUTION to it
  - SVD also solve linear LEAST-SQUARE problems (overdetermined problems)
- This is an advanced topic that could be covered in depth in a project

SPARSE MATRICES

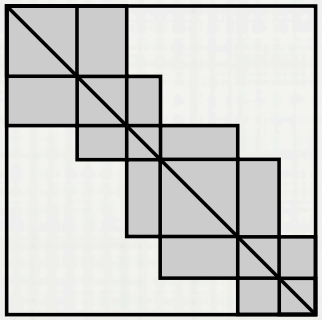
ADVANCED TOPIC



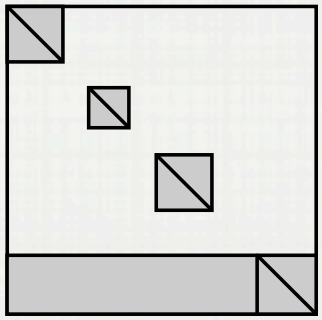
Band-diagonal



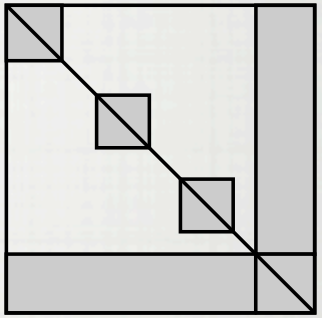
Block triangular



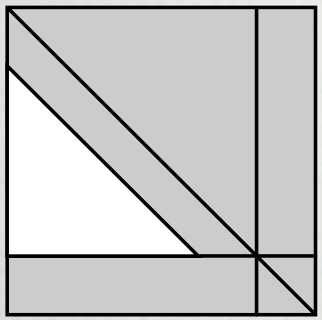
Block tridiagonal



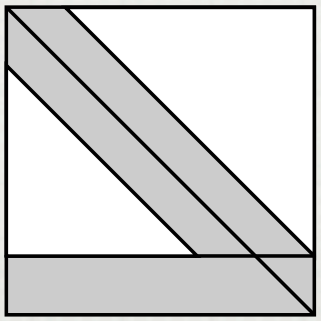
singly bordered block diagonal

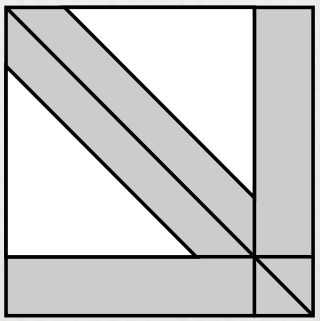


doubly bordered block diagonal

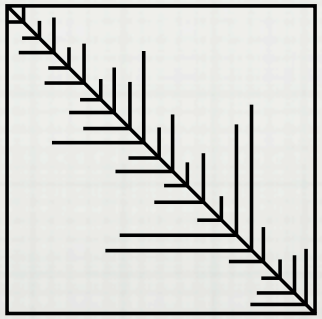


bordered band-triangular





singly and doubly bordered band-diagonal



OTHERS

INDEXED STORAGE: COMPRESSED COLUMN STORAGE

ADVANCED TOPIC

$$\begin{bmatrix} 3.0 & 0.0 & 1.0 & 2.0 & 0.0 \\ 0.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 7.0 & 5.0 & 9.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 6.0 & 5.0 \end{bmatrix}$$

index k	0	1	2	3	4	5	6	7	8
val[k]	3.0	4.0	7.0	1.0	5.0	2.0	9.0	6.0	5.0
row_ind[k]	0	1	2	0	2	0	2	4	4

index i	0	1	2	3	4	5
col_ptr[i]	0	1	3	5	8	9

ADVANCED TOPIC

HARWELL-BOEING FORMAT

- 3 vectors are used
  - val : non zero-values
  - row_ind: row index
  - col_ptr: column index ***
    - col_ptr[0]=0;
    - col_ptr[ncol+1]=nsparse
    - col_ptr[i] =>
      - the first nonzero in column i is in col_ptr[i]; the last is at col_ptr[i+1]-1
      - [[answers the question: *How many non-zero elements have we encountered in previous column?]]*

$$\begin{bmatrix} 3.0 & 0.0 & 1.0 & 2.0 & 0.0 \\ 0.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 7.0 & 5.0 & 9.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 6.0 & 5.0 \end{bmatrix}$$

index k	0	1	2	3	4	5	6	7	8
val[k]	3.0	4.0	7.0	1.0	5.0	2.0	9.0	6.0	5.0
row_ind[k]	0	1	2	0	2	0	2	4	4

index i	0	1	2	3	4	5
col_ptr[i]	0	1	3	5	8	9

$$\begin{bmatrix} 3.0 & 0.0 & 1.0 & 2.0 & 0.0 \\ 0.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 7.0 & 5.0 & 9.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 6.0 & 5.0 \end{bmatrix}$$

index k	0	1	2	3	4	5	6	7	8
val[k]	3.0	4.0	7.0	1.0	5.0	2.0	9.0	6.0	5.0
row_ind[k]	0	1	2	0	2	0	2	4	4

