CornellEngineering
Electrical and Computer Engineering

ECE 4960: Computational and Software Engineering

Spring 2017

Note 4: Linear Algebra and Sparse Matrices

Reading Assignments:

1. Chaps. 4 and 5, D. Bindel and J. Goodman, *Principles of Scientific Computing*, 2009.
2. Chaps. 4 – 6, 13, B. Einarsson, Ed., *Accuracy and Reliability in Scientific Computing*, SIAM 2005. (Chaps. 4 and 13 mostly).

1. Class logistics

- Programming Assisgnment 2
- Reading review 2 (will be multiple choice questions on Blackboard)

2. Introduction

Although we live in a three-dimensional world with a unilateral time direction, we often have much more "degrees of freedom" (DOF) to formulate problems to be solved. For example, when we are modeling the Dow Jones index in the future market, each participating stock (about 10^7 variables of companies, funds and bills) can be treated as an independent variable. The interdependency among companies and its own forecast can make the problem nonlinear. Some environmental and engineering problems have DOF larger than 10^{12}! For example, a weather simulation of the area of 100km×100km×10km on 10m grids has 10^{11} variables[1]. An ECE example will be the microprocessor. When the output under interest is represented by the voltages as a function of time on the common data bus line (64 bits), the independent variable can be the voltages as a function of time of the internal nodes in the arithmetic-logic unit (ALU), which can typically be on the order of 10^7. The transistor is a nonlinear element with exponential to quadratic current-voltage relation, which makes the circuit description nonlinear.

Most mathematical and physical problems, after linearization, can be translated into problems involving vectors and matrices. For the circuit problem above, we can use Kerchhoff's laws to write down how the output bus nodes depend on the internal nodes. After linearization, this can be described by a matrix A of dimension $[10^7, 64]$ that transforms the 10^7 voltage node vector to the 64 output node vector. When we study how the internal node will change in time, we can use Kerchhoff's laws to express every RC branch to write a matrix A of dimension $[10^7, 10^7]$ that transforms the 10^7 voltage node vector at $t = t_0$ to the 10^7 voltage node vector at $t = t_0 + \Delta t$.

The above example shows an eminent problem in software modeling. With the full representation, a matrix of rank n will need $O(n^2)$ storage and $O(n^3)$ in the solution of $Ax = b$. The order of matrix solution computation will be explained later, but for now we notice that this superlinear dependence will already make many problems untreatable for present computers. For example, if we use 8 bytes (64 bits) to store a floating point of the voltage values in circuit example above, a matrix of $[10^7, 10^7]$ dimension will need about 10^{15} bytes (or 1,000TB) just to store it, which will require the resource of a gigantic cloud machine!

[1] In your everyday experience, you know that the local air concentration and temperature can vary significantly within 10m when a cold breeze blowing on your face in winter! Therefore, 10m grid is very coarse!

Fortunately, most matrices of practical problems are "sparse", i.e., most of their elements have zero values, and we just need to deal with a finite number of non-zero entries. If the problem is derived from circuits, if every node has on average 10 elements connected to it, the matrix storage is just $10n$ instead of n^2. If the problem is derived from finite-element or finite-difference of the partial differential equations, 3D formulation has an average of 7 – 21 elements per node (depending on the number of equations that need to be solved self consistently), which can still reduce the matrix storage in a very significant way!

Handling sparse matrices efficiently in terms of memory usage and computation is the heart of many computing problems, which can be a video game, a stock market analysis or an engineering simulation. We will not only treat matrix properties below, but always have the "sparse" structure in mind to ensure that we can treat larger practical problems.

Before we go on for more treatment and analysis, the vector and matrix software has been at the heart of many gaming, scientific and engineering problems. Free resources are abundant. Below is just a sample list for many established programs. You can regard the entire Matlab as a superb generic vector-matrix engine, but it is just not free!

- BLAS (basic linear algebra subprograms): Legacy Fortran and C functions that are highly optimized for CPU with limited cache space. Although the library had been created and maintained from the 1960's, it is still the workhorse for many scientific simulation software programs. See the web page at http://www.netlib.org/blas/.
- Sparselib++: Production of the National Institute of Standards and Technology. See the web page at: http://math.nist.gov/sparselib++/.
- Trilinos: A more recent effort with all modern language support. Actually the programs in this collection can serve as a reasonable demonstration of the software engineering principle we teach in this class. Trilinos is an international collaboration. In US, it is mostly maintained by the Sandia National Lab. See the web page at: https://trilinos.org/.

2.1 Sparse matrix representations

Although object oriented programming (OOP) teaches that you should only prescribe "methods" instead of "data structures" in the application procedural interface (API) of a library function, the most important consideration of sparse matrices is the efficient memory handling, so we will explain through the data structure of the sparse matrix. Notice that for the most basic operation of $Ax = b$, where A is of dimension $[m, n]$ and x of dimension m and b of dimension n, we will only need sparse representation for A. Unless A is severely degenerate (i.e., rank(A) << min(m, n)), b will not be sparse even though x is sparse.

For many applications, matrices with equal row and column ranks (assumed to be n here) are most common to operate on vector of rank n. We will use the nonsymetric matrix A of rank $n = 5$ below as an example[2]:

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}$$

[2] Following the example given in Sparselib++.

where rank(A) = n = 5, dim(A) = [5, 5], and we have 12 non-zero elements within the 25 matrix elements. We will follow the C/C++ convention of row index of 0…4 and column index of 0…4, instead of the Fortran convention of index starting from 1.

2.1.1 Coordinate storage

The most direct sparse matrix representation is the coodinate storage, where each nonzero element is stored in the structure of (rowInd, colInd, value). This should be used in the sparse matrix API as it is most intuitive. For the example above, we will have a data structure in Table 1:

Table 1. Coordinate storage of the matrix *A*.

value( )	1	2	3	4	5	6	7	8	9	10	11	12
rowInd( )	0	0	0	1	1	1	2	2	2	3	4	4
colInd( )	0	1	4	0	1	2	1	2	4	3	0	4

Notice that the entry in the coordinate storage can have random order and the content of the coordinate storage will not need to be sorted. This makes the addition and depletion of nonzero elements straightforward.

2.1.2 Compressed row or column storage

We notice that we can save a little more in storage space in the row and column indices. Hence it is more popular to use the compressed row or column storage as shown in Tables 2 and 3. Although the compressed row storage only gives slight storage saving, it is more efficient to trasverse the entire non-zero elements by using the compressed format with row or column pointers. For example, we notice in Table 1, the rowInd( ) has repeated values that equal to the number of the nonzero element in that row. Therefore, we can use the rowPtr( ) as an integer to denote the accumulative number of nonzero elements from row 0 to $n - 1$. The entry in compressed format is assumed to be sorted by rows or columns, as evident in Tables 2 and 3. This gives slight overhead in element insertion and deletion. However, for many practical problems, the nonzero entry has fixed positions in the matrix, although their values need constant updates. Therefore, this is not a serious overhead. For example of the circuit simulation above, the nonzero element is determined by the circuit topology (which nodes are connected to a given node by a circuit element, or aka, the circuit netlist), with the respective voltage value evolves in time.

Table 2. Compressed row storage of the matrix *A*.

value( )	1	2	3	4	5	6	7	8	9	10	11	12
rowPtr( )	0	3	6	9	10	12						
colInd( )	0	1	4	0	1	2	1	2	4	3	0	4

Similarly, we can use the colPtr( ) in the compressed column storage format, as shown in Table 3. The choice between the compressed row or column will be dictated by the most common way to trasverse the matrix. For example, in computing $Ax = b$, compressed row is preferred.

