Ch 09
Multidimensional arrays & Linear Systems

Andrea Mignone

Physics Department, University of Torino

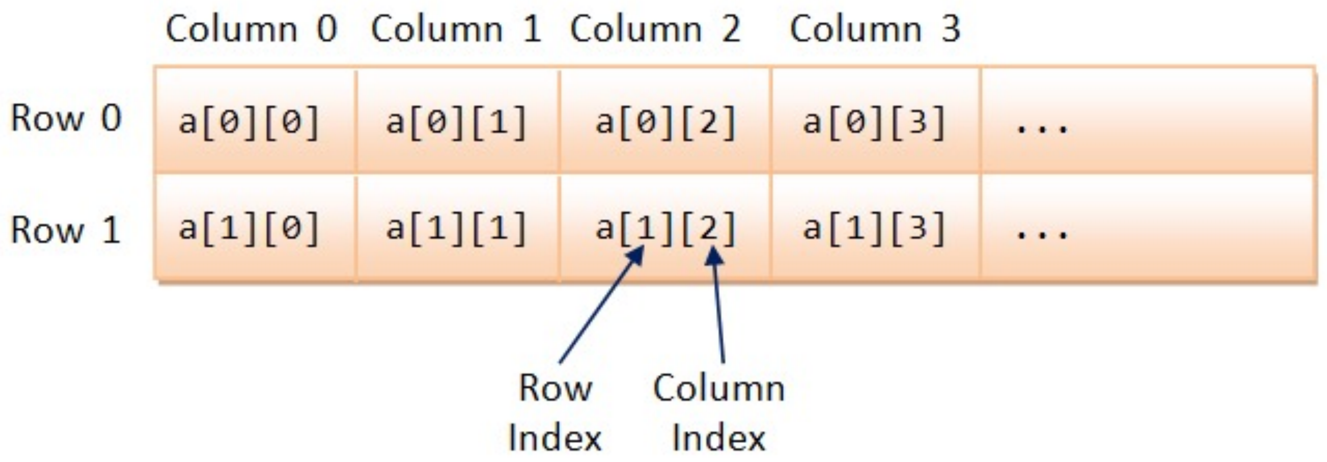
AA 2022-2023

Multidimensional Arrays

- A multidimensional array is an array containing one or more arrays.
- They are very similar to standard arrays but they have multiple sets of square brackets after the array identifier:

```
double arr[4][6]; // Declares arr as array of 4x6 elements in dbl precision
arr[0][0] = 1;    // first element
  ...
arr[3][5] = 2;
```

- In a two-dimensional array, the *first* subscript represents row number, and the *second* represents the *column* number. For instance:



Row or column-major order

- Memory storage is linear: array elements are actually stored sequentially in a computer memory.
- Consider the matrix $$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$$
- In a column-major order (Fortran), consecutive elements of the columns are contiguous:

Index	0	1	2	3	4	5
Value	11	21	12	22	13	23

- In a row-major order (C, C++), consecutive elements of the rows of the array are contiguous:

Index	0	1	2	3	4	5
Value	11	12	13	21	22	23

Creation of Multi-D Arrays

- There exists different methods to create multidimensional array in C or C++.

- We will describe two approaches:

1. Standard native way (beginners)

2. Dynamic allocation with pointers to pointers (expert users)

Method #1: Standard way

- The standard C/C++ multidimensional array declaration is:

```
double arr[3][5]; // arr has 3 rows and 5 columns
Func (arr);       // Pass arr to the function "Func()"
```

- Passing this array to a function requires knowing the numbers of columns:

```
void Func (double arr[][5])
{
   ...
}
```

