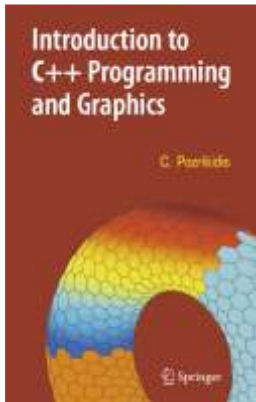


Programming and Numerical Analysis

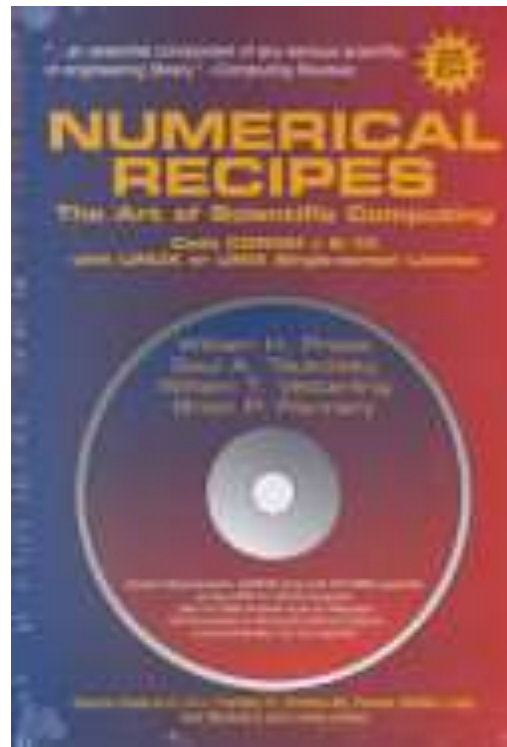
simultaneous linear equations

筒井 広明 / TSUTSUI Hiroaki

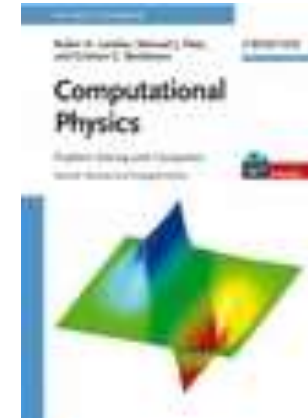
Text Book



Introduction to C++ Programming and Graphics /
Constantine Pozrikidis, Springer **eBooks** Computer
Science, ISBN:
9780387689920 [0387689923],
<https://link.springer.com/10.1007/978-0-387-68993-7>



Numerical recipes in C++ : the art of scientific
computing / William H. Press ... [et al.],
ISBN: 9780521750332 [0521750334]



Computational Physics - Problem Solving with Computers /
Rubin H. Landau, Cristian C. Bordeianu, Manuel José Páez Mejía
Wiley Online Library **Online Books** ISBN: 9783527406265
[3527406263] 9783527618835 [352761883X],
<https://onlinelibrary.wiley.com/book/10.1002/9783527618835>

Matrices

A set of linear algebraic equations looks like this:

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

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

By convention, the first index on an element a_{ij} denotes its row, the second index its column. For most purposes you don't need to know how a matrix is stored in a computer's physical memory; you simply reference matrix elements by their two-dimensional addresses, e.g., $a_{34} = a[3][4]$.

Tasks of Computational Linear Algebra

- Solution of the matrix equation $A \bullet x = b$ for an unknown vector x . where A is a square matrix of coefficients, raised dot denotes matrix multiplication, and b is a known right-hand side vector (§ 2.1- § 2.10).
- Solution of more than one matrix equation $A \bullet x_j = b_j$, for a set of vectors x_j , $j = 1, 2, \dots$, each corresponding to a different, known right-hand side vector b_j . In this task the key simplification is that the matrix A is held constant. while the right-hand sides. the b 's, are changed (§ 2. 1- § 2. 10).
- Calculation of the matrix A^{-1} which is the matrix inverse of a square matrix A , i.e., $A \bullet A^{-1} = A^{-1} \bullet A = \mathbf{1}$, where $\mathbf{1}$ is the identity matrix (all zeros except for ones on the diagonal). This task is equivalent, for an $N \times N$ matrix A , to the previous task with N different b_j 's ($j = 1, 2, \dots, N$), namely the unit vectors ($b_j =$ all zero elements except for 1 in the j th component). The corresponding x 's are then the columns of the matrix inverse of A (s2.1 and s2.3).
- Calculation of the determinant of a square matrix A (§ 2.3).

