Class Progress

Basics of Linux, gnuplot, C
Visualization of numerical data
Roots of nonlinear equations
  (Midterm 1)
Solutions of systems of linear equations
Solutions of systems of nonlinear equations
Monte Carlo simulation
Interpolation of sparse data points
Numerical integration
  (Midterm 2)
Solutions of ordinary differential equations

Systems of Linear Equations

$$
\begin{gathered}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1\mathrm{n}}x_n = b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2\mathrm{n}}x_n = b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3\mathrm{n}}x_n = b_3 \\
\vdots \\
a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_n
\end{gathered}
$$

coefficients a_{ij} where $i=$equation index, $j=$variable index

Matrix Formulation

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n-1} & a_{3,n} \\ & & \vdots & & & \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

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

coefficients a_{ij} where $i=$row index, $j=$column index

Reduction to Upper Diagonal Matrix

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ 0 & a_{2,2} & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ 0 & 0 & a_{3,3} & \cdots & a_{3,n-1} & a_{3,n} \\ & & \vdots & & & \\ 0 & 0 & 0 & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

Note: $a_{i,j}$ and b_i are not the same as in original matrix!

Example Elimination Process

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n-1} & a_{3,n} \\ & & \vdots & & & \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

For example, to eliminate a value from the $a_{n,1}$ position

subtract $\left(\dfrac{a_{n,1}}{a_{1,1}}\right)$ times row 1

from row n

Don't forget to subtract

$$\left(\frac{a_{n,1}}{a_{1,1}}\right)\cdot b_1 \text{ from } b_n$$

to keep the solution the same!

General Elimination Case

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n-1} & a_{3,n} \\ & & \vdots & & & \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

In general, for some diagonal pivot element $a_{k,k}$ to eliminate a value from the $a_{i,k}$ position

subtract $(\frac{a_{i,k}}{a_{k,k}})$ times row k

from row i

Don't forget to subtract

$$(\frac{a_{i,k}}{a_{k,k}})\cdot b_k \text{ from } b_i$$

to keep the solution the same!