Table 3. Compressed column storage of the matrix *A*.

value( )	1	4	11	2	5	7	6	8	10	3	9	12
rowInd( )	0	1	4	0	1	2	1	2	3	0	2	4
colPtr( )	0	3	6	8	9	12						

The common utilities functions for sparse matrix access in addition to constructors and destructors include:

```
matrix*     createMatrix(n);  // intend to create a sparse matrix of rank n,
                              // but without reserving all storage.
                              // Memory scheme varies.
void        addElement(*matrix, rowInd, colInd, value);
void        deleteElement(*matrix, rowInd, colInd);
double      retrieveElement(*matrix, rowInd, colInd);
integer     rankMatrix(*matrix);     // return 5 for above
integer     countElementMatrix(*matrix); // return 12 for above
```

Hacker Practice 4.1:

Create the sparse matrix *A* in the compressed row format of Table 2 with three vectors: value[12]; rowPtr[6]; colInd[12]. Implement the function retrieveElement( ). Notice that the entry can be zero. Use the 5 by 5 printout to verify your program.

2.1.3 An example for module testing: the Wilkonson principle

The same operation implemented by different data structures can be used for modular testing. This can be performed with known or unknown ground truth. In the case of the unknown truth, we rely on the fact that the software bugs in different data structure implementations will render different answers. When the two implementations give exactly the same answer, the probability that both implementations are correct is very high.

This is a special case of the **Wilkinson principle**[3]: "The computed solution *x* is the EXACT solution of a nearby (perturbed) problem."[4] Or paraphrased here: "The computed solution *x* with incorrect implementation is the EXACT solution of the wrongly specified problem". Two implementations by two different data structures, if there are bugs, will likely NOT give the same EXACT solution!

Surely, if the common algorithm is wrong, then both implementations can give the SAME wrong answer, and therefore, this validation is useful, but incomplete. However, this Wilkinson technique remains critical in software validation (as well as in accounting).

We will use the sparse matrix computation for illustration in the practice below. Three common matrix operations will be implemented in the full matrix format and in the row-compressed format. The `rowPermute( )` function will switch the position of the *i*-th and *j*-th rows. The `rowScale( )` function will add the *i*-th row multiplied by a constant *a* to the *j*-th row. The `productAx( )` function will implement the product of matrix *A* and vector *x*. The return integer is reserved for the indicator of successful operations (e.x., 0 means no error or exception, 1 means unmatched rank of *A* and *x*, 2 means INF/NINF exception, 3 means NaN exception, etc. If you want to improve the code readability, use local `define` statement). You can implement the three methods in the full matrix and the sparse matrix

[3] J. H. Wilkinson, *The state of the art in error analysis*, NAG Newsletter, 2/85 (1985), pp. 5–28. (Invited lecture for the NAG 1984 Annual General Meeting.)
[4] The Wilkinson principle is probably more famous (and applicable) in the overall precision testing. When two computation implementations (no software bug in either) give similar but different results, each answer can be treated as an exact solution of the respective perturbed problem. We will illustrate that after introducing matrix pivoting.

formats and then make element-by-element of the final results after random order of the three operations! You do not have to know the ground truth. You can easily automate the check process! If your matrix manipulation has software bugs, it is likely that you can find out in the simple test.

The three basic operations can actually implement the direct solver. We will see very soon in the next section. Also notice that different tests cover different domains of validation. Checking $\|Ax - b\|$ is effective for the solution of $Ax = b$, but did little to check `productAx( )` except the norm calculation.

Hacker Practice 4.2:

Use the row-compressed storage and the full-matrix representations to implement the vector product:

```
int productAx(matrix* A, vec* x, vec* b);
// Compute the product of Ax = b
```

Write a test function to compare the resulting matrix and vector (all elements) for validation. For the full-matrix representation, feel free to use built-in utility functions. Use the previous A matrix with rank 5, and $x = (5, 4, 3, 2, 1)^T$.

2.2 Problem definition for solution of $Ax = b$

The basic linear algebra review is provided in Bindel Chap. 4 including definitions of vectors, vector space, norm properties, orthogonality, base decomposition, matrix, row mutation, matrix multiplication, determinant and inversion, and we will not repeat here. We will focus mostly on the applications that demands uses of matrix solution, while leave the other vector-matrix operations to you (when the matrix is in the sparse format as well). The mathematical rigor will take quite some time, and standard proof abounds in elementary books of linear algebra. We will take a more intuitive view without the rigorous proof for the purpose of our class.

The multi-variate problems, whether the independent variables come from geometry, topology, circuits, stock prices, test scores, naturally map into vectors and matrices. If we “visualize” vectors in the geometrical spaces, matrices perfrom transformation of one vector to the other, with scaling (diagonals), rotation, and skewing. Matrices can also be seen as collection of column or row vectors. A lot of the real-world problems map into the following matrix-vector forms:

$$Ax = b \qquad (1)$$

$$\text{Minimization of } \|Ax - b\|_2 \qquad (2)$$

$$Ar = \lambda r \qquad (3)$$

where λ is a scalar (eigenvalues), x, b, and r are vectors, and A is the matrix of interest. Eq. (2) is another form of Eq. (1) when precision, nonlinearity and uncertainties are directly in consideration, and Eq. (3), the eigenvalue/eigenvector problem contains all information of matrix A, although it is often much more expensive to solve. The waves in eletromagnetics, fluidics and quantum mechanics directly map into problems in the form of Eq. (3). The three problems share many of the same properties of A. For example, if we know the solutions of Eq. (3), i.e., we know all eigenvectors r_i with the corresponding eigenvalue λ_i, then we know a precise way to transform A into a diagonal or Jordon-block matrix. For a matrix of rank n, there would be n pairs of eigenvectors and eigenvalues. If we use the eigenvector as the column vector to compose a $n\times n$ matrix R, we can have

$$R = \begin{bmatrix} r_1 & r_2 & \dots & r_n \end{bmatrix} \qquad (4)$$

$$AR = R\Lambda \qquad (5)$$

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \dots & 0 \\ 0 & 0 & \lambda_n \end{bmatrix} \qquad (6)$$

The $Ax = b$ problem can be viewed by decomposing x into the linear superposition of r (because r will be a complete base set if A is nonsingular, this decomposition is always possible) with unknown coefficients c_i:

$$x = c_1 r_1 + c_2 r_2 + \dots + c_n r_n \qquad (7)$$

Notice that when we apply A to x, as $Ar_i = \lambda_i r_i$ in the eigenvalue problem:

$$Ax = A\cdot(c_1 r_1 + c_2 r_2 + \dots + c_n r_n) = c_1\lambda_1 r_1 + c_2\lambda_2 r_2 + \dots + c_n\lambda_n r_n = b \qquad (8)$$

Therefore, if we know how to decompose b into superpoistion of r_i, we have the solution x for $Ax = b$! Eigenvalue problems provide the full matrix solution, but it is often too computationally expensive to find all eigenvalues and eigenvectors. Its main applications are in wave equations, electromagnetic, fluidic, or quantum mechanics. We will discuss a few facts on the eigenvalue problems after establishing this equivalency, and will develop our intuition mostly with the $Ax = b$ problem.

(**Exercise**) For A as an identify matrix I, find the corresponding eigenvalues and eigenvectors and see the validity of Eq. (5). Observe the effect of row permutation as well.

3. Computing concerns for the solution of *Ax = b*

We will first use two examples to see how precision and conditioning are important in matrix solution of $Ax = b$.

Example 1: Ill conditioning for $Ax = b$: Give $\begin{pmatrix} 100 & 99 \\ 99 & 98 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 199 \\ 197 \end{pmatrix}$. The solution is exact at $x = y = 1$.

A small perturbation to A: $\begin{pmatrix} 100 & 99 \\ 99 & 98.009 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 199 \\ 197 \end{pmatrix}$ will give solution to be $x = -7.91$ and $y = 10$.

The small perturbation of A causes huge change in the solution: definition of ill conditioning previously, i.e., error in input is greatly amplified in the output. The origin of this amplification can be understood graphically. The two lines are nearly identical!!! A slight change in the slope will move the intersection over large distance.