2. 1 Gauss-Jordan Elimination

- For inverting a matrix, *Gauss-Jordan elimination* is about as efficient as any other method. For solving sets of linear equations, Gauss-Jordan elimination produces both the solution of the equations for one or more right-hand side vectors ***b***, and also the matrix inverse \mathbf{A}^{-1} .
- However, its principal weaknesses are
 - i. that it requires all the right-hand sides to be stored and manipulated at the same time, and
 - ii. that when the inverse matrix is not desired, Gauss-Jordan is three times slower than the best alternative technique for solving a single linear set.

Elimination on Column-Augmented Matrices

Consider the linear matrix equation

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \left[\begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \end{pmatrix} \sqcup \begin{pmatrix} x_{12} \\ x_{22} \\ x_{32} \\ x_{42} \end{pmatrix} \sqcup \begin{pmatrix} x_{13} \\ x_{23} \\ x_{33} \\ x_{43} \end{pmatrix} \sqcup \begin{pmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \\ y_{41} & y_{42} & y_{43} & y_{44} \end{pmatrix} \right]$$

$$= \left[\begin{pmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{pmatrix} \sqcup \begin{pmatrix} b_{12} \\ b_{22} \\ b_{32} \\ b_{42} \end{pmatrix} \sqcup \begin{pmatrix} b_{13} \\ b_{23} \\ b_{33} \\ b_{43} \end{pmatrix} \sqcup \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right] \quad (2.1.1)$$

Here the raised dot (\cdot) signifies matrix multiplication, while the operator \sqcup just signifies column augmentation, that is, removing the abutting parentheses and making a wider matrix out of the operands of the \sqcup operator.

Elimination on Column-Augmented Matrices

In other words, the matrix solution of

$$[\mathbf{A}] \cdot [\mathbf{x}_1 \sqcup \mathbf{x}_2 \sqcup \mathbf{x}_3 \sqcup \mathbf{Y}] = [\mathbf{b}_1 \sqcup \mathbf{b}_2 \sqcup \mathbf{b}_3 \sqcup \mathbf{1}] \quad (2.1.2)$$

where \mathbf{A} and \mathbf{Y} are square matrices, the \mathbf{b}_i 's and \mathbf{x}_i 's are column vectors, and $\mathbf{1}$ is the identity matrix, simultaneously solves the linear sets

$$\mathbf{A} \cdot \mathbf{x}_1 = \mathbf{b}_1 \quad \mathbf{A} \cdot \mathbf{x}_2 = \mathbf{b}_2 \quad \mathbf{A} \cdot \mathbf{x}_3 = \mathbf{b}_3$$

$$\mathbf{A} \cdot \mathbf{Y} = \mathbf{1}$$

Elimination on Column-Augmented Matrices

Now it is also elementary to verify the following facts about (2.1.2):

$$[\mathbf{A}] \cdot [\mathbf{x}_1 \sqcup \mathbf{x}_2 \sqcup \mathbf{x}_3 \sqcup \mathbf{Y}] = [\mathbf{b}_1 \sqcup \mathbf{b}_2 \sqcup \mathbf{b}_3 \sqcup \mathbf{1}] \quad (2.1.2)$$

- Interchanging any two rows of \mathbf{A} and the corresponding rows of the \mathbf{b} 's and of $\mathbf{1}$, does not change (or scramble in any way) the solution \mathbf{x} 's and \mathbf{Y} . 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 \mathbf{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 \mathbf{b} 's and $\mathbf{1}$ (which then is no longer the identity matrix, of course).
- Interchanging any two columns of \mathbf{A} gives the same solution set only if we simultaneously interchange corresponding rows of the \mathbf{x} 's and of \mathbf{Y} . 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.

Gauss-Jordan elimination uses one or more of the above operations to reduce the matrix \mathbf{A} to the identity matrix.

Pivoting

- In "Gauss-Jordan elimination with no pivoting," only the second operation in the above list is used.
 1. The first row is divided by the element a_{11} (this being a trivial linear combination of the first row with any other row - zero coefficient for the other row). Then the right amount of the first row is subtracted from each other row to make all the remaining a_{i1} 's zero. The first column of A now agrees with the identity matrix.
 2. We move to the second column and divide the second row by a_{22} , then subtract the right amount of the second row from rows 1, 3, and 4, so as to make their entries in the second column zero. The second column is now reduced to the identity form.
 3. And so on for the third and fourth columns. As we do these operations to A , we of course also do the corresponding operations to the \mathbf{b} 's and to $\mathbf{1}$ (which by now no longer resembles the identity matrix in any way!).

Pivoting

- Obviously we will run into trouble if we ever encounter a zero element on the (then current) diagonal when we are going to divide by the diagonal element. (The element that we divide by, incidentally, is called the *pivot element* or ***pivot***.)


So what is this magic pivoting? Nothing more than interchanging

- rows (*partial pivoting*) or
- rows and columns (*full pivoting*).

we can choose among elements that are both

- (i) on rows below (or on) the one that is about to be normalized, and also
- (ii) on columns to the right (or on) the columns we are about to eliminate.