C Arithmetic Modify Operators

`x = x + 1.0;` Can be written `x += 1.0;`

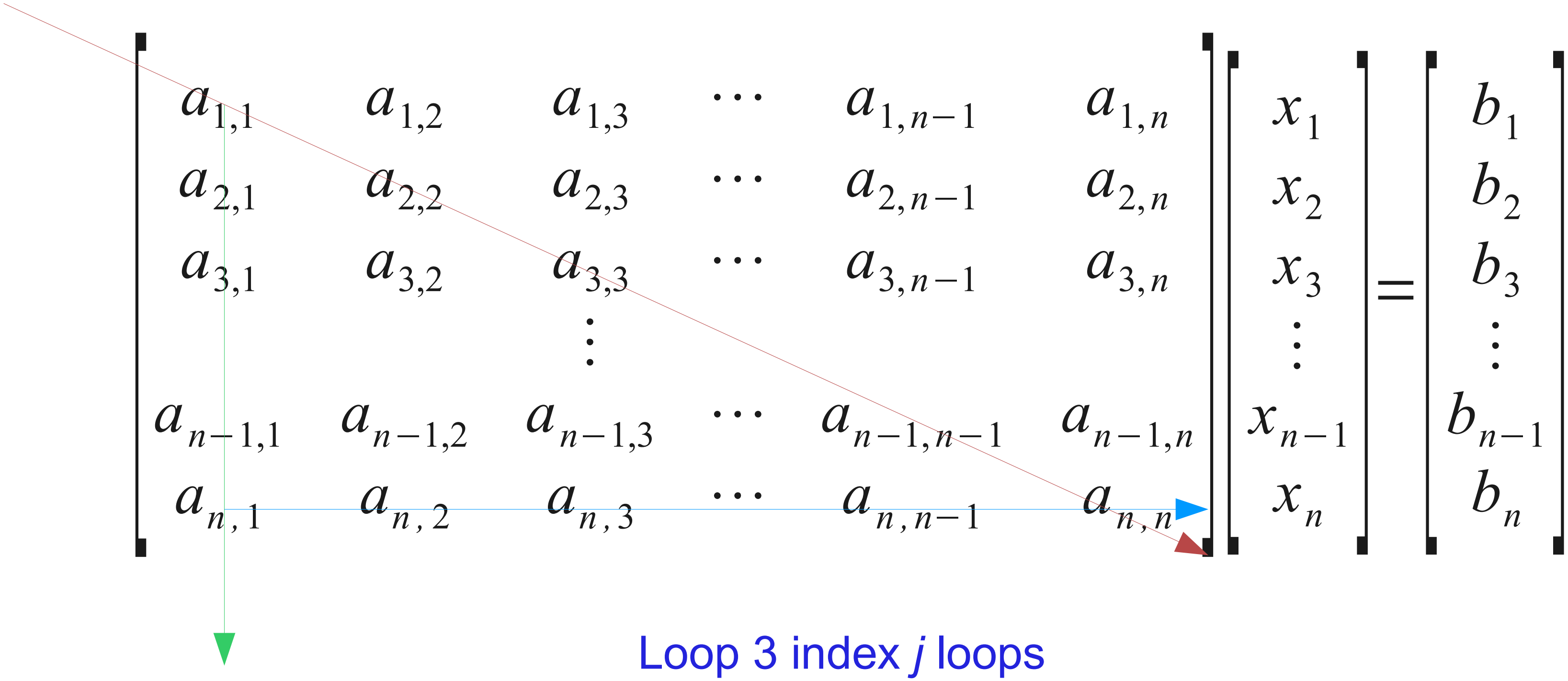
`i = i - 13;` Can be written `i -= 13;`

`a = a * x;` Can be written `a *= x;`

`p = p  / 5.0;` Can be written `p /= 5.0;`

Algorithm Loop Index Plan

Loop 1 index *k* loops down diagonal pivot elements



$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n-1} & a_{3,n} \\ & & \vdots & & & \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

Loop 2 index *i* loops down pivot column

Loop 3 index *j* loops across elimination row

C Code for Forward Elimination

Indices in C start at 0!

```
k = 0;
while (k < n - 1) { /*loop down diagonal pivot elements*/
  i = k + 1;
  while (i < n) { /*loop down rows under pivot row*/
    j = k;
    while (j < n) { /*loop over columns to the right of pivot column*/
      a[i][j] -= a[k][j] * a[i][k] / a[k][k];
      j++;
    }
    b[i] -= b[k] * a[i][k] / a[k][k];
    i++;
  }
  k++;
}
```

We don't really have to write all those zeroes into the matrix, as they are never read back again

Pull out a common factor of *a*[i][k]/*a*[k][k] which doesn't depend on *j* and save it

Improved C Code for Forward Elimination

```
k = 0;
while (k < n - 1) { /*loop down diagonal pivot elements*/
  i = k + 1;
  while (i < n) { /*loop down rows under pivot row*/
    factor = a[i][k] / a[k][k];
    j = k + 1;
    while (j < n) { /*loop over columns to the right of pivot column*/
      a[i][j] -= a[k][j] * factor;
      j++;
    }
    b[i] -= b[k] * factor;
    i++;
  }
  k++;
}
```

Back Substitution to Find x_i

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ 0 & a_{2,2} & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ 0 & 0 & a_{3,3} & \cdots & a_{3,n-1} & a_{3,n} \\ & & \vdots & & & \\ 0 & 0 & 0 & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

$$x_n = \frac{b_n}{a_{n,n}}$$

$$x_{n-1} = \frac{b_{n-1} - a_{n-1,n} x_n}{a_{n-1,n-1}}$$

$$\vdots$$

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{i,j} x_j}{a_{i,i}}$$

C Code for Backward Substitution

```
i = n - 1;
while (i >= 0) {   /*loop up rows for back substitution*/
  sum = 0.0;
  j = i + 1;
  while (j < n) {   /*loop over columns for summation*/
    sum += a[i][j] * x[j];
    j++;
  }
  x[i] = (b[i] - sum)/ a[i][i];
  i--;
}
```

Native Two Dimensional Arrays in C

The declaration

```
double a[4][6];
```

would allocate a linear array of 24 double precision floating point variables. The symbol `a` gets defined as type 'pointer to array of doubles'.