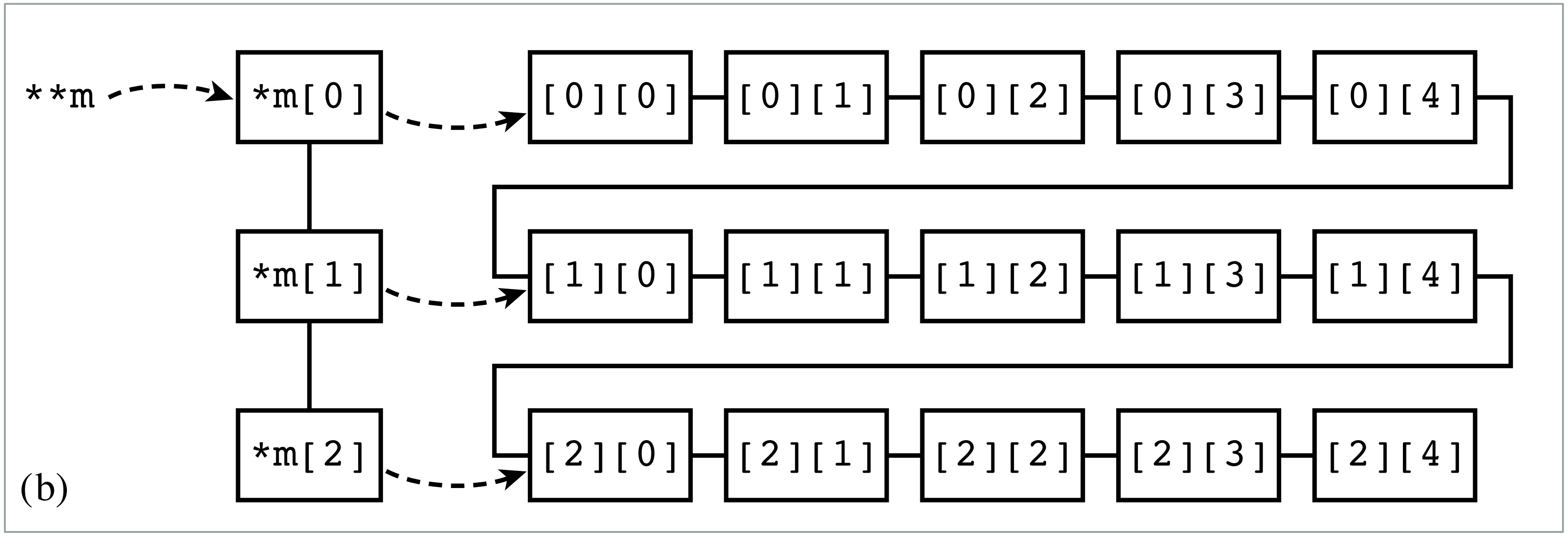
- *Advantages*: simple, straightforward to implement, built-in.
- *Limitations*: functions must know size of the array (can't call the function with array of different size) → problems with dynamical allocation.

- Initialization may be done using a brace-enclosed list of array elements:

```
int main()
{
  int arr[2][3] = { {11, 22, 33}, {44, 55, 66} };
}
```

Method #2: Using pointers to pointers

- An alternative way requires allocating memory for one single array of N_{row} X N_{col} elements and then a second array of N_{row} pointers.
- The elements of this arrays points to the _rows_ of the original 1D array:



- Pointer to an array of pointers to rows; Dotted lines denote address reference, while solid lines connect sequential memory locations.

- This is, in essence, _dynamical allocation_ of memory.

Method #2: Using pointers to pointers

- In C++, the pseudo-code for dynamical allocation of 2D array is

```
// Dynamical creation of a multi-D array A[nrow][ncol] in C++

double **A;
A     = new double*[nrow];
A[0] = new double [ncol*nrow];
for (int i = 1; i < nrow; i++)
  A[i] = A[i-1] + ncol;
```

- Passing this array to a function is simply done through

```
void Func (double **arr)
{
  ...
}
```

- Advantages: more flexible and versatile. Allows for dynamic allocation when array size is not known at compilation time.

- Limitations: requires good knowledge of pointers and (IMPORTANT) *memory must be freed at the end of the function !!*

Method #2: Using pointers to pointers

- In C, dynamical allocation can be achieved using the `malloc()` function:

```
// Dynamical creation of a multi-D array A[nrow][ncol] in C

m    = (double **)malloc ((size_t) nrow*sizeof(double *));
m[0] = (double *) malloc ((size_t) nrow*ncol*sizeof(double));
for (i = 1; i < nrow; i++) m[i] = m[i-1] + ncol*sizeof(double);
```

- **<u>! Important !</u>** Allocated memory must be freed once the array is no longer needed,

```
delete[] A[0]; // In C++
delete[] A;
```

```
free ((double *) A[0]); // In C
free ((double *) A);
```

Practice Session #0

- `matrix.cpp`: write a simple program that creates a `NxN` matrix `A` using either standard native or dynamical method (use, e.g. `N = 4`).
- Print the matrix on screen using nice-formatted output. An example is:

```
// Use a fixed precision of 4 digits
cout << fixed << setprecision(4);

// Print the matrix
for (i = 0; i < n; i++){
  for (j = 0; j < n; j++){
    cout << setw(10) << right << A[i][j] << "  ";
  }
  cout << endl;
}
```