Example



$$\left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 2 & 5 & 3 & 0 & 1 & 0 \\ 1 & 0 & 8 & 0 & 0 & 1 \end{array} \right) \xrightarrow[\substack{(\text{ii}) + (-2) \times (\text{i}) \\ (\text{iii}) + (-1) \times (\text{i})}]{\quad} \left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & -3 & -2 & 1 & 0 \\ 0 & -2 & 5 & -1 & 0 & 1 \end{array} \right)$$

$$\xrightarrow{(\text{iii}) + 2 \times (\text{ii})} \left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & -3 & -2 & 1 & 0 \\ 0 & 0 & -1 & -5 & 2 & 1 \end{array} \right)$$

$$\xrightarrow{(\text{iii}) \times (-1)} \left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 1 & -3 & -2 & 1 & 0 \\ 0 & 0 & 1 & 5 & -2 & -1 \end{array} \right)$$

$$\xrightarrow[\substack{(\text{i}) + (-3) \times (\text{iii}) \\ (\text{ii}) + 3 \times (\text{iii})}]{\quad} \left(\begin{array}{ccc|ccc} 1 & 2 & 0 & -14 & 6 & 3 \\ 0 & 1 & 0 & 13 & -5 & -3 \\ 0 & 0 & 1 & 5 & -2 & -1 \end{array} \right)$$

$$\xrightarrow{(\text{i}) + (-2) \times (\text{ii})} \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & -40 & 16 & 9 \\ 0 & 1 & 0 & 13 & -5 & -3 \\ 0 & 0 & 1 & 5 & -2 & -1 \end{array} \right).$$

したがって,

$$A^{-1} = \left(\begin{array}{ccc} -40 & 16 & 9 \\ 13 & -5 & -3 \\ 5 & -2 & -1 \end{array} \right).$$

Gauss-Jordan elimination with full pivoting

```
#include <math.h>
#include "nrutil.h"
#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}
```

```
void gaussj(float **a, int n, float **b, int m)
```

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

Gauss-Jordan elimination with full pivoting

```
{
    int *indxc,*indxr,*ipiv;
    int i,icol,irow,j,k,l,ll;
    float big,dum,pivinv,temp;

    indxc=ivector(1,n);
    indxr=ivector(1,n);
    ipiv=ivector(1,n);
    for (j=1;j<=n;j++) ipiv[j]=0;
    for (i=1;i<=n;i++) {
        big=0.0;
        for (j=1;j<=n;j++)
            if (ipiv[j] != 1)
                for (k=1;k<=n;k++) {
                    if (ipiv[k] == 0) {
                        if (fabs(a[j][k]) >= big) {
                            big=fabs(a[j][k]);
                            irow=j;
                            icol=k;
                        }
                    } else if (ipiv[k] > 1) nrerror("gaussj: Singular Matrix-1");
                }
        ++(ipiv[icol]);
    }
```

The integer arrays `ipiv`, `indxr`, and `indxc` are used for bookkeeping on the pivoting.

This is the main loop over the columns to be reduced.

This is the outer loop of the search for a pivot element.

Gauss-Jordan elimination with full pivoting

We now have the pivot element, so we interchange rows, if needed, to put the pivot element on the diagonal. The columns are not physically interchanged, only relabeled: `indxc[i]`, the column of the i th pivot element, is the i th column that is reduced, while `indxr[i]` is the row in which that pivot element was originally located. If `indxr[i] \neq indxc[i]` there is an implied column interchange. With this form of bookkeeping, the solution b 's will end up in the correct order, and the inverse matrix will be scrambled by columns.

```
if (irow != icol) {  
    for (l=1;l<=n;l++) SWAP(a[irow][l],a[icol][l])  
    for (l=1;l<=m;l++) SWAP(b[irow][l],b[icol][l])  
}
```

```
indxr[i]=irow;  
indxc[i]=icol;
```

We are now ready to divide the pivot row by the pivot element, located at `irow` and `icol`.

```
if (a[icol][icol] == 0.0) nrerror("gaussj: Singular Matrix-2");  
pivinv=1.0/a[icol][icol];  
a[icol][icol]=1.0;  
for (l=1;l<=n;l++) a[icol][l] *= pivinv;  
for (l=1;l<=m;l++) b[icol][l] *= pivinv;
```

Gauss-Jordan elimination with full pivoting

```
for (ll=1;ll<=n;ll++)  
    if (ll != icol) {  
        dum=a[ll][icol];  
        a[ll][icol]=0.0;  
        for (l=1;l<=n;l++) a[ll][l] -= a[icol][l]*dum;  
        for (l=1;l<=m;l++) b[ll][l] -= b[icol][l]*dum;  
    }
```