The array elements are indexed with two indices in separate [ ] brackets. The convention in C is that consecutive values in the second index are stored in consecutive memory locations. In other words, the sequence of the 24 array elements in consecutive 8-byte memory locations would be

```
a[0][0] a[0][1] a[0][2] a[0][3] a[0][4] a[0][5] a[1][0] a[1][1] a[1][2]
a[1][3] a[1][4] a[1][5] a[2][0] a[2][1] a[2][2] a[2][3] a[2][4] a[2][5]
a[3][0] a[3][1] a[3][2] a[3][3] a[3][4] a[3][5]
```

If the array indexing is used in the conventional way so that `a[i][j]` refers to an element in the ith row and the jth column, then C stores arrays “by row”, that is, rows are in consecutive memory locations.

To locate an array element from variable indices, standard C uses the declared (and fixed) array size and computes the memory location. The memory address of `a[i][j]` would be computed as `(6*i + j) * sizeof(double)` relative to the first element. This computation can take significant processor time for complex algorithms.

Fast, Variable Sized Two Dimensional Arrays in C

Instead of native two dimensional arrays, first allocate an array of pointers to double that will point to each row

```
double *a[4];
```

Then allocate one dimensional arrays that will hold the values in each row

```
double row0[6],row1[6],row2[6],row3[6];
```

and initialize row pointers

```
a[0] = row0;
a[1] = row1;
a[2] = row2;
a[3] = row3;
```

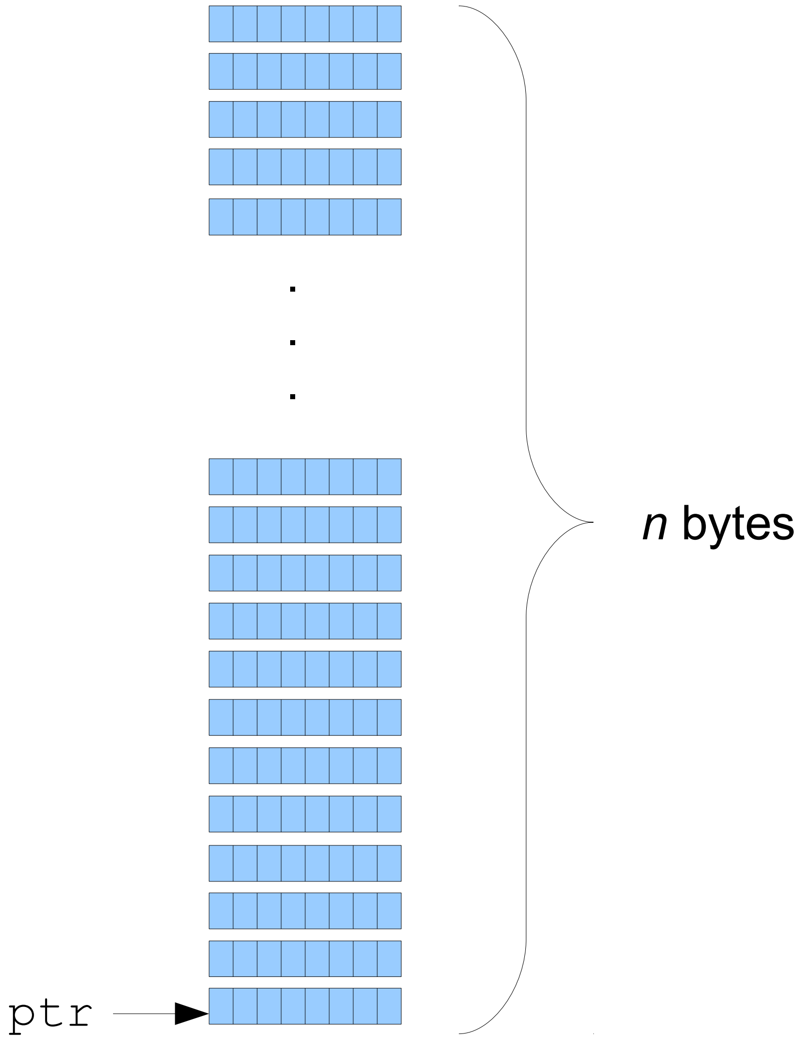
Array elements are still referred to as `a[i][j]`, but the meaning is different in the executable code. The first set of [ ] will index the array of pointers to a row, and the second set of [ ] will index the corresponding row array. The row array indexing can be performed with binary shifts instead of full multiplies since the number of bytes in a double is a power of 2. No pointer arithmetic is needed, and execution is faster. In addition, memory allocation can be done at run time for variable sized arrays.