- Now construct a a function that perform Matrix-vector multiplication with an array `b` of size `N`. Verify that, using

$$A := \begin{bmatrix} 1 & 3 & 2 & -4 \\ 7 & 2 & 4 & 1 \\ 0 & -1 & 2 & 2 \\ 6 & 3 & 0 & 1 \end{bmatrix} \qquad b := \begin{bmatrix} 1 \\ 0 \\ 3 \\ 2 \end{bmatrix} \quad \longrightarrow \quad A\,\mathbf{b} = \begin{bmatrix} -1 \\ 21 \\ 10 \\ 8 \end{bmatrix}$$

Linear System of Equations

- A linear system of equations has the form

$$\boxed{\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3N}x_N &= b_3 \\ \cdots \qquad & \quad \cdots \\ a_{M1}x_1 + a_{M2}x_2 + a_{M3}x_3 + \cdots + a_{MN}x_N &= b_M \end{aligned}}$$

- Here the `N` unknowns `x`$_j$ `(j=1..N)` are related by `M` equations . The coefficients aij as well the `b`$_i$ `(i=1..N)` on the right hand side are known numbers.

- The system can also be written in matrix notations as $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ where

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1N} \\ a_{21} & a_{22} & \ldots & a_{2N} \\ & \cdots & & \\ a_{M1} & a_{M2} & \ldots & a_{MN} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_M \end{bmatrix}$$

- By convention, the 1st index on an element denotes its row, the 2nd index its column

Linear System of Equations

- The system can be solved when `N=M`, i.e. the number of equations matches the number of unknowns.

- A unique solution exists if none of the equations can be written as a linear combination of the others: otherwise we are in presence of a row or column degeneracy and the set of equations is called degenerate.

- From a computational view, at least two things can go wrong:

  - Even for non-singular matrices, some of the equations may be so close to linearly dependent that roundoff errors render them linearly dependent at some stage in the solution process. In this case your numerical procedure will fail.
  - Accumulated roundoff errors in the solution process can swamp the true solution. This problem particularly emerges if N is too large. The numerical procedure does not fail algorithmically. However, it returns a set of x's that are wrong, as can be discovered by direct substitution back into the original equations. The closer a set of equations is to being singular, the more likely this is to happen, since increasingly close cancellations will occur during the solution. In fact, the preceding item can be viewed as the special case where the loss of significance is unfortunately total.

Standard Libraries

- In this lecture we will only scratch the surface about this vast subject.
- In many cases you will have no alternative but to use sophisticated black-box program packages: LINPACK, LAPACK, NAG, PETSc, etc…
- Keep in mind that the sophisticated packages are designed with very large linear systems in mind. They therefore go to great effort to minimize not only the number of operations, but also the required storage.
- Routines for the various tasks are usually provided in several versions, corresponding to several possible simplifications in the form of the input coefficient matrix: symmetric, triangular, banded, positive definite, etc. If you have a large matrix in one of these forms, you should certainly take advantage of the increased efficiency provided by these different routines, and not just use the form provided for general matrices.
- Algorithms are divided into routines that are *direct (i.e., execute* in a predictable number of operations) from routines that are *iterative (i.e., attempt* to converge to the desired answer in however many steps are necessary).

Computational Cost by Analytical Methods

We begin by giving a short illustration in operation counting for the computation of $\det(A)$, for a given $n \times n$ matrix A. From the formula in elementary linear algebra,

$$\det(A) = \sum_{\sigma} \operatorname{sign}(\sigma) a_{1\sigma_1} \cdots a_{n\sigma_n}, \tag{1.1}$$

where the summation is taken for all permutation $\sigma : \{1, \cdots, n\} \mapsto \{1, \cdots, n\}$ and the signature function is defined as usual:

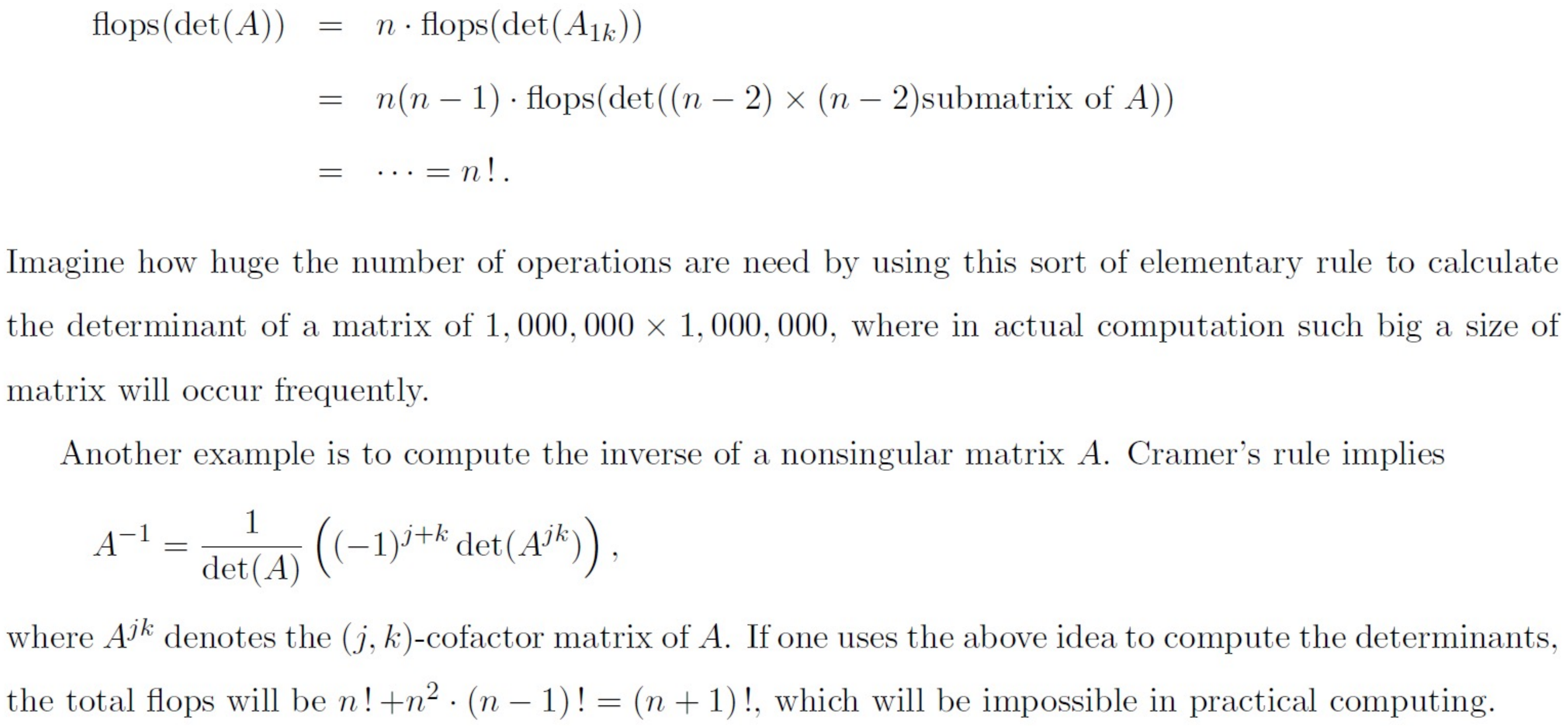
$$\operatorname{sign}(\sigma) = \begin{cases} 1, & \sigma \text{ is an even permutation,} \\ -1, & \sigma \text{ is an odd permutation.} \end{cases} \tag{1.2}$$

Moreover, the determinant of A can be computed using cofactor matrices in the following manner:

$$\det(A) = \sum_{k=1}^{n} (-1)^{1+k} a_{1k} \det(A_{1k}), \tag{1.3}$$

where A_{1k} is the submatrix of size $(n-1) \times (n-1)$ obtained by deleting the 1st row and kth column from A, which is identical to the $(k, 1)$-cofactor matrix of A. The determinants of A_{1k}'s then can be computed by using the formula (1.3), recursively, until the size of cofactor matrices become 1×1. Therefore,

Computational Cost by Analytical Methods



$$\begin{aligned}\text{flops}(\det(A)) &= n\cdot\text{flops}(\det(A_{1k}))\\ &= n(n-1)\cdot\text{flops}(\det((n-2)\times(n-2)\text{submatrix of }A))\\ &= \cdots = n!\,.\end{aligned}$$

Imagine how huge the number of operations are need by using this sort of elementary rule to calculate the determinant of a matrix of $1,000,000 \times 1,000,000$, where in actual computation such big a size of matrix will occur frequently.