Next, we reduce the rows...
...except for the pivot one, of course.

```
}
```

This is the end of the main loop over columns of the reduction. It only remains to unscramble the solution in view of the column interchanges. We do this by interchanging pairs of columns in the reverse order that the permutation was built up.

```
for (l=n;l>=1;l--) {  
    if (indx[l] != indxc[l])  
        for (k=1;k<=n;k++)  
            SWAP(a[k][indx[l]],a[k][indxc[l]]);  
}
```

And we are done.

```
free_ivector(ipiv,1,n);  
free_ivector(indxr,1,n);  
free_ivector(indxc,1,n);
```

```
}
```

gaussj.c

Gauss-Jordan matrix inversion and linear equation solution

```
#include <math.h>
#define NRANSI
#include "nrutil.h"
#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}

void gaussj(float **a, int n, float **b, int m)
{
    int *indxc,*indxr,*ipiv;
    int i,icol,irow,j,k,l,ll;
    float big,dum,pivinv,temp;

    indxc=ivector(1,n);
    indxr=ivector(1,n);
    ipiv=ivector(1,n);
```


nrutil.c

The utility file nrutil.c includes functions for allocating (using malloc ()) arbitrary-offset vectors of arbitrary lengths

```
#if defined(__STDC__) || defined(ANSI) || defined(NRANSI) /* ANSI */
```

```
#include <stdio.h>
```

```
#include <stddef.h>
```

```
#include <stdlib.h>
```

```
#define NR_END 1
```

```
#define FREE_ARG char*
```

```
int *ivector(long nl, long nh)
```

```
/* allocate an int vector with subscript range v[nl..nh] */
```

```
{
```

```
    int *v;
```

```
    v=(int *)malloc((size_t) ((nh-nl+1+NR_END)*sizeof(int)));
```

```
    if (!v) nrerror("allocation failure in ivector()");
```

```
    return v-nl+NR_END;
```

Pointer arithmetic

- The following code illustrates the memory layout of a two-dimensional array (matrix):

```
#include <iostream>
using namespace std;
int main()
{
    float A[2][2]={ {1.1, 1.2}, {1.3, 1.4} };
    float * memad1, * memad2, * memad3, * memad4;
    memad1 = &A[0][0];
    memad2 = memad1+1;
    memad3 = memad2+1;
    memad4 = memad3+1;
    cout << memad1 << " " << *memad1 << endl;
    cout << memad2 << " " << *memad2 << endl;
    cout << memad3 << " " << *memad3 << endl;
    cout << memad4 << " " << *memad4 << endl;
    return 0;
}
```



The output of the code is:

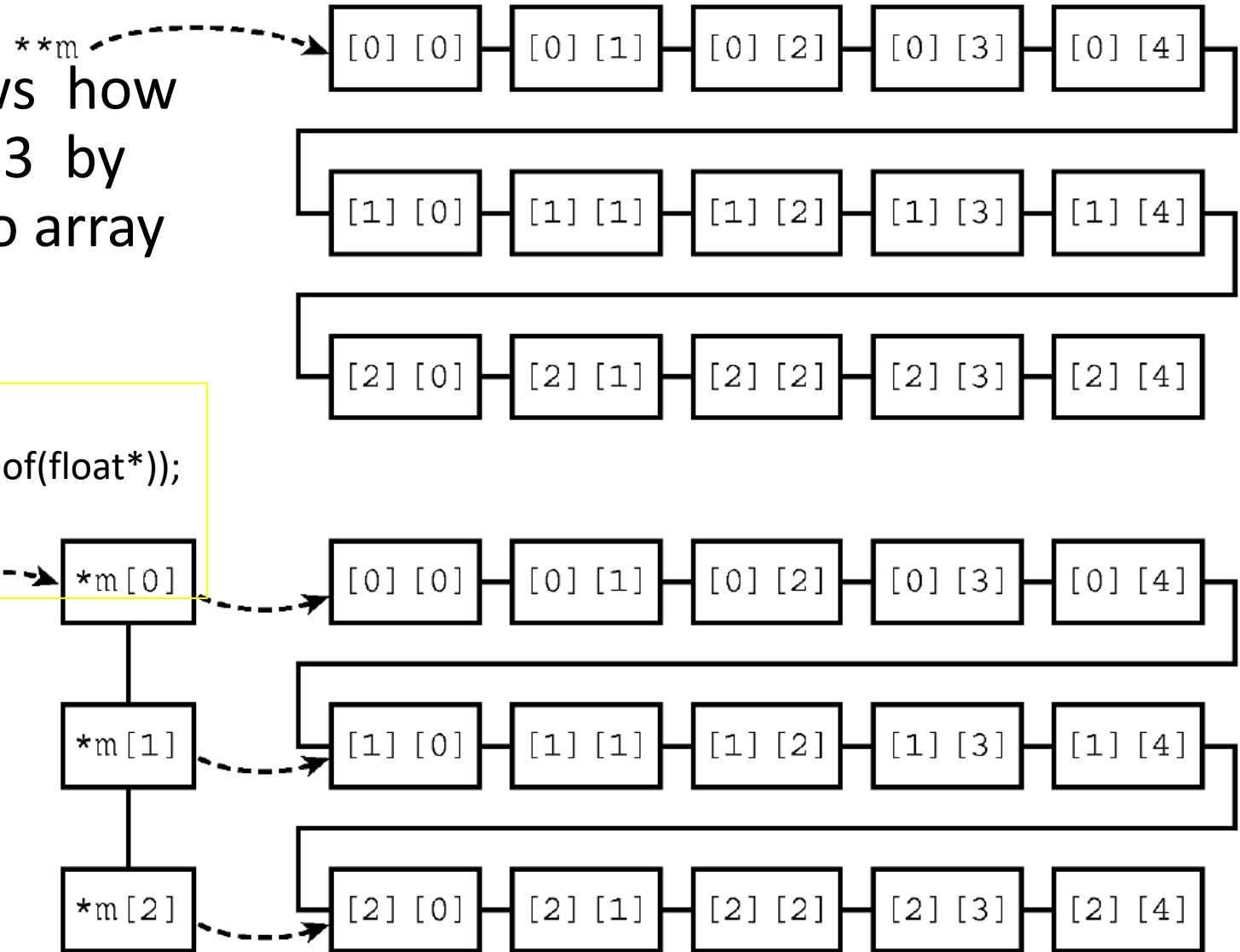
0xbfafdf0	1.1
0xbfafdf4	1.2
0xbfafdf8	1.3
0xbfafdfc	1.4