Run Time Allocation of Data Memory

Asking the operating system to allocate a block of memory for use by your program

Allocation of Programmable Lengths of RAM

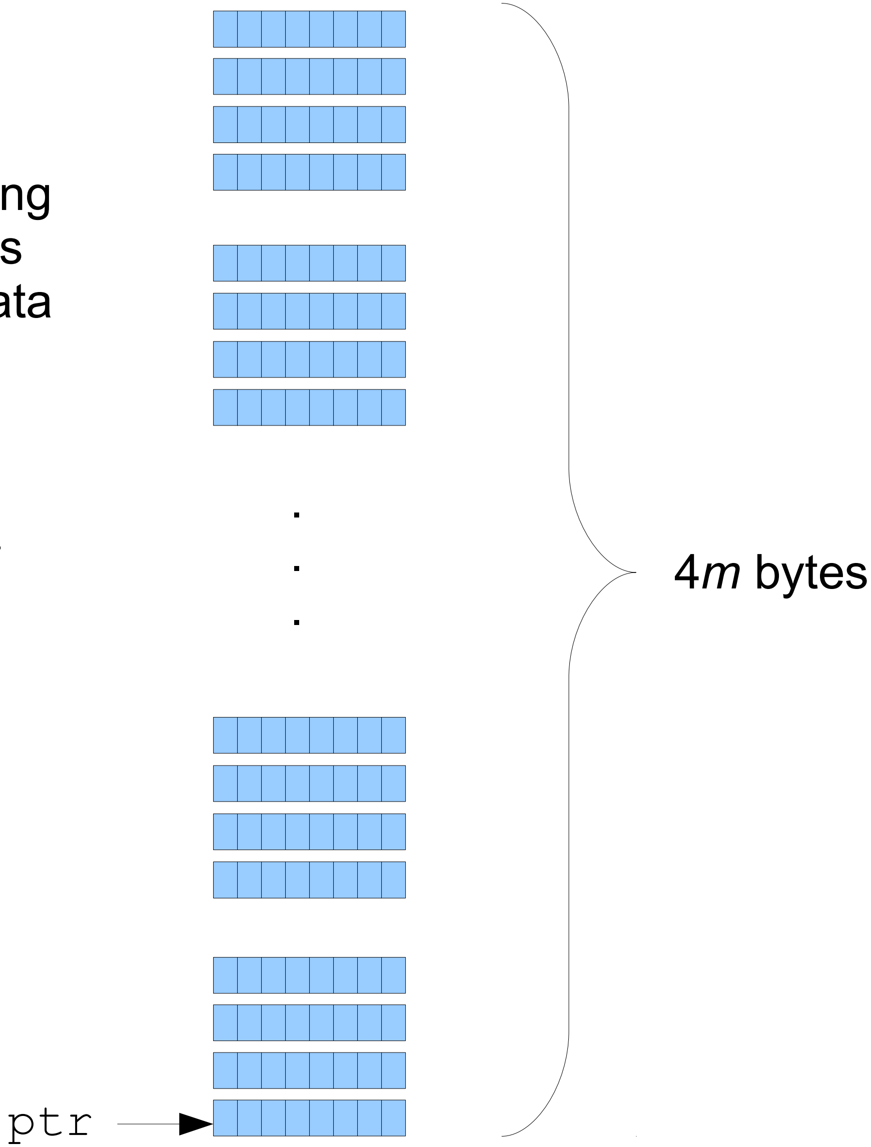
```
ptr = malloc(n);
```



Allocation of Programmable Lengths of RAM

Almost always allocate memory allowing the compiler to decide how many bytes to allocate, based on the number of data items needed and their size