Example 2: Ill conditioning for eigenvalue problems: Give $A = \begin{pmatrix} 10 & 100 & 0 & 0 \\ 0 & 10 & 100 & 0 \\ 0 & 0 & 10 & 100 \\ 0 & 0 & 0 & 10 \end{pmatrix}$. Due to the similar structure to the Jordon blocks, the eigenvalues are $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 10$. If we perturb the Jordon structure just a little bit, and give $A = \begin{pmatrix} 10 & 100 & 0 & 0 \\ 0 & 10 & 100 & 0 \\ 0 & 0 & 10 & 100 \\ 10^{-6} & 0 & 0 & 10 \end{pmatrix}$: the eigenvalues are shifted to $\lambda_1 = 11$, $\lambda_2 = 10+i$, $\lambda_3 = 10 - i$,and $\lambda_4 = 9$.

Hacker Practice 4.3:

Use any of your matrix solver for $\begin{pmatrix} 100 & 99 \\ 99 & 98.01 - e \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 199 \\ 197 \end{pmatrix}$ with $e = 10^{-2}, 10^{-3}, \ldots, 10^{-9}$.

Print out the value of (x, y) and the second norm of the residual vector:

$$\left\| \begin{pmatrix} 100 & 99 \\ 99 & 98.01 - e \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} 199 \\ 197 \end{pmatrix} \right\|_2 .$$

Further reading: My favorite reading treating this matrix solver topis is the classic book: R. L. Burden and J. D. Faires, *Numerical Analysis, 3rd Ed*, PWS, 1985, where rigorous proofs are accompanied by immediate numerical examples. If your next interview is expected to have many linear algebra questions, do read Chap. 6 for direct solvers and Chap. 8 for iterative solvers.

3.1 Intuitive look into Gaussian elimination and LU decomposition

You probably have learned Gaussian elimination for $Ax = b$ since Grade 7, and let's see how it accomplishes simultaneously matrix LU decomposition, and the implications to sparse matrix fill-in and pivoting choices. In the sample 3×3 matrix in Table 1, we opt to use a full matrix, so that you can see clearly the most operatons, especially in the choice of "**pivots**", and the corresponding **triangular matrix** that accomplishes the pivoting and elimination process.

Before the Gaussian elimination procedure, let's first make some observation on the system of equations in Table 1 for future preparation of generalized pivot selection, as we will use the most common choice of diagonal pivoting in the first example. The row number of the matrix is the equation number, which is arbitrarily assigned. Any **row permutation** will NOT change the solution, as long as the right hand side is changed consistently. The column number of the matrix is the variable sequence, which is again arbitrarily assigned. Any **column permutation** will just change the order of the elements in the solution vector x. In the sparse matrix storage formats (coodinate, row-compressed or column compressed), the vectors and pointers may change, but the total number of nonzero elements will not change by the permutation operation.

Table 1. Correspondence of Gaussian elimination and LU decomposition

Gaussian elimination	**LU decomposition**
$4x_1 + 4x_2 + 2x_3 = 2$ $4x_1 + 5x_2 + 3x_3 = 3$ $2x_1 + 3x_2 + 3x_3 = 5$	$A = \begin{pmatrix} 4 & 4 & 2 \\ 4 & 5 & 3 \\ 2 & 3 & 3 \end{pmatrix}; \quad b = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix};$
Eliminate x_1 from 2nd and 3rd rows (a_{11} is called the **pivot** in this operation): $4x_1 + 4x_2 + 2x_3 = 2$ $x_2 + x_3 = 1$ $x_2 + 2x_3 = 4$	$M_1 = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -0.5 & 0 & 1 \end{pmatrix}; \; M_1A = \begin{pmatrix} 4 & 4 & 2 \\ 0 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}; \; M_1b = \begin{pmatrix} 2 \\ 1 \\ 4 \end{pmatrix}$
Eliminate x_2 from 3rd row (a_{22} is the pivot in this operation): $4x_1 + 4x_2 + 2x_3 = 2$ $x_2 + x_3 = 1$ $x_3 = 3$	$M_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}; \; M_2M_1A = \begin{pmatrix} 4 & 4 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix};$ $M_2M_1b = \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}$

$$M_2M_1Ax = M_2M_1b; A = LU$$

$U=M_2M_1A$ is an upper triangular matrix

M_1, M_2 and $L = M_1^{-1}M_2^{-1}$ are lower triangular matrices

Notice how the Gaussian elimination step chooses the "pivot", and how the lower triangular matrices M_i corresponds to the step of making the resulting $\prod_i M_i A$ to be an upper triangular matrix.

All upper U and lower L triangular matrices with **unit diagonals** have product and inversion rules, i.e., the product of upper triangular matrices will be an upper triangular matrix; the inverse of an upper triangular matrix will be an upper triangular matrix. The triangular matrix can be readily solved by backward substitution with $O(n^2)$ computational cost.

Gaussian elimination or the equivalent LU decomposition can be viewed to provide a framework (the sequential order) for matrix element manipulation. The sequence of operations only depends on the matrix structure (the location of the nonzero elements), instead of the actual element values. However, the actual element values may affect precision of calculation (we will discuss later about how to choose the proper pivot in each operation)!

3.2 First objective: Minimal fill-in for sparse matrices

The sparse matrix structure can change during the Gaussian elimination, or equivalently, LU decomposition. For both memory and computing (numerical LU decomposition and backward substitution have computational cost proportional to the number of non-zeroes in the sparse matrix), we hope to keep these "fill-ins" to a minimal by reordering the rows or columns (i.e., row or column permutation). We can see from the previous example that there would be three fill-ins after the operation that makes the first column all zero for elements below the chosen pivot:

$$\begin{pmatrix} X & X & X & 0 \\ X & X & 0 & 0 \\ 0 & 0 & X & X \\ X & 0 & 0 & X \end{pmatrix} \Rightarrow \begin{pmatrix} X & X & X & 0 \\ 0 & X & \otimes & 0 \\ 0 & 0 & X & X \\ 0 & \otimes & \otimes & X \end{pmatrix}$$

However, if we permute the first and second rows (which has more zeroes, or its non-zero elements aligning with many lower rows), then there would be only one fill-in during the operation of a_{11} pivoting:

$$\begin{pmatrix} X & X & 0 & 0 \\ X & X & X & 0 \\ 0 & 0 & X & X \\ X & 0 & 0 & X \end{pmatrix} \Rightarrow \begin{pmatrix} X & X & 0 & 0 \\ 0 & X & X & 0 \\ 0 & 0 & X & X \\ 0 & \otimes & 0 & X \end{pmatrix}$$

Row permutation is the change of the order of the objective functions, which does not change the property of A except that the determinant is multiplied by –1. Row permutation does change the **sparse structure** significantly (but not the total number of nonzero elements) and can affect the number of fill-ins for subsequent operations and hence the memory usage and total computational time.

Notice that the LU decomposition can be a symbolic step, i.e., the minimal fill-in can be figured out without knowing the exact numerical values of the elements in A, but just the sparse structure. For example, in circuit simulation, once the circuit topology is fixed, this sparse structure will not change, and all following DC and transient simulation can share the first *symbolic LU factorization*. For finite-difference and finite-element simulation, once the gridding and discretization of the geometry are fixed, we can perform one symbolic LU factorization and use it to organize simulation in all following stages. As symbolic LU factorization is the most expensive step O(n^3) with all other numerical computation on O(n^2), this represents significant saving in direct sparse matrix solver which has no iteration convergence concerns.

Similarly, we can use column permutation to tidy up the matrix so that the pivot (the element chosen to zero out the lower elements in the same column) is always on the diagonal. We just need to keep the solution vector indices correspondingly.

Therefore, for problems originated from circuits or partial-differential equations, we often choose direct sparse solver in three major computational steps whenever possible for the hardware memory allowance, not only for stability, but also for computational efficiency:

1. **Symbolic LU factorization** to determine the order of computing for minimal fill-in and best anticipated pivoting (described in the next section): O(n^3)
2. **Numerical calculation of *L* and *U* matrices**: O(nz), where nz is the total number of nonzero elements during the LU factorization.
3. **Backward substitution** (one for L and one for U) to obtain the solution: O($n \times nz$).