Matrices and Two-Dimensional Arrays

- The following fragment shows how a fixed-size array `a` of size 13 by 9 is converted to a "pointer to array of pointers" reference `aa`:

```
float a[13][9], **aa; int i;  
aa=(float **) malloc((unsigned) 13*sizeof(float*));  
for(i=0; i<=12; i++) aa[i]=a[i];  
// a[i] is a pointer to a[i][0]
```

The identifier `aa` is now a matrix with index range `aa[0 .. 12][0 .. 8]`. You can use or modify its elements ad lib, and more importantly you can pass it as an argument to any function by its name `aa`. That function, which declares the corresponding dummy argument as `float **aa`, can address its elements as `aa[i][j]` without knowing its physical size.



nrutil.c

- additional utility routines in nrutil.c (Appendix B) which allocate and deallocate matrices of arbitrary range. The synopses are

```
float **matrix(long nrl, long nrh, long ncl, long nch)
// Allocates a float matrix with range [nrl..nrh][ncl..nch].
```

```
void free_matrix(float **m, long nrl, long nrh, long ncl, long nch)
//Frees a matrix allocated with matrix.
```

A typical use is as follows:

```
float **a;
a=matrix(1,13,1,9);

a[3][5]= ...
... +a[2][9]/3.0 ...
someroutine(a, ...);
...
free_matrix(a,1,13,1,9);
```

aa could be addressed with the range
a[1 .. 13][1 .. 9].

Read from a file

- To read from a file named *stresses.dat*, we simply associate the file with a *device* that replaces **cin** of the **iostream**:

```
#include<fstream>
ifstream dev1;
dev1.open("stresses.dat");
dev1 >> variable1 >> variable2;
dev1.close();
```

In compact notation, the lines

```
ifstream dev1;
dev1.open("stresses.dat");
```

can be consolidated into one,

```
ifstream dev1("stresses.dat");
```

which bypasses the explicit use of the open statement.

The first line declares the device *dev1* as a member of the “input file stream.” The second line opens the file through the device, the third line writes to the device, and the fourth line closes the device.

Read from a file

- The implementation of the algorithm is:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream file9("vector.dat");
    int i=1;
    double a[10];
    while(file9 >> a[i])
    {
        cout << i << " " << a[i] << endl;
        i++;
    }
    file9.close();
    return 0;
}
```

If the file `vector.dat` reads:

```
3.4 9.8
3.0 9.1
0.45
```



the output of the code will be:

```
1      3.4
2      9.8
3      3
4      9.1
5      0.45
```

Write to a file

- To write to a file named *post process.dat*, we simply associate the file with a device that replaces **cout** of the **iostream**:

```
#include<fstream>
ofstream dev2;
dev2.open("post process.dat");
dev2 << variable1 << variable2;
dev2 << variable << " " << variable1 << " total" << endl;
dev2.close();
```

The second and third statements can be consolidated into one,
ofstream dev2("post process.dat");

The second line declares the device dev2 as a member of the “output file stream.” The third line opens the device, the fourth line writes to the device, and the fifth line closes the device.

Programming and Numerical Analysis

interpolation and extrapolation

筒井 広明 / TSUTSUI Hiroaki

3.0 Introduction

- We sometimes know the value of a function $f(x)$ at a set of points x_1, x_2, \dots, x_N (say, with $x_1 < \dots < x_N$), but we don't have an analytic expression for $f(x)$ that lets us calculate its value at an arbitrary point. For example, the $f(x_i)$'s might result from some physical measurement or from long numerical calculation that cannot be cast into a simple functional form. Often the x_i 's are equally spaced, but not necessarily.
- The task now is to estimate $f(x)$ for arbitrary x by, in some sense, drawing a smooth curve through (and perhaps beyond) the x_i . If the desired x is in between the largest and smallest of the x_i 's, the problem is called *interpolation*; if x is outside that range, it is called *extrapolation*.
- Interpolation and extrapolation schemes must model the function, between or beyond the known points, by some plausible functional form. The form should be sufficiently general so as to be able to approximate large classes of functions which might arise in practice. By far most common among the functional forms used are polynomials (§ 3.1).

Introduction

- the interpolation process has two stages:
 1. Fit an interpolating function to the data points provided.
 2. Evaluate that interpolating function at the target point x .

However, this two-stage method is generally not the best way to proceed in practice. Typically it is computationally less efficient.

In situations where continuity of derivatives is a concern, one must use the "stiffer" interpolation provided by a so-called spline function. A spline is a polynomial between each pair of table points, but one whose coefficients are determined "slightly" nonlocally. The nonlocality is designed to guarantee global smoothness in the interpolated function up to some order of derivative. Cubic splines (§ 3.3) are the most popular.

Introduction

- The number of points (minus one) used in an interpolation scheme is called the *order* of the interpolation. Increasing the order does not necessarily increase the accuracy, especially in polynomial interpolation.



(a) A smooth function (solid line) is more accurately interpolated by a high-order polynomial (shown schematically as dotted line) than by a low- order polynomial (shown as a piecewise linear dashed line) . (b) A function with sharp corners or rapidly changing higher derivatives is less accurately approximated by a high-order polynomial (dotted line). which is too "stiff," than by a low-order polynomial (dashed lines) . Even some smooth functions, such as exponentials or rational functions, can be badly approximated by high-order polynomials.

3. 1 Polynomial Interpolation and Extrapolation

Through any two points there is a unique line. Through any three points, a unique quadratic. Et cetera. The interpolating polynomial of degree $N - 1$ through the N points $y_1 = f(x_1)$, $y_2 = f(x_2)$, \dots , $y_N = f(x_N)$ is given explicitly by Lagrange's classical formula,

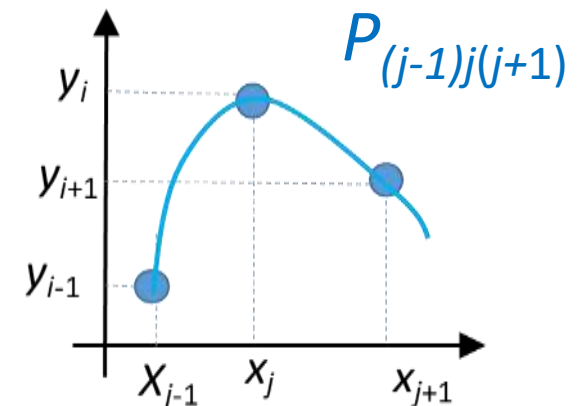