Another example is to compute the inverse of a nonsingular matrix A. Cramer's rule implies

$$A^{-1} = \frac{1}{\det(A)}\Big((-1)^{j+k}\det(A^{jk})\Big)\,,$$

where A^{jk} denotes the (j,k)-cofactor matrix of A. If one uses the above idea to compute the determinants, the total flops will be $n!+n^2\cdot(n-1)! = (n+1)!$, which will be impossible in practical computing.

- For $n = 20$, this would require 2×10^{18} operations which, on your powerful mac pro (≈10 gigaflops) would require ≈ 6.3 years to complete.
- We will show how the flops will be substantially reduced by decomposing the matrix A into a product of lower and upper triangular matrices which results from a Gaussian elimination procedure.

Gaussian Elimination

- Gaussian elimination is a powerful method to approach the solution of linear system in the form

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

- It is based on the fact that
  - Interchanging any two *rows of* ***A and the corresponding rows of the b's,*** **does not change** (or **scramble in any way) the solution x's. Rather, it just corresponds to writing the same set of linear equations** in a different order.
  - Likewise, the solution set is unchanged and in no way scrambled if we replace any row in **A by a linear combination of itself and any other row,** as long as we do the same linear combination of the rows of the **b's** (which then is no longer the identity matrix, of course).
  - Interchanging any two *columns of* ***A gives the same solution set only*** if we simultaneously interchange corresponding *rows of the* **x's. In other words, this interchange scrambles the order of the rows in** the solution. If we do this, we will need to unscramble the solution by restoring the rows to their original order.

Gaussian Elimination

- Gaussian elimination attempts to reduce the matrix `A` to an upper triangular matrix by elementary row operations.
- As an example, consider the 3x3 matrix

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \longrightarrow \begin{array}{lll} r_1: & (a \quad b \quad c) & \\ r_2: & (d \quad e \quad f) & -= \frac{d}{a}(a \quad b \quad c) \\ r_3: & (g \quad h \quad i) & -= \frac{g}{a}(a \quad b \quad c) \end{array} \longrightarrow A' = \begin{pmatrix} a & b & c \\ 0 & e' & f' \\ 0 & h' & i' \end{pmatrix}$$

$$A' = \begin{pmatrix} a & b & c \\ 0 & e' & f' \\ 0 & h' & i' \end{pmatrix} \longrightarrow \begin{array}{lll} r_1: & (a \quad b \quad c) & \\ r_2: & (0 \quad e' \quad f') & \\ r_3: & (0 \quad h' \quad i') & -= \frac{h'}{e'}(0 \quad e' \quad f') \end{array} \longrightarrow A'' = \begin{pmatrix} a & b & c \\ 0 & e' & f' \\ 0 & 0 & i'' \end{pmatrix}$$

- The same operations must be also applied to the right-hand side vector `b` → `b"`.

- Now `A"` is in upper triangular form and therefore the solution can be easily found by starting from the last equation going backward (backsubstitution)

$$\begin{cases} x_3 = b_3''/i'' \\ x_2 = (b_2'' - f'x_3)/e' \\ x_1 = (b_1'' - cx_3 + bx_2)/a \end{cases}$$

Gaussian Elimination

- In general, subtracting the k^{th} row from all the rows m with $m > k$ is achieved by means of a Gaussian transformation matrix

$$G_k = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -g_{k+1} & 1 & & \\ & & \vdots & & \ddots & \\ & & -g_N & & & 1 \end{pmatrix} \quad \rightarrow \quad G_k A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ & \ddots & & & & \\ & & a_{kk} & & & \\ & & 0 & a'_{k+1,k+1} & & \\ & & \vdots & & \ddots & \\ & & 0 & & & a'_{NN} \end{pmatrix}$$

where the g_k are properly chosen to produce zeros on the corresponding elements of G_kA:

$$g_i = \frac{A_{ik}}{A_{kk}} \qquad \text{for} \quad i > k$$

- Note that G_k is lower triangular.

Gaussian Elimination

- By applying n-1 Gaussian transformation matrices, we arrive at

$$G_{n-1}G_{n-2}...G_1A = U \quad \rightarrow \quad A = LU$$

where U is upper-triangular and L is lower triangular (exercise):

$$L = (G_{n-1}G_{n-2}...G_1)^{-1} = G_1^{-1}...G_{n-2}^{-1}G_{n-1}^{-1}$$

- At this point we can easily solve for the unknowns x:

$$x_k = \left(b_k - \sum_{j>k} a_{kj}x_j\right)/a_{kk}$$

starting at k=n, n-1, … , 1.