After successful LU factorization of A, we can see now the problem is transformed to:

$$A = LU;\ Ax = b \rightarrow LUx = b;\ Ux = L^{-1}b \text{ or } x = U^{-1}L^{-1}b. \qquad (9)$$

The inverse of a triangular matrix is much easier to obtain, and the matrix solution has become just backward substitution!

(**Exercise**) From the degree of freedom (DoF) point of view, how many elements are in *A*? How many elements are in *L* and *U*? If the sum of DoF of *L* and *U* is larger than that of *A*, what does that imply?

The matrix *A* has many different representations for LU decomposition, as just observed from the degree of freedom. Notice that we have accomplished a lot by LU factorization, as many matrix properties of *L* and *U* can be easily calculated:

1. Make $Ux = v$. We can solve $Lv = b$ and $Ux = v$ by backward substitution in $O(n^2)$ time.
2. L^{-1} and U^{-1} are also triangular matrices, and can be readily computed (however, they may have a lot of fill-ins, so this is not typically done in view of memory usage).
3. The **eigenvalues** of *L* and *U* are their respective diagonals (hence the "**spectral radii**" of *L* and *U*, which are the largest magnitude of the diagonals, are known).
4. The matrix determinant of *L* and *U* can be easily determined:

$$Det(L) = \prod_{i=1}^{n} L_{ii}; \qquad Det(U) = \prod_{i=1}^{n} U_{ii}$$

There are three popular choices to obtain a specific solution for LU decomposition by applying more constraints. The method in Sec. 3.1 has the *L* matrix to have unit diagonal, which is called the unit lower trangular matrix. Notice that unit lower triangular matrices obey product and inversion rules, i.e., the product of two unit *L* triangular matrices will be a unit *L* triangular matrix, and the inverse of the unit *L* triangular matrix (as seen clearly by the Cramer's rule and cofactor matrix in the matrix inversion) will be a unit *L* triangular matrix. Constraining *L* to have a unit diagonal is called the **Doolittle** LU factorization. Similarly we can constrain *U* to have a unit diagonal, which is called **Crout** LU factorization.

Another popular choice is the **Choleski** LU factorization where we constrain $L_{ii} = U_{ii}$. With the high precision in floating points, the choice among Doolittle, Crout and Choleski is often not very critical for the solution accuracy.

The following statements are equivalent to describe the square matrix with rank *n*:

1. *A* has a rull rank of *n*.
2. *A* can be inverted (A^{-1} exists).
3. *A* is non-singular or non-degenerate.
4. *A* can have successful LU factorization with no zero eigenvalues in *L* and *U* (precision will be subject to the matrix condition number).
5. The matrix determinant is not zero: $\text{Det(A)} \neq 0$.
6. The eigenvalues of *A* are not zero (but they can have the repeated nonzero elements), and *A* can be diagonalized into the Jordon block structure.

Hacker Practice 4.4:

Use the full matrix format and row permutation (easier to implement in a short time, and a good check for your sparse matrixlater on), perform the minimal fill-in algorithm for choosing the sequence of pivoting to solve:

$$Ax=b; \quad \begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$$

Use the backward substitution to check your answer. For an interesting process, change your data structure to row-compressed format. However, your choices of pivoting should be the same, and your LU matrix identical.

3.3 Second objective: choosing pivots to control the round-off errors

Round-off errors can be amplified in the Gaussian elimination process. An example is shown below, assuming only four digits of precision:

$$\begin{cases} 0.003000x_1 + 59.14x_2 = 59.17 \\ 5.291x_1 - 6.130x_2 = 46.78 \end{cases} \tag{10}$$

The exact solution is x_1 = 10.00 and x_2 = 1.000. If we perform Gaussian elimination on x_1, we will find that the precision of x_2 will be truncated out, and results in x_1 = –10.00 and x_2 = 1.001!!! A careful observation will find that the precision is lost when $59.14\times5.291/0.003000 = 1.043\times10^5$ is much larger than 6.130 by 4 orders of magnitude, and hence adding 6.130 to 1.043×10^5 with only 4 digits of precision will lose the important precision. This round-off error can be remedied by change the "pivoting" of the first Gaussian elimination step to x_2, instead of x_1, (i.e., we perform a column permutation) where x_2 is much larger in magnitude, and hence will eliminate the other rows more precisely under finite precision.

Choice of the appropriate element for **pivoting** can be guided by highest **precision preservation**, instead of by **minimal fill-ins** for the sparse structure. Actually these two criteria can be in conflict: the pivoting choice for precision preservation and the one for minimal fill-in can be conflicting and can cause severe degradation for the other goal! If only column vectors are considered when we choose the pivot element, it is called **partial** or **column pivoting**. If we consider both column and row permutations, it is then called **total**, **full** or **maximum pivoting**. The search of the best pivoting for precision preservation is an O(n^3) operation, similar to the symbolic LU factorization for minimal fill-in. However, we need to know the numerical elements of each element to choose pivoting for precision preservation.

To emphasize again, choice of minimal fill-in and maximum pivoting has conflicts in general, has a trade-off in controlling the round-off error and the computing efficiency. The tradeoff is explained in the next section. Notice that we have NOT perturbed the values of the matrix element yet or considered how degenerate the matrix is (i.e., is det(A) $\rightarrow$ 0), so pivoting will control round-off errors but will NOT affect the matrix conditioning, which can be inherent in the magnification of errors during solving $Ax = b$.

3.4 Practical heuristics for sparse matrix with some properties

The heuristic to resolve the choices between minimum fill-in and maximum pivoting is called **pivot-on-the-fly**, where a pivoting tolerance is set to invoke alternative pivoting choices from minimal fill-ins and take sacrifice of the increase in non-zero fill-in. Pivot-on-the-fly is expensive too, as choosing the pivot and changing the computing order in LU decomposition are both O(n^3) operations. Fortunately for many practical matrices, we do not need maximum pivoting to control the round-off errors. The first category is called **diagonally dominant**, which can be achieved through row and column permutation:

$$|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \text{ for all } i \qquad (11)$$

Notice that the diagonal dominant matrix has better numerical properties in controlling round-off erros, but cannot achieve minimal fill-ins.

The second category is called **positive definite**:

$$x^t A x > 0 \text{ for all } x \qquad (12)$$

Surely, a test of the complete base functions will be sufficient due to the linear superpositioning principle. When A is an identity matrix, $x^t A x$ will be the second norm of the vector x. A matrix that is not diagonally dominant can not be positive definite. For example,

$$\begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \end{bmatrix} = -2 < 0$$

But the row mutation $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ is positive definite. If a matrix is either diagonally dominant or positive definite, then the round-off error will not seriously accumulate. A practical heustic for pivot-on-the-fly is therefore to perform symbolic LU factorization according to minimal fillin, and check if the condition:

$$|a_{ii}| > S_{th} \cdot \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \text{ for all } i \qquad (13)$$

where S_{th} is the scaling threshold for tolerance. A typical value for S_{th} in double-precision will be 10^{-1} – 10^{-5}. A larger S_{th} will guarantee the condition that diagonal dominance will not be seriously violated. As the minimal fill-in algorithm is performed at the symbolic step when the value of a_{ij} is not yet known, the pivot-on-the-fly algorithm will be triggered to alter the LU factorization operation with different pivoting choices only when Eq. (13) is violated. We often set S_{th} to 10^{-1} when the memory and computational time constraints are not stringent, and to 10^{-5} when we hope to keep most of the minimal fill-in pivoting choices to save memory and computational time.