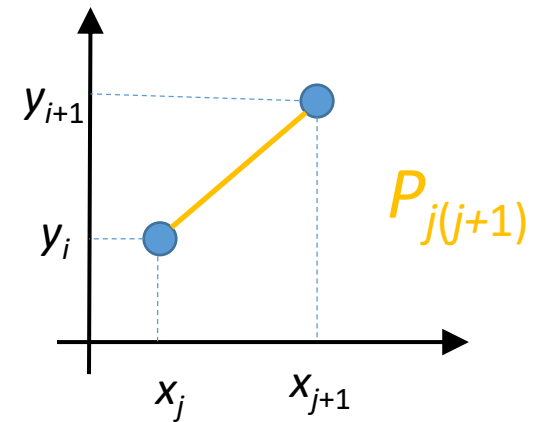
$$P(x) = \frac{(x - x_2)(x - x_3)\dots(x - x_N)}{(x_1 - x_2)(x_1 - x_3)\dots(x_1 - x_N)}y_1 + \frac{(x - x_1)(x - x_3)\dots(x - x_N)}{(x_2 - x_1)(x_2 - x_3)\dots(x_2 - x_N)}y_2 \\ + \dots + \frac{(x - x_1)(x - x_2)\dots(x - x_{N-1})}{(x_N - x_1)(x_N - x_2)\dots(x_N - x_{N-1})}y_N \quad (3.1.1)$$

There are N terms, each a polynomial of degree $N - 1$ and each constructed to be zero at all of the x_i except one, at which it is constructed to be y_i .

Neville's algorithm

- It is not terribly wrong to implement the Lagrange formula straightforwardly, but it is not terribly right either. A much better algorithm (for constructing the same, unique, interpolating polynomial) is Neville's algorithm.

Let P_1 be the value at x of the **unique polynomial of degree zero** (i.e., a constant) passing through the point (x_1, y_1) ; so $P_1 = y_1$. Likewise define P_2, P_3, \dots, P_N . Now let P_{12} be the value at x of the **unique polynomial of degree one** passing through both (x_1, y_1) and (x_2, y_2) . Likewise $P_{23}, P_{34}, \dots, P_{(N-1)N}$. Similarly, for higher-order polynomials, up to $P_{123\dots N}$, which is the value of the unique interpolating polynomial through all N points, i.e., the desired answer.



Neville's algorithm

Neville's algorithm is a recursive way of filling in the numbers in the tableau a column at a time, from left to right. It is based on the relationship between a "daughter" P and its two "parents,"

$$\begin{array}{rcl}
 x_1 : & y_1 = P_1 & \\
 & & P_{12} \\
 x_2 : & y_2 = P_2 & P_{123} \\
 & & P_{23} & P_{1234} \\
 x_3 : & y_3 = P_3 & P_{234} \\
 & & P_{34} \\
 x_4 : & y_4 = P_4 &
 \end{array}$$

$$P_{i(i+1)\dots(i+m)} = \frac{(x - x_{i+m})P_{i(i+1)\dots(i+m-1)} + (x_i - x)P_{(i+1)(i+2)\dots(i+m)}}{x_i - x_{i+m}} \quad (3.1.3)$$

This recurrence works because the two parents already agree at points $x_{i+1} \dots x_{i+m-1}$.

Neville's algorithm

An improvement on the recurrence (3. 1.3) is to keep track of the small *differences* between parents and daughters, namely to define (for $m = 1, 2, \dots, N - 1$),

$$C_{m,i} \equiv P_{i\dots(i+m)} - P_{i\dots(i+m-1)}$$

$$D_{m,i} \equiv P_{i\dots(i+m)} - P_{(i+1)\dots(i+m)}.$$

Then one can easily derive from (3.1 .3) the relations

$$D_{m+1,i} = \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}}$$

$$C_{m+1,i} = \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}}$$

At each level m , the C 's and D 's are the corrections that make the interpolation one order higher. The final answer $P_{1\dots N}$ is equal to the sum of any y_i plus a set of C 's and/or D 's that form a path through the family tree to the rightmost daughter.

3.3 Cubic Spline Interpolation

- Given a tabulated function $y_i = y(x_i)$, $i = 1 \dots N$, focus attention on one particular interval, between x_j and x_{j+1} . Linear interpolation in that interval gives the interpolation formula

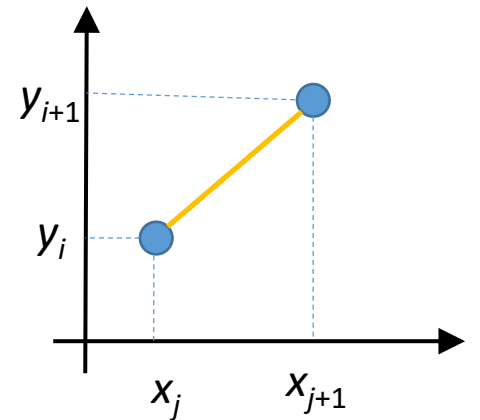
where

$$y = Ay_j + By_{j+1} \tag{3.3.1}$$

$$A \equiv \frac{x_{j+1} - x}{x_{j+1} - x_j} \quad B \equiv 1 - A = \frac{x - x_j}{x_{j+1} - x_j} \tag{3.3.2}$$

Equations (3.3.1) and (3.3.2) are a special case of the general Lagrange interpolation formula (3.1.1).

Since it is (piecewise) linear, equation (3.3.1) has zero second derivative in the interior of each interval, and an undefined, or infinite, second derivative at the abscissas x_j . **The goal of cubic spline interpolation is to get an interpolation formula that is smooth in the first derivative**, and continuous in the second derivative, both within an interval and at its boundaries.