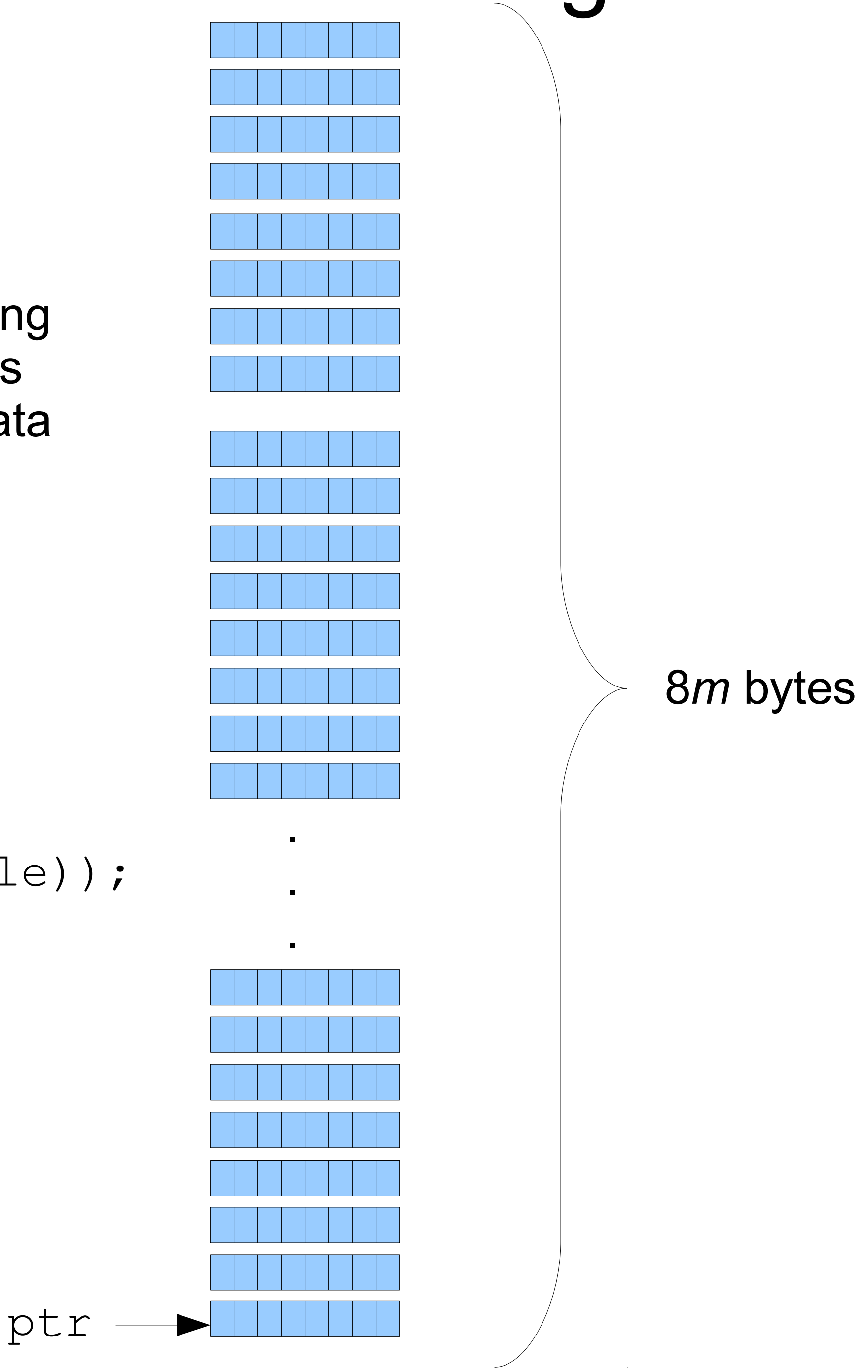
```
ptr = malloc(m * sizeof(int));
```



Allocation of Programmable Lengths of RAM

Almost always allocate memory allowing the compiler to decide how many bytes to allocate, based on the number of data items needed and their size

```
ptr = malloc(m * sizeof(double));
```



Memory Allocation in Programs

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

double x1 = 1.23;
int j = 42;

double func(double t) {
  double y;
  int i;
    ...
  return(y);
}

int main(int argc,char *argv[]) {
  double y,a[6],*ptr;
  int i;
    ...
  ptr = malloc(i * sizeof(double));
    ...
  free(ptr);
    ...
  exit(0);
}
```

Global variables, optionally with initializations, allocated in executable file at compile and link time

Local variables within program blocks allocated with fixed size at run time

Dynamic allocation with variable size at run time

Free allocated memory when done with it

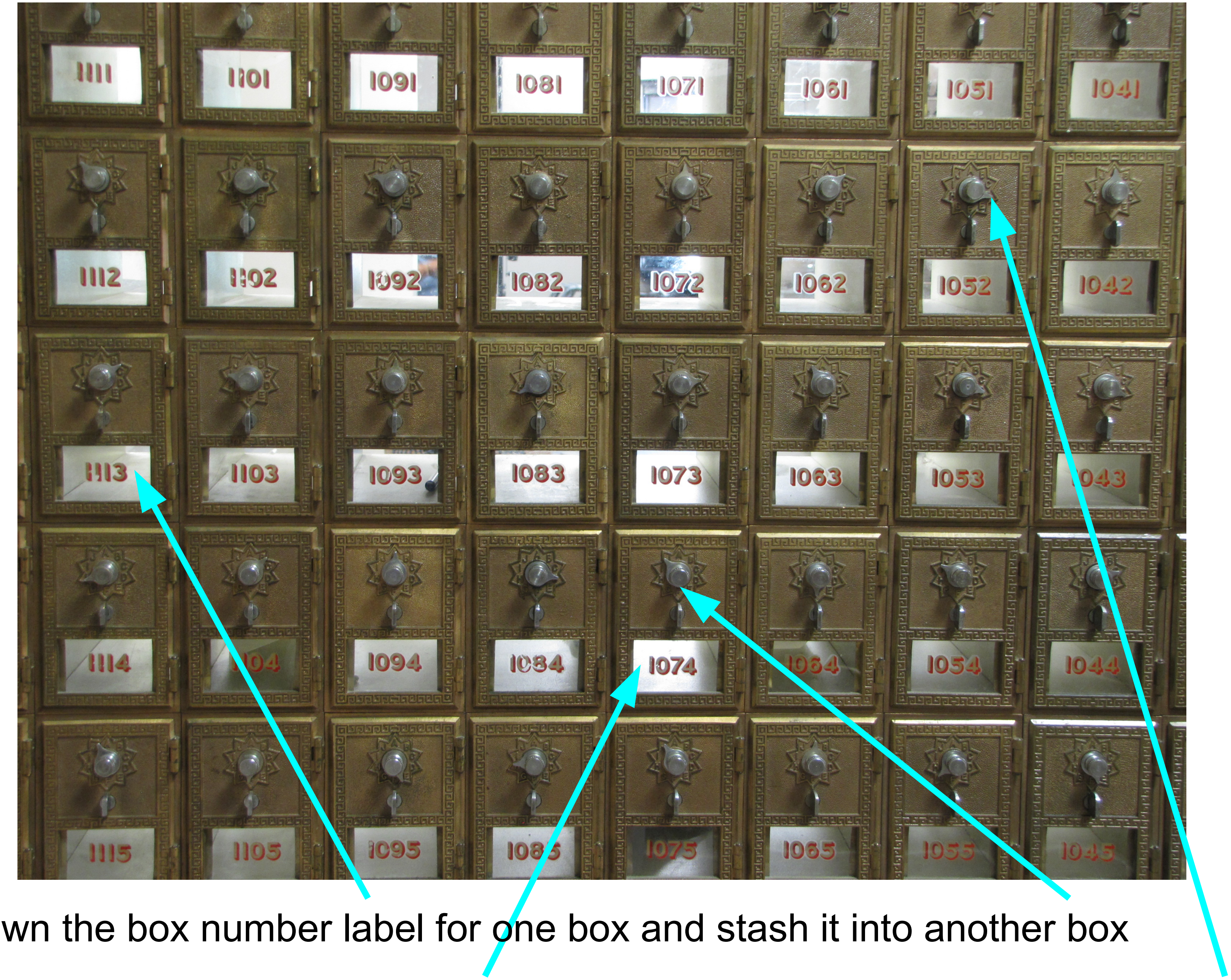
Demonstration Program for malloc()

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

int main(int argc,char *argv[]) {
  double x,*ptr;
  int size,count,i;

  if (argc != 3) {
    fprintf(stderr,"%s <malloc size> <count stored>\n",argv[0]);
    exit(1);
  }
  size = atoi(argv[1]);
  ptr = malloc(size * sizeof(double));
  count = atoi(argv[2]);
  x = 0.0;
  i = 0;
  while (i < count) {
    ptr[i] = x;
    x += 1.0;
    i++;
  }
  free(ptr);
  exit(0);
}
```

Running malloc() Demonstration Program

```
$ ./malloc 1000 1000
$ ./malloc 1000 999
$ ./malloc 1000 1001
*** glibc detected *** ./malloc: double free or corruption (!prev): 0x08b51008 ***
======= Backtrace: =========
/lib/libc.so.6[0xb5ffb6]
./malloc[0x8048561]
/lib/libc.so.6(__libc_start_main+0xe6)[0xb07e36]
./malloc[0x8048411]
======= Memory map: ========
00ad0000-00aed000 r-xp 00000000 08:02 551220     /lib/ld-2.13.so
00aed000-00aee000 r--p 0001c000 08:02 551220     /lib/ld-2.13.so
00aee000-00aef000 rw-p 0001d000 08:02 551220     /lib/ld-2.13.so
00af1000-00c74000 r-xp 00000000 08:02 551225     /lib/libc-2.13.so
00c74000-00c75000 ---p 00183000 08:02 551225     /lib/libc-2.13.so
00c75000-00c77000 r--p 00183000 08:02 551225     /lib/libc-2.13.so
00c77000-00c78000 rw-p 00185000 08:02 551225     /lib/libc-2.13.so
00c78000-00c7b000 rw-p 00000000 00:00 0
00c95000-00c96000 r-xp 00000000 00:00 0          [vdso]
00ce4000-00d00000 r-xp 00000000 08:02 551253     /lib/libgcc_s-4.5.1-20100924.so.1
00d00000-00d01000 rw-p 0001b000 08:02 551253     /lib/libgcc_s-4.5.1-20100924.so.1
08048000-08049000 r-xp 00000000 08:06 37396552   /home/a0179794/data/SMU/C/Phys3340/malloc
08049000-0804a000 rw-p 00000000 08:06 37396552   /home/a0179794/data/SMU/C/Phys3340/malloc
08b51000-08b73000 rw-p 00000000 00:00 0          [heap]
b7891000-b7892000 rw-p 00000000 00:00 0
b78ae000-b78af000 rw-p 00000000 00:00 0
bfd84000-bfda5000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
$
```

Image for C Pointers to Pointers