3.5 Vector and matrix norms

To estimate the conditioning of vectors and matrices, we often use “norms” to sum up the contribution from each element. We will first give the formal definitions of the “norm” for vectors and matrices. There are three general properties for any norm definition for vectors:

1. $\|x\|$ is positive, and only equal to zero when all $x_i = 0$.
2. $\|ax\| = |a| \cdot \|x\|$ where a is any complex scalar.
3. $\|x+y\| \le \|x\| + \|y\|$ (this is also referred as the “triangular rule”)

There are a few familiar vector norms (1st, 2nd and infinite) which are denoted as subscripts outside the norm brackets:

1. First norm: $\|x\|_1 = \sum_{i=1}^{n} |x_i|$.
2. Second norm: $\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$.
3. Infinite norm: $\|x\|_\infty = \max_{i=1,n} |x_i|$

The matrix norm can use similar definitions, such as the Frobenius norm:

$$\|A\|_{F2} = \sqrt{\sum_{i,j=1}^{n} a_{ij}^2} \qquad (14)$$

However, this norm does not indicate the matrix properties in a useful manner. A better and more general definition is to view *A* as a transformation for the vector *x* that it applies to:

$$\|A\| = \max_{\forall x \neq 0} \frac{\|Ax\|}{\|x\|} \qquad (15)$$

Equation (15) measures the maximum “stretching” magnitude of *x* by *A*, and can be applied to any vector norms. This can be directly related to the geometrical properties of vectors, and all of the vector norm criteria can be maintained.

For matrix with real entities, a fast and useful way to estimate the matrix norm can be written as:

$$\|A\|_1 \le \max_k \sum_j |a_{jk}| \qquad \text{Maximum column vector sum} \qquad (16)$$

$$\|A\|_\infty \le \max_j \sum_k |a_{jk}| \qquad \text{Maximum row vector sum} \qquad (17)$$

This can be understood from the triangular rule of the vector sum, where the maximum stretching of a vector will be smaller than the maximum stretching of each of its components.

Hacker Practice 4.5:

Calculate the upper bounds of $\|A\|_1$ and $\|A\|_\infty$ in the full-matrix and sparse-matrix formats. Preserve the function, as we will use the results later.

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}$$

3.6 Choice of pivoting and matrix conditioning

After defining the vector and matrix norms, we have enough tools to represent the matrix conditioning. From the original definition, we are interested in how perturbation in matrix elements of a_{ij} will affect the solution of $Ax = b$. For nonsingular matrices, we know that A^{-1} exists and $AA^{-1} = I$, although often it is hard to compute the inverse directly[5]. A^{-1} is most often not sparse at all even if A has high sparsity. Taking a full differentiation operation (denoted by Δ) on $Ax = b$, we obtain:

$$\Delta A \cdot x + A\Delta x = 0 \qquad (18)$$

$$\|\Delta x\| \le \|A^{-1}\| \cdot \|\Delta A\| \cdot \|x\| \qquad (19)$$

The inequality comes again from the triangular rule. By arranging the terms, we obtain the matrix conditioning κ_A as:

$$\frac{\|\Delta x\|}{\|x\|} \le \|A^{-1}\| \cdot \|A\| \cdot \frac{\|\Delta A\|}{\|A\|} \qquad (20)$$

where the matrix condition $\kappa_A \equiv \|A^{-1}\| \cdot \|A\|$. The result is simple but not surprising, as κ_A describes the maximum stretching of A on $A^{-1}b$ for all possible b! The norm of b does not enter Eq. (20) directly, but will indeed affect the numerical values and variations in $\|\Delta x\|$, which has been ignored when we only look for variations in A as the only variation source and treat b as a constant.

[5] As a review for obtaining the matrix inverse, the Cramer's rule and the cofactor matrix $subA_{ij}$ are needed. $subA$ is the matrix reduced from A by deleting the i-th row and j-th column. $A_{ij}^{-1} = \frac{1}{\det(A)}\left(\det\left(subA_{ij}\right)\right)$. For example, if

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \Rightarrow \quad A^{-1} = \frac{1}{(ad-bc)}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

Equation (20) is often only theoretically helpful, as $\left\|A^{-1}\right\|$ is usually beyond the reach of most practical computational applications. Simplified estimate of $\left\|A^{-1}\right\|$ is only possible[6] after we obtain *L* and *U* matrices with LU factorization, but we cannot get a beforehand knowledge. Therefore, Eq. (20) is useful to check the validity after we have almost finished solving the question, but not an evaluation before we solve it.

A popular estimate of A^{-1} is the inverse of the diagonal element of *A*, i.e., $A^{-1}=\begin{bmatrix} a_{11}^{-1} & \dots & 0 \\ \dots & \dots & \dots \\ 0 & \dots & a_{nn}^{-1} \end{bmatrix}$, which just "normalize" the diagonal element of *A* to be 1. This is simple but reasonably useful. This will transform the problems in Eq. (10) to (keeping only 4 digits of precision):

$$\begin{cases} x_1 + 19710 x_2 = 19720 \\ -0.8631 x_1 + x_2 = 7.631 \end{cases}$$

If we use the diagonal dominance to choose the pivoting order, we will perform a column permutation:

$$\begin{bmatrix} 19710 & 1 \\ 1 & -0.8631 \end{bmatrix} \cdot \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 19720 \\ 7.631 \end{bmatrix}$$

or after diagonal normalization:

$$\begin{bmatrix} 1 & 5.074\times 10^{-5} \\ -11.59 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1.001 \\ -8.841 \end{bmatrix}$$

This is the popular way to prepare or condition the matrix before LU factorization. The matrix conditioning can still be observed by the relative magnitude of the off-diagonal elements.

4. Iterative matrix solvers

There are specially dedicated courses on iterative matrix solvers, and for VERY large problems where any amount of fill-in cannot be tolerated, we will NEED to use iterative solvers. I will leave the detailed treatment to your future classes, but only introduce the very important principle to compare with the direct solver. For many circuit simulation and finite-element based simulation, direct solvers are more reliable, and often faster due to the symbolic LU factorization executed only once especially when the memory is not very stringent. One more reason that I shorten the introduction for iterative solvers.

For $Ax = b$, iterative solvers mean that we can find an initial guess $x^{(0)}$, a *T* matrix and a constant *C* vector that the following operation can be performed:

$$x^{(k)} = Tx^{(k-1)} + C \qquad (21)$$

[6] The inverse of *L* and *U* are still triangular matrices, and the spectral radius (largest eigenvalue) is on the diagonal.

$$\frac{\left\| b - Ax^{(k)} \right\|}{\left\| b - Ax^{(k-1)} \right\|} < 1 \qquad (22)$$

Equation (22) is the convergence criteria for successive application of Eq. (21) where the residual vector is defined as $b - Ax^{(k)}$ at the k-th iteration. If T and C also depend on $x^{(k-1)}$, then the problem is nonlinear, which we will encounter in the next Chapter. We will restrict ourselves now for T and C that are independent of x.

4.1 Jacobi and Gauss-Seidel iterative methods

What is a good choice for T? The Jacobi iteration for a four-variable example below gives the first clue:

$$\begin{aligned} 10x_1 - x_2 + 2x_3 \qquad &= 6 \\ -x_1 + 11x_2 - x_3 + 3x_4 &= 25 \\ 2x_1 - x_2 + 10x_3 - x_4 &= -11 \\ 3x_2 \quad - x_3 + 8x_4 &= 15 \end{aligned} \qquad \begin{pmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 6 \\ 25 \\ -11 \\ 15 \end{pmatrix} \qquad (23)$$

We can express the equation set with a diagonal preconditioning and moving the off-diagonal elements to the right hand side as:

$$\begin{aligned} x_1 &= \qquad \frac{1}{10}x_2 - \frac{1}{5}x_3 \qquad + \frac{3}{5} \\ x_2 &= \frac{1}{11}x_1 \qquad + \frac{1}{11}x_3 - \frac{3}{11}x_4 + \frac{25}{11} \\ x_3 &= -\frac{1}{5}x_1 + \frac{1}{10}x_2 \qquad + \frac{1}{10}x_4 - \frac{11}{10} \\ x_4 &= \qquad -\frac{3}{8}x_2 \quad + \frac{1}{8}x_3 \qquad + \frac{15}{8} \end{aligned} \qquad T = \begin{pmatrix} 0 & \frac{1}{10} & -\frac{1}{5} & 0 \\ \frac{1}{11} & 0 & \frac{1}{11} & -\frac{3}{11} \\ \frac{1}{5} & \frac{1}{10} & 0 & \frac{1}{10} \\ 0 & -\frac{3}{8} & \frac{1}{8} & 0 \end{pmatrix} \qquad (24)$$