An Example

- Using paper and pen (!) apply the Gaussian elimination algorithm on the following 3x3 matrix A_0:

$$A_0 := \begin{bmatrix} 1 & 2 & -1 \\ 2 & 3 & 4 \\ 0 & -1 & 2 \end{bmatrix} \qquad G_1 := \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$A_1 := \begin{bmatrix} 1 & 2 & -1 \\ 0 & -1 & 6 \\ 0 & -1 & 2 \end{bmatrix} \qquad G_2 := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}$$

$$A_2 := \begin{bmatrix} 1 & 2 & -1 \\ 0 & -1 & 6 \\ 0 & 0 & -4 \end{bmatrix}$$

Pseudocode

- The Gaussian elimination algorithm requires approximately $2n^3/3$ operations and remains the most generally applicable method of solving systems of linear equations.

- In C, given a matrix `A` of dimension `nxn` and an array of dimension `n`, the pseudocode can be written

```
for (k = 0...n-2) {                 // Loop over the Gk's
  for (i = k+1...n-1) {             // Loop over rows
    g = A[i][k]/A[k][k];
    for (j = k+1...n-1) A[i][j] -= g*A[k][j];
    A[i][k] = 0.0;
    b[i]   -= g*b[k];
  }
}
```

- Once the matrix has been reduced to upper triangular form, the corresponding system of linear equations can be solved by back substitution,

```
for (i = n-1...0){
  tmp = b[i];
  for (j = n-1...j > i) tmp -= x[j]*A[i][j];
  x[i] = tmp/A[i][i];
}
```

Practice Session #1

- `gauss_elim.cpp`: implement the Gaussian elimination algorithm for a general linear system of `n x n` equations.

→ Test the algorithm on the following 3x3 system:

$$Ax = b \qquad \text{with} \quad A = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$$

The solution is `x = {2,3,2}.`

→ Similarly, test the algorithm on the following 4x4 system

$$Ax = b \qquad \text{with} \quad A = \begin{pmatrix} 1 & 2 & 1 & -1 \\ 3 & 2 & 4 & 4 \\ 4 & 4 & 3 & 4 \\ 2 & 0 & 1 & 5 \end{pmatrix}, \quad b = \begin{pmatrix} 5 \\ 16 \\ 22 \\ 15 \end{pmatrix}$$

The solution is `x = {16, -6, -2, -3}.`

Partial Pivoting

- It can be seen from the algorithms for Gaussian elimination and back-substitution can break down if a diagonal element of the matrix is equal to zero. In order to work around this potential pitfall, another elementary row operation can be used: a row interchange.

- Of course, the same operation must be carried out on the vector b.

- If, when computing the multiplier g_{ik} during Gaussian elimination, the entry a_{kk} is equal to zero then, to avoid breakdown of the algorithm, row k of the matrix A can be interchanged with row i, for some $i > k$, where $a_{ik} \neq 0$, and then Gaussian elimination can continue.

- This is called partial pivoting. Partial pivoting can be applied at every step k of the algorithm in order to make the inversion more stable: at stage k of our algorithm we search the row $j > k$ such that $|a_{jk}|$ is maximum and larger than $|a_{kk}|$. We then interchange rows k and j.

Practice Session #2

- Modify the previous program by introducing partial pivoting. The modification should search for the row `jmax` such that |`A[jmax][k]`| `>=` |`A[j][k]`| for `j > k`.

- Test Gaussian elimination with partial pivoting on the system

$$Ax = b \qquad \text{with} \quad A = \begin{pmatrix} 1 & 2 & 1 & -1 \\ 3 & 6 & 4 & 4 \\ 4 & 4 & 3 & 4 \\ 2 & 0 & 1 & 5 \end{pmatrix}, \quad b = \begin{pmatrix} 5 \\ 16 \\ 22 \\ 15 \end{pmatrix}$$