Cubic Spline Interpolation

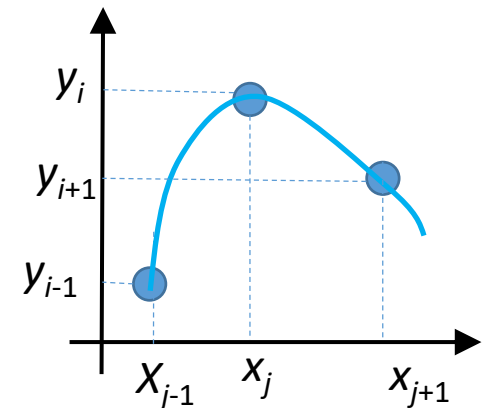
- Suppose, contrary to fact, that in addition to the tabulated values of y_i , we also have tabulated values for the function's second derivatives, y'' , that is, a set of numbers y_i'' .

$$y = Ay_j + By_{j+1} + Cy_j'' + Dy_{j+1}'' \quad (3.3.3)$$

Where A and B are defined in (3.3.2) and

$$C \equiv \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2 \quad D \equiv \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2 \quad (3.3.4)$$

Notice that the dependence on the independent variable x in equations (3.3.3) and (3.3.4) is entirely through the linear x -dependence of A and B , and (through A and B) the cubic x -dependence of C and D .



Cubic Spline Interpolation

- We can readily check that y'' is in fact the second derivative of the new interpolating polynomial.

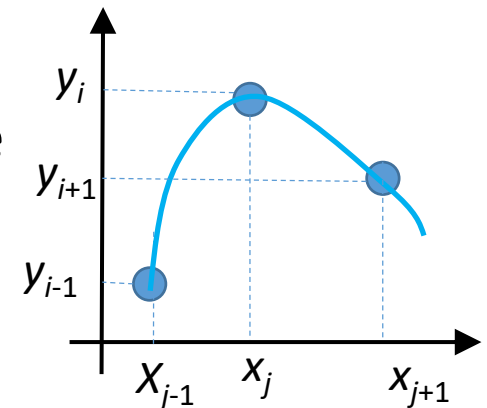
We take derivatives of equation (3.3.3) with respect to x , using the definitions of A , B , C , D to compute dA/dx , dB/dx , dC/dx , and dD/dx . The result is

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y_j'' + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y_{j+1}'' \quad (3.3.5)$$

for the first derivative, and

$$\frac{d^2y}{dx^2} = Ay_j'' + By_{j+1}'' \quad (3.3.6)$$

for the second derivative. Since $A = 1$ at x_j , $A = 0$ at x_{j+1} , while B is just the other way around, (3.3.6) shows that y'' is just the tabulated second derivative, and also that the second derivative will be continuous across (e.g.) the boundary between the two intervals (x_{j-1}, x_j) and (x_j, x_{j+1}) .



Cubic Spline Interpolation

- The only problem now is that we supposed the y''' 's to be known, when, actually, they are not. However, we have not yet required that the *first* derivative, computed from equation (3.3.5), be continuous across the boundary between two intervals. The key idea of a cubic spline is to require this continuity and to use it to get equations for the second derivatives y_i'' .

The required equations are obtained by setting equation (3.3.5) evaluated for $x=x_j$ in the interval (x_{j-1}, x_j) equal to the same equation evaluated for $x = x_j$ but in the interval (x_j, x_{j+1}) . With some rearrangement, this gives (for $j = 2, \dots, N-1$)

$$\frac{x_j - x_{j-1}}{6} y_{j-1}'' + \frac{x_{j+1} - x_{j-1}}{3} y_j'' + \frac{x_{j+1} - x_j}{6} y_{j+1}'' = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{y_j - y_{j-1}}{x_j - x_{j-1}} \quad (3.3.7)$$

These are $N - 2$ linear equations in the N unknowns y_i'' , $i = 1, \dots, N$. Therefore there is a two-parameter family of possible solutions. For a unique solution, we need to specify two further conditions, typically taken as boundary conditions at x_1 and x_N .

Cubic Spline Interpolation

The most common ways of doing this are either

- set one or both of y_1'' and y_N'' equal to zero, giving the so-called *natural cubic spline*, which has zero second derivative on one or both of its boundaries, or
- set either of y_1'' and y_N'' to values calculated from equation (3.3.5) so as to make the first derivative of the interpolating function have a specified value on either or both boundaries.

One reason that cubic splines are especially practical is that the set of equations (3.3.7), along with the two additional boundary conditions, are not only linear, but also *tridiagonal*. Each y_j'' is coupled only to its nearest neighbors at $j \pm 1$. Therefore, the equations can be solved in $O(N)$ operations by the tridiagonal algorithm (§ 2.4).