We can see symbolically by decomposing A into $D - L - U$ where D is the diagonal of A, $-L$ is the lower triangular part of A (without diagonal) and $-U$ is the upper triangular part of A (without diagonal):

$$(D - L - U)x = b \quad \Rightarrow \quad Dx = (L + U)x + b \qquad (25)$$

$$x = D^{-1}(L + U)x + D^{-1}b \quad \Rightarrow \quad x^{(k)} = D^{-1}(L + U)x^{(k-1)} + D^{-1}b \qquad (26)$$

This is called the **Jacobi iterative method**. Actually we can do better, as we know not only how to invert D, but also $(D - L)$ or $(D - U)$ efficiently. Equation (26) can be modified to:

$$x^{(k)} = (D - L)^{-1}Ux^{(k-1)} + (D - L)^{-1}b \qquad (27)$$

This is called the **Gauss-Seidel iterative method**, which is very popular for well-conditioned linear equations. The convergence criterion can be simply expressed as:

$$\left\|(D-L)^{-1}U\right\| < 1 \qquad (28)$$

where we can use the matrix norm definition in Eqs. (16) and (17). Row and column permutation may be necessary if the original diagonal element is zero.

Notice that the Jacobi method will not expand the memory usage in Eq. (26), as the inverse of the diagonal matrix is still a diagonal matrix. However, in the Gauss-Seidel iteration, although $(D-L)^{-1}$ remains a lower triangular matrix, there is no guarantee that the number of fill-in is small. We often use the backward substitution in Gauss-Seidel method to obtain the solution vector instead of storing $(D-L)^{-1}$ to ensure small memory storage.

4.2 Convergence of iterative solvers and matrix preconditioning

Now why the iterative method actually can converge? If we observe from the point that convergence can only be achieved that there is no further change in the solution vector $x^{(k)}$ (alternatively we can observe how b approaches 0), then the mapping matrix from $x^{(k-1)}$ to $x^{(k)}$ has to have a norm (or **spectral radius**) from the definition in Eq. (15) to be < 1:

- For Jacobi iteration: $||D^{-1}(L+U)|| < 1$
- For Gauss Seidel iteration: $||(D-L)^{-1}U|| < 1$
- For SOR: $||1 - \omega D^{-1}A|| < 1$

However, the matrix norm cannot be easily obtained (although its upper bound can be estimated with Eqs. (16) and (17)), so this is the correct answer but not very practical. We often perform the **matrix preconditioning** step, i.e., choosing the **diagonally dominant pivoting** by row and column permutation before applying the iterative method. Notice that row and column permutations alone CANNOT guarantee that you reach diagonal dominance, although the sparsity (i.e., the number of nonzero elements) is preserved. You may do pretty well, but sometime not sufficient. Just like the pivoting for direct solvers, to guarantee achievement of diagonal dominance, after column and row permutation, Gaussian steps are needed to finish the task at the cost of fill-in! A trade-offs similar to pivot-on-the-fly is reached: we can choose the best pivot OR the minimum fill-in, but not both at the same time. The detailed choice of row/column permutation and Gaussian steps to make the matrix stably and accurately solvable are the main tasks of matrix preconditioning.

The preconditioning method for the iterative solvers can be life or death just like choosing the pivoting in direct solvers, but this broad topic is out of the scope of this class.

We will use a simple example to illustrate the importance of choosing the diagonal dominance for iterative methods. This is mainly for illustration, and rigorous derivation will be left for your future classes.

Example 1

Use the Jacobi iteration to solve $\begin{bmatrix} 1 & 0.1 \\ 0.1 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.1 \\ 1.1 \end{bmatrix}$. We know the solution is $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$, and will use the initial guess as $x^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

From Eq. (24), we know the iteration should be $x^{(k)} = D^{-1}(L+U)x^{(k-1)} + D^{-1}b$.

$$D^{-1}(L+U) = \begin{bmatrix} 0 & -0.1 \\ -0.1 & 0 \end{bmatrix}. \quad D^{-1}b = \begin{bmatrix} 1.1 \\ 1.1 \end{bmatrix}.$$

We will have $x^{(1)} = \begin{bmatrix} 1.1 \\ 1.1 \end{bmatrix}$; $x^{(2)} = \begin{bmatrix} 0.99 \\ 0.99 \end{bmatrix}$; $x^{(3)} = \begin{bmatrix} 1.001 \\ 1.001 \end{bmatrix}$; $x^{(4)} = \begin{bmatrix} 0.9999 \\ 0.9999 \end{bmatrix}$. We can see that the improvement is 10 times in every iteration from the matrix norm.

For an equivalent problem where we do one row or column iteration and row scaling of the above problem: $\begin{bmatrix} 1 & 10 \\ 10 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 11 \\ 11 \end{bmatrix}$. We know the solution is still $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$, and will still use the initial guess as $x^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. Now we do NOT have diagonal dominance! The Jacobi iteration becomes:

$$D^{-1}(L+U) = \begin{bmatrix} 0 & -10 \\ -10 & 0 \end{bmatrix}. \quad D^{-1}b = \begin{bmatrix} 11 \\ 11 \end{bmatrix}.$$

We will have $x^{(1)} = \begin{bmatrix} 11 \\ 11 \end{bmatrix}$; $x^{(2)} = \begin{bmatrix} -99 \\ -99 \end{bmatrix}$; $x^{(3)} = \begin{bmatrix} 1001 \\ 1001 \end{bmatrix}$; $x^{(4)} = \begin{bmatrix} 9999 \\ 9999 \end{bmatrix}$. The solution diverges.

Example 2: Row and column permutation procedures

Now we will apply the simple diagonal pivoting (row and column permutations only) to the problem:

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$$

We will first search the largest element in the matrix, and find a_{55} = 12. We will do rowPermute(1, 5) and then columnPermute(1, 5) to swith 12 to a_{11}.

rowPermute(1, 5) $$\begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 11 & 0 & 0 & 0 & 12 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 1 & 2 & 0 & 0 & 3 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 3 \\ 2 \\ 5 \end{pmatrix}$$

columnPermute(1, 5) $\begin{pmatrix} 11 & 0 & 0 & 0 & 12 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 1 & 2 & 0 & 0 & 3 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 3 \\ 2 \\ 5 \end{pmatrix} \Rightarrow \begin{pmatrix} 12 & 0 & 0 & 0 & 11 \\ 0 & 5 & 6 & 0 & 4 \\ 9 & 7 & 8 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 \\ 3 & 2 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x_5 \\ x_2 \\ x_3 \\ x_4 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 3 \\ 2 \\ 5 \end{pmatrix}$

We will now search the largest element for the 4 by 4 matrix not containing the first row and column (as a11 is fixed as the first pivot). We find that would be $a_{44} = 10$ We will need to make it a_{22} by row and column permutation.

rowPermute(2, 4) $\begin{pmatrix} 12 & 0 & 0 & 0 & 11 \\ 0 & 5 & 6 & 0 & 4 \\ 9 & 7 & 8 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 \\ 3 & 2 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x_5 \\ x_2 \\ x_3 \\ x_4 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 3 \\ 2 \\ 5 \end{pmatrix} \Rightarrow \begin{pmatrix} 12 & 0 & 0 & 0 & 11 \\ 0 & 0 & 0 & 10 & 0 \\ 9 & 7 & 8 & 0 & 0 \\ 0 & 5 & 6 & 0 & 4 \\ 3 & 2 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x_5 \\ x_2 \\ x_3 \\ x_4 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}$

columnPermute(2, 4) $\begin{pmatrix} 12 & 0 & 0 & 0 & 11 \\ 0 & 0 & 0 & 10 & 0 \\ 9 & 7 & 8 & 0 & 0 \\ 0 & 5 & 6 & 0 & 4 \\ 3 & 2 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x_5 \\ x_2 \\ x_3 \\ x_4 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} \Rightarrow \begin{pmatrix} 12 & 0 & 0 & 0 & 11 \\ 0 & 10 & 0 & 0 & 0 \\ 9 & 0 & 8 & 7 & 0 \\ 0 & 0 & 6 & 5 & 4 \\ 3 & 0 & 0 & 2 & 1 \end{pmatrix}\begin{pmatrix} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}.$

Notice here $a_{31} = 9 > a_{33} = 8$, but we cannot use row and column permutation any more, as $a_{11} = 12$ cannot be moved. We can tolerate it, and "hopefully" the Jacobi iteration can converge, or we can use a Gaussian step to use the first row to eliminate (or reduce) a_{31}.