with solution x={4, -12, 22, -3}

LU Decomposition

- As shown before, Gaussian elimination allows to decompose the matrix A into a lower-triangular and an upper triangular matrix: `A = LU`.
- In many applications, when you solve `Ax=b`, the matrix `A` remains unchanged, while the right hand side vector b keeps changing. For instance:
  - solving a partial differential equation for different forcing functions: the matrix A only depends on the mesh parameters and hence remains unchanged for the different forcing functions which define the `b`'s.
  - solving a time dependent problem, where the unknowns evolve with time. If the time stepping is constant, the matrix `A` remains unchanged and the only the right hand side vector `b` changes at each time step.
- The key idea behind solving using the LU factorization is to decouple the *factorization phase* (usually computationally expensive) from the *actual solving phase*. The factorization phase only needs the matrix `A`, while the *actual* solving phase makes use of the factored form of `A` and the right hand side b to solve the linear system. Hence, once we have the factorization, we can make use of the factored form of `A`, to solve for different right hand sides at a relatively moderate computational cost.
- The cost of factorizing the matrix `A` into `LU` is $O(N^3)$. Once you have this factorization, the cost of solving i.e. the cost of solving `L(Ux)=b` is just $O(N^2)$, since the cost of solving a triangular system scales as $O(N^2)$. (See Numerical Recipe, sect. 2.3)

Tridiagonal Matrices

- The special case of a system of linear equations that is *tridiagonal (that is, has* nonzero elements only on the diagonal plus or minus one column) is one that occurs frequently:

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & 0 & 0 & 0 & 0 & a_N & b_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_{N-1} \\ r_N \end{pmatrix}$$

- This system has tridiagonal form and can be efficiently inverted in (order) N operations (rather than N^2) by using a recurrence relation rather than inverting the full matrix.

- Written in components

$$\begin{cases} & & b_1 x_1 & + & c_1 x_2 & = r_1 & \\ a_i x_{i-1} & + & b_i x_i & + & c_i x_{i+1} & = r_i & \text{for} \quad i = 2 \ldots N-1 \\ a_N x_{N-1} & + & b_N x_N & & & = r_N & \end{cases}$$

- Of course, there's no need to store the entire matrix and only three arrays `a[ ], b[ ]` and `c[ ]` must be stored. The computation can proceed using Gaussian elimination which, in this case, can be coded very concisely.

Tridiagonal Solver

- We solve the matrix equation by manipulating the individual equations until the coefficient matrix is upper triangular with all the elements of the main diagonal equal to 1. We start by dividing the first equation by b_1, then subtract a_2 times the 1st row from the second one:

$$\begin{pmatrix} 1 & c_1/b_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & b_2 - a_2c_1/b_1 & c_2 & 0 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & 0 & 0 & 0 & 0 & a_N & b_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} r_1/b_1 \\ r_2 - a_2r_1/b_1 \\ r_3 \\ \vdots \\ r_{N-1} \\ r_N \end{pmatrix}$$

  We then divide the 2nd equation by the second diagonal element, etc…
- It is easily proven that the final system is

$$\begin{pmatrix} 1 & h_1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & h_2 & 0 & \cdots & 0 \\ 0 & 0 & 1 & h_3 & \cdots & 0 \\ 0 & \cdots & \cdots & \ddots & \ddots & \cdots \\ 0 & 0 & 0 & 0 & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \ddots \\ \cdot \\ x_N \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ \ddots \\ \cdot \\ p_N \end{pmatrix},$$

where

$$\begin{cases} h_i = \dfrac{c_i}{b_i - a_ih_{i-1}} \\ p_i = \dfrac{r_i - a_ip_{i-1}}{b_i - a_ih_{i-1}} \end{cases}$$

with $h_1 = c_1/b_1$, $p_1 = r_1/b_1$.

Tridiagonal Solver

- Finally, using backsubstitution, we obtain the explicit solution for the unknowns:

$$x_i = p_i - h_i x_{i+1}, \quad \text{for} \quad i = n-1, \ldots, 1$$

- Note that the 1D arrays can be safely defined to have N elements: in practice a_1 and c_N will never enter in the solution process.

- Note also that there is no pivoting in tridiagonal. It is for this reason that the algorithm can fail even when the underlying matrix is nonsingular: a zero pivot can be encountered even for a nonsingular matrix. In practice, this is not something to lose sleep about.

- The kinds of problems that lead to tridiagonal linear sets usually have additional properties which guarantee that the algorithm will succeed.

- The most usual one is called diagonal dominance:

$$|b_j| > |a_j| + |c_j| \qquad j = 1, \ldots, N$$

- It is not uncommon to find tridiagonal matrices with $N = 10^4$ elements or more.