index i	0	1	2	3	4	5
col_ptr[i]	0	1	3	5	8	9

```
/* in this first example I explicitely define the variables
to be used later to build a sparse matrix */
    NRsparseMat A(5,5,9);

//the following should not be needed since they have been
given above
    A.nrows=5;
    A.ncols=5;
    A.nvals=9;

// colum index (see lecture notes)
    A.col_ptr[0]=0;
    A.col_ptr[1]=1;
    A.col_ptr[2]=3;
    A.col_ptr[3]=5;
    A.col_ptr[4]=8;
    A.col_ptr[5]=9;

// rows where sparse elements will be located
    A.row_ind[0]=0;
    A.row_ind[1]=1;
    A.row_ind[2]=2;
    A.row_ind[3]=0;
    A.row_ind[4]=2;
    A.row_ind[5]=0;
    A.row_ind[6]=2;
    A.row_ind[7]=4;
    A.row_ind[8]=3; //this is modified compared to NR book
par.2.7 since the matrix there is singular

//now the values themselves
    A.val[0] = 3.;
    A.val[1] = 4.;
    A.val[2] = 7.;
    A.val[3] = 1.;
    A.val[4] = 5.;
    A.val[5] = 2.;
    A.val[6] = 9.;
    A.val[7] = 6.;
    A.val[8] = 5.;
```

ADVANCED TOPIC

```cpp
#include <iostream>
#include "nr3.h"
#include "sort.h"
#include "sparse.h"

int main (int argc, char * const argv[]) {


    int nsparse=0;
    int i, j;
    double eps=0.00001;

    //example from lecture notes
    int n=5;
    MatDoub Full(n,n,0.0);

    Full[0][0]=3.0;
    Full[0][2]=1.0;
    Full[0][3]=2.0;
    Full[1][1]=4.0;
    Full[2][1]=7.0;
    Full[2][2]=5.0;
    Full[2][3]=9.0;
    Full[4][3]=6.0;
    Full[4][4]=5.0;

    //first we need to find out how many non-zero elements we have
    // we store that information in "nsparse"
    for (i= 0; i<n; i++){
          for (j = 0; j<n; j++){
               if(fabs((Full[i][j]))>eps) {
                    nsparse++;
               }
          }
    }
    //now we can create a sparse object
    NRsparseMat Sparse(n,n,nsparse);

    nsparse=0;
    for (j = 0; j<n; j++){
          if(j>0) {
               Sparse.col_ptr[j+1]=Sparse.col_ptr[j];
          }
          else
          {
               Sparse.col_ptr[0]=0;
          }
          for (i = 0; i<n; i++){
               if(fabs(Full[i][j])>eps) {
                    Sparse.row_ind[nsparse]=i;
                    Sparse.val[nsparse]=Full[i][j];
                    Sparse.col_ptr[j+1]++;
                    nsparse++;
               }
          }
    }

          COUT << "NSPARSE=" << NSPARSE << ENDL;
          COUT << "I= \T";
          FOR(I=0;I<NSPARSE;I++){
                COUT << I << " \T" ;
          }
          COUT << ENDL;
          COUT << "VAL= \T";

          FOR(I=0;I<NSPARSE;I++){
                COUT << SPARSE.VAL[I] << " \T";
          }
          COUT << ENDL;
          COUT << "ROW= \T";

          FOR(I=0;I<NSPARSE;I++){
                COUT << SPARSE.ROW_IND[I] << " \T";
          }
          COUT << ENDL;
          COUT << ENDL;
          COUT << "I= \T";
          FOR(I=0;I<N+1;I++){
                COUT << I << " \T";
          }
          COUT << ENDL;
          COUT << "COL= \T";

          FOR(I=0;I<N+1;I++){
                COUT << SPARSE.COL_PTR[I] << " \T";
          }
          COUT << ENDL;


          RETURN 0;
    }
```

```
vinces-MacBook-Pro:Debug vm2$ ./Resistors
Nsparse=9
i=      0       1       2       3       4       5       6       7       8
val=    3       4       7       1       5       2       9       6       5
row=    0       1       2       0       2       0       2       4       4

i=      0       1       2       3       4       5
col=    0       1       3       5       8       9
vinces-MacBook-Pro:Debug vm2$
```

ADVANCED TOPIC

IN PRACTICE: SPARSE.H

```
struct NRsparseMat
Sparse matrix data structure for compressed column storage.
{
    Int nrows;                          Number of rows.
    Int ncols;                          Number of columns.
    Int nvals;                          Maximum number of nonzeros.
    VecInt col_ptr;                     Pointers to start of columns.  Length is ncols+1.
    VecInt row_ind;                     Row indices of nonzeros.
    VecDoub val;                        Array of nonzero values.

    NRsparseMat();                                  Default constructor.
    NRsparseMat(Int m,Int n,Int nnvals);            Constructor.  Initializes vector to zero.
    VecDoub ax(const VecDoub &x) const;             Multiply A by a vector x[0..ncols-1].
    VecDoub atx(const VecDoub &x) const;            Multiply A^T by a vector x[0..nrows-1].
    NRsparseMat transpose() const;                  Form A^T.
};
```

SPARSE: EXAMPLE (SEE SPARSE.CPP)

ADVANCED TOPIC

```
#include "sparse.h"


#include "linbcg.h"
#include "asolve.h"

NRsparseMat A(5,5,9);

//here we define A

NRsparseLinbcg C(A);

VecDoub x(5,1.0);
VecDoub b(5,0.0);

//here we define x and b

Int itol=1;
Doub tol=1.0E-72;;
Int itmax=10000;
Int iter;
Doub err;

C.solve(b,x,itol,tol,itmax,iter,err);
```

SCALING

- Gaussj -> N^3
- LU -> $N^3/3$
- (Cholesky -> $N^3/6$)
- Sparse -> N^2