Row1$\times\dfrac{-9}{12}$ + Row3 $\begin{pmatrix} 12 & 0 & 0 & 0 & 11 \\ 0 & 10 & 0 & 0 & 0 \\ 9 & 0 & 8 & 7 & 0 \\ 0 & 0 & 6 & 5 & 4 \\ 3 & 0 & 0 & 2 & 1 \end{pmatrix}\begin{pmatrix} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} \Rightarrow \begin{pmatrix} 12 & 0 & 0 & 0 & 11 \\ 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 8 & 7 & \frac{-99}{12} \\ 0 & 0 & 6 & 5 & 4 \\ 3 & 0 & 0 & 2 & 1 \end{pmatrix}\begin{pmatrix} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ \frac{3}{4} \\ 4 \\ 5 \end{pmatrix}.$

This creates two problems: first there can be new fill-ins. Second, there can be other elements larger than a_{33} (for now a_{35}). This can be further resolved by one column permutation to maintain the diagonal dominance. We can see in this example that matrix conditioning for iterative methods is similar to the direct solver. Minimum fill-in and the best pivoting cannot be achieved at the same time. Many methods are like pivot-on-the-fly to take the best behavior for one, but tolerate the shortcoming in the other.

columnPermute(3, 5) $\begin{pmatrix} 12 & 0 & 0 & 0 & 11 \\ 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 8 & 7 & \frac{-99}{12} \\ 0 & 0 & 6 & 5 & 4 \\ 3 & 0 & 0 & 2 & 1 \end{pmatrix}\begin{pmatrix} x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ \frac{3}{4} \\ 4 \\ 5 \end{pmatrix} \Rightarrow \begin{pmatrix} 12 & 0 & 22 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & \frac{-99}{12} & 7 & 8 \\ 0 & 0 & 4 & 5 & 6 \\ 3 & 0 & 1 & 2 & 0 \end{pmatrix}\begin{pmatrix} x_5 \\ x_4 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ \frac{3}{4} \\ 4 \\ 5 \end{pmatrix}.$

Hacker Practice 4.6:

Use the Jacobi iterations to solve the same problem below.

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$$

Use $D^{-1}b$ as your initial guess and $\|\Delta x\|_2 < 10^{-7}$ as the convergence criteria. Observe how many iterations are needed for convergence and what precision has been achieved against the direct solver in your previous hacker practice by checking the evolution of the residual vector: $\|b - Ax^{(k)}\|_2$. Now check the first and infinite norms of $\|D^{-1}(L+U)\|$. Which one gives a better indication of the convergence property?

If you have the function to calculate $(D - L)^{-1}U$, repeat the above for the Gauss-Seidel method.

4.3 Other iterative solvers

By defining the residual vector during iterations as

$$r^{(k)} = b - Ax^{(k)} \qquad (29)$$

We can now devise an iterative solution as:

$$x^{(k)} = x^{(k-1)} + \omega D^{-1} r^{(k-1)} \qquad (30)$$

If $\omega < 1$, this is called under-relaxation. More often, we choose $\omega > 1$ for faster convergence (e.x., $\omega =$ 2 to 10), which is called **successive over-relaxation (SOR)**.

If the original matrix is less diagonal dominant, we can choose to solve initially

$$(A + \xi I)x = b \qquad (31)$$

by Gauss-Seidel or SOR, and gradually let ξ go to 0. This will relax the requirement of a good initial guess by increasing the diagonal dominance of the problem, which is equivalent to "preconditioning" A for the iterative solver. Many other forms of preconditioning can be applied, but are out of the scope here.

Hacker Practice 4.7:

Now use the SOR iterations $x^{(k)} = x^{(k-1)} + \omega D^{-1} r^{(k-1)}$ to solve the problem below with $\omega = 2$.

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$$

Use $D^{-1}b$ as your initial guess and $\|r\|_2 < 10^{-7}$ as the convergence criteria. Observe how many iterations are needed for convergence and what precision has been achieved against the direct solver in your previous hacker practices.

All these iterative methods can have their specific implementations to make the computation faster for different platforms. For example, on a serial platform, during $x^{(k)}$ calculations, we can use as many already known elements in $x^{(k)}$. Actually, the Jacobi iteration can become the Gauss-Seidel iteration by applying this principle. However, this does increase the data dependence within the present iteration, and may not be the most efficient way for vectorized- or parallel-computing platforms. For very large matrix computation, the answer is often application specific, specific to the problem and to the platform. For computer gamers, this should not be surprising. With image computation on GPU (and the rising stock price of nVidia), many game execution can only be optimized with specific platform architecture.

5. Software aspect of large data set and program with library functions

When we start constructing programs to deal with large data set, often we will use a lot of "library functions" written by other people. As we encourage code reuse as much as possible for both programming efficiency and less debugging, there are two additional effects we will introduce below.

5.1 Namespace in C++

The more extensive programming becomes, there are higher chances that some function names or global variables (which should be minimally or sparingly used anyway) can overlap. Originally in C or Java, this is resolved by giving library functions a fixed prefix, such as MV_ for the matrix-vector library and SP_ for the sparse matrix library. In C++, we introduce "namespace" to further identify possibly overlapped names, even though the programmer does not know all of the other function names in the library. For example, if we create a class called `LinkedList` to store your implementation of a linked list, it is surely possible that some of the library functions you are using has already a class with the same name but a different implementation. In any complete programs, you cannot have two classes with the same name without creating confusion in the linker. To avoid this conflict, you can extend to the basic name of the type by creating a **namespace**. I can put a qualifier to my linked list class into the namespace called `com::cprogramming`, so that the fully qualified name of my implemented class is `com::cprogramming::LinkedList`. Using a namespace drastically reduces the chance of a naming conflict, although it is still possible that people overlapping the same name of "namespace". The :: operator used here is the same as the one used for accessing static members of a class or declaring a method, but instead of being used to access elements of a class, it is being used to access elements of a namespace.

5.2 Computational efficiency for large data set

Even though we have CPU, MPU, and GPU with clocks higher than several GHz, there are many engineering and gaming applications that the computer is still too slow, even with the best available algorithms. It is important to have your code to run to its full algorithmic potential, if not for your personal satisfaction, it will be for the commercial interest. You probably already feel your reality game runs too slowly, or there is no real-time speech or image processing. Many codes that do not run as fast as they can because of the lack of the knowledge of the underlying platform. Yes, there is Internet programming that tries to be totally independent of the hardware platform, but those programs are for collecting information from the database where the computing efficiency is not important in comparison with the network transaction time, which is often not limited by the link but by the security protocol. On the other hand, we focus on the heavy computing here, such as those encountered in engineering design simulation or reality games with a lot of graphics and geometry handling.

In addition to the clock speed, the most dominant factor of computing is the **memory wall** and **data locality**. **Rent's rule** (a mathematical rule) dictates that the larger the database, the longer the delay to retrieve information from its peripheral circuits. This will NOT change even though we have totally new ways of implementing the electronics, as it is a basic rule that derives from the smaller surface area of a large object. The data retrieval can be even slower for many new proposals of computing architecture such as neuromorphic computing. After the code has been implemented with all unit and regressional tests, it will enter the "performance optimization" scheme. In the compiler, this can be as simple as changing the –g flag to –O flag, but often the following cycle will be ensued:

1. Give compiler directives about how to optimize performance on the specific platform by querying the platform about memory sizes, instruction delay and cache structures (often in BIOS or boot processes).
2. Perform profiling of each function, if not each line, to identify the present performance bottleneck. Many IDEs will provide such profiling functions.
3. Give further compiler directives on the data dependency so that vectorization and parallelization available on the specific platform can be fully utilized. Today's computing platforms often have many "cores", and vectorization and parallelization have become more common, as well as important. As this step is platform dependent, the code needs to be "ported" for performance optimization.
4. Benchmark the performance on all of the available units and regression tests. For the next cycle of code development with change in algorithms and data structure, repeat from Step 1.