Practice Session #3

- `tridiag.cpp`: implement the tridiagonal matrix solver and test it on the following system:

$$A := \begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{bmatrix} \qquad b := \begin{bmatrix} 1 \\ 0 \\ 3 \\ 1 \\ 0 \end{bmatrix}$$

with solution x ={2,-3,4,-2,1}.

Tridiagonal Systems: Application to BVP

- A typical situation where tridiagonal systems have to be solved it the case of a second-order ODE in which the 2nd derivative is discretized using a 2nd order finite difference approximation. In a Boundary Value Problem (BVP), for instance:

$$\begin{cases} \dfrac{d^2y}{dx^2} = f(x,y) \\ y(a) = \alpha, \quad y(b) = \beta \end{cases} \quad \rightarrow \quad \begin{cases} \dfrac{y_{i+1} - 2y_i + y_{i-1}}{h^2} = f(x_i, y_i) \\ y(x_1) = \alpha, \quad y(x_N) = \beta \end{cases}$$

- The previous set of equations has to be solved on a lattice of N points with assigned boundary conditions $y_1 = y(x_1) = \alpha$ and $y_N = y(x_N) = \beta$.

- The resulting equations may, in general, be nonlinear and a local linearization with Newton method usually results in an iterative scheme that, starting with a guess, improves it iteratively: the result is said to *relax to the true solution.*

- This is the idea behind *relaxation methods*, where the ODE is approximated by finite-difference equations on points that spans the domain of interest.

Tridiagonal Systems: Application to BVP

- If, for example, the r.h.s does not contain `y`, then one has to solve a linear system of equations in the unknowns y_i `(i=2..N-1)`.

$$\begin{aligned}
y_1 - 2y_2 + y_3 &= h^2 f(x_2) \\
y_2 - 2y_3 + y_4 &= h^2 f(x_3) \\
\vdots \quad &= \vdots \\
y_{N-2} - 2y_{N-1} + y_N &= h^2 f(x_{N-1})
\end{aligned}$$

- Which defines the tridiagonal system:

$$\begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_{N-2} \\ y_{N-1} \end{pmatrix} = \begin{pmatrix} h^2 f_2 - \alpha \\ h^2 f_3 \\ h^2 f_4 \\ \vdots \\ h^2 f_{N-2} \\ h^2 r_{N-1} - \beta \end{pmatrix}$$

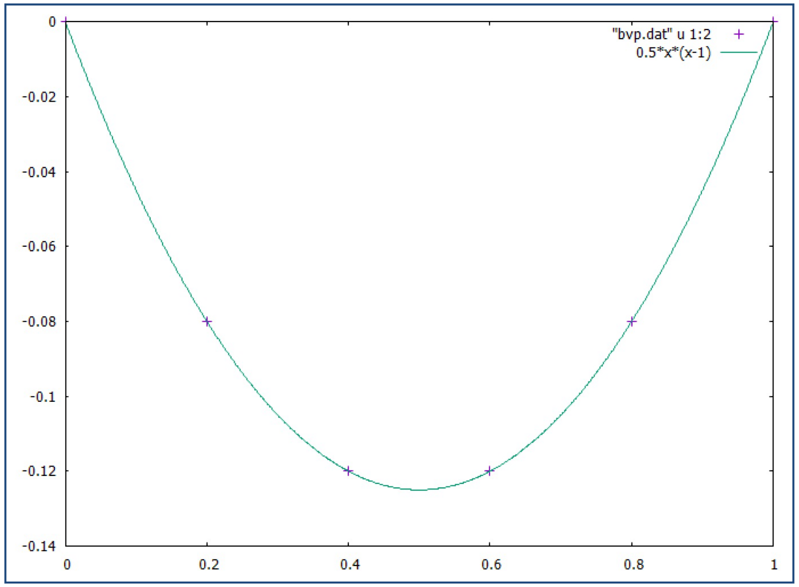
Practice Session #4

- **bvp.cpp**: solve the Boundary Value Problem using finite difference scheme:

$$\frac{d^2y}{dx^2} = 1, \qquad y(0) = 0, \quad y(1) = 0$$

→ Use 32 points (31 intervals) in the range `0 ≤ x ≤ 1.`

→ How does your results compare with the analytical solution ?



- Now change your b.c. and solve

$$\frac{d^2y}{dx^2} = 1, \qquad y(0) = 1, \quad y(1) = 0.9$$