It is not **uncommon** that these steps can speed up computing to 10 – 100 times! For example, LU factorization of a full n×n matrix will take about $\frac{2}{3}n^3$ flops (floating point operations). For a given CPU with 3GHz clock, we should solve the whole 1,000×1,000 matrix within 1 seconds. However, an unoptimized code can run for more than minutes, due to the latency by memory walls. Remember that computation is ONLY as fast as the available data. If the data are not directly or speedily available, the CPU speed will NOT matter. For typical CPU, the memory has the hierarchy of register, level-1 (L1) cache, level-2 (L2) cache, three of which are embedded on the chip, main memory (often on PCB by SDRAM) and nonvolatile storage (NAND Flash as solid state disk or magnetic disk). A typical computing platform has the following size and latency for these memories listed in Table 2.

Table 2. Typical numbers for memory hierarchy (in Year 2016 under the Moore's Law)

Data storage	System size (bits or bytes)	Instruction cycle delay
Registers	256 – 1,024 bits	1
L1 cache	64 – 256 KB	1 – 2
L2 cache	2 – 64 MB	3 – 8
Main off-chip memory	4 – 16 GB	10 – 30
Virtual memory on disk	1 – 4 TB	100 – 500 (buffer dependent)

If your computing has to retrieve information often from virtual memory, surely it can be 100 times slower when most of the computing with data available in the on-chip cache. As you can see we need to compute with the most data locality, maximizing the use of data in Registers and L1 cache and minimizing data retrieval from virtual memory. For sure, due to the large data set, the cache cannot store all data. However, if the data are operated fully in each subset without too much swapping, we can still achieve much better computing efficiency. When the compiler cannot decide on a data set that can fit into the available cache, cache thrashing can happen, where cache miss rate will be high and memory bus will be crowded. Therefore, it is important to give the compiler enough direction to chop the data set into pieces that can reasonably fit into cache during heavy computing.

Data locality can be separated to temporal and spatial locality. Both are important for efficient computing. For temporal locality, we will keep the most often used data in cache by informing the compilers (global data by default will belong to this set). For spatial locality, accessing the array stored sequentially will help greatly. To optimize the use of data locality, it is important for the compiler to know the computing order, and perform computing accordingly. The more predictable the computing order, the less the opportunity for cache miss and for pipeline flushing in the multi-threading and speculative computing domain. You can see that symbolic LU factorization provides the compiler with known computing order, and is critical for efficient array computing.

The entire program can be too large for the programmer to go through the optimization compiling process, and the next best practice is to use specifically compiled library functions to handle large data set, such as blas (basic linear algebra subprograms), sparselib, OpenGL (graphic library), etc., available in many platforms.

5.3 Vectorization and parallelization

Vectorization is to maximize command pipelining and executing the same instruction for multiple data set, while parallelization utilizes the multiple threads/cores within a chip to execute several parts of computing at the same time. Vectorization belongs to the computing architecture category of SIMD (single instruction multiple data), while parallelization to the computing architecture of MIMD (multiple instruction multiple data). Most memory architecture supports both SIMD and MIMD, but the memory can be distributed to each processing unit (distributed memory) or share a common large data memory (shared memory). The size and latency tradeoffs are similar to Table 2, and we often have a hybrid shared and distributed memory system in the modern multi-core CPU, as shown in Fig. 1.

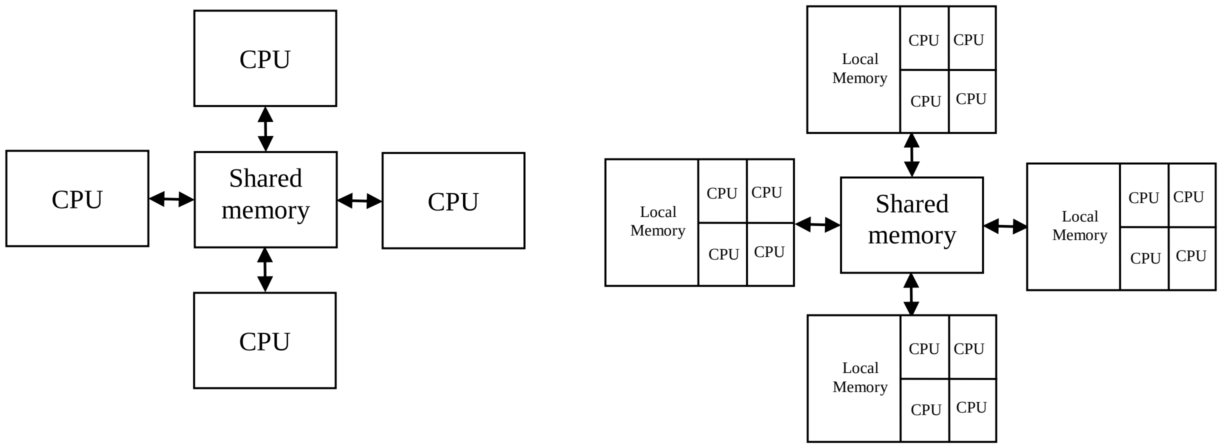

Fig. 1. (a) Shared memory for four CPU; (b) hybrid multi-core architecture with shared memory.

At the high-level programming for multiple possible hardware platform, we can inform the compiler about the data dependency so that best decision for the specific platform for vectorization and parallelization can be made. In C++, this is often achieved by OpenMP (multi-processing) API, where data that can or should not be parallelized are given explicit directives. The format is often like:

```
#OMP parallel default (shared) private (local)
```

For parallelization, the compiler will do the thread assignment as:

1. Spawning a parallizable region for multiple thread execution
2. Dividing blocks of code among threads
3. Distributing loops among threads
4. Synchronization of threads for the non-parallelizable part.

The ultimate parallel processing speedup is eventually dependent on the portion of computing that cannot be parallelized, which is called the Amdahl's law. Assume the parallelizable fraction of the computing is *f* and the number of available processor is *N*, then the total speedup *S* of computing by using *N* processors in the most optimal way will be:

$$S = \frac{1}{(1-f)+\frac{f}{N}} \qquad (31)$$

Amdahl's law has many variations, and basically dictates that the bottleneck controls the resource usage.

6. Epilog on matrix computing

Even the Moore's law has given us 60 years of exponential growth in computing capability and memory capacity, efficient computing algorithms and programming practice become MORE important, instead of less important. As the computer is finally fast and large enough for important problems such as genomic prediction, weather forecast, and reliable engineering design domain, the program scope and complexity grow ever higher in history. Efficient algorithms and coding practice for complex problems actually give even MORE speedup than the generic coding and inefficient algorithms!

Although the matrix solver almost covers the entire world of present scientific and engineering problems, it does have a twist in the fundamental way that the problem is dealt with. Matrix solver is something a digital computer can do much better than human. Before electronic computers exist, computers are human operators going through computing tables and calculation sliding rules (if not abacus). The movie "Imitation Games" retold the story of how human and machine computing could be perceived by Alan Turing. The bottom line is: there are things that machine can do better than human, but there are also domains that human can produce better decision than machine. Matrix computation certainly belongs to the first category, even though machine has finite precision. Actually, sometime we asked the question whether to be more "human", we need to lose some precision and exactness. For sure, if error pollutes the solution entirely, the more computing will just become "GIGO" (garbage in garbage out). However, computing with inexactness under insufficient information (such as compressed sensing) is still a research topic that demands many of your effort to give better answers to improve artificial intelligence to true intelligence. This is especially important when information and computing become the essential part of our daily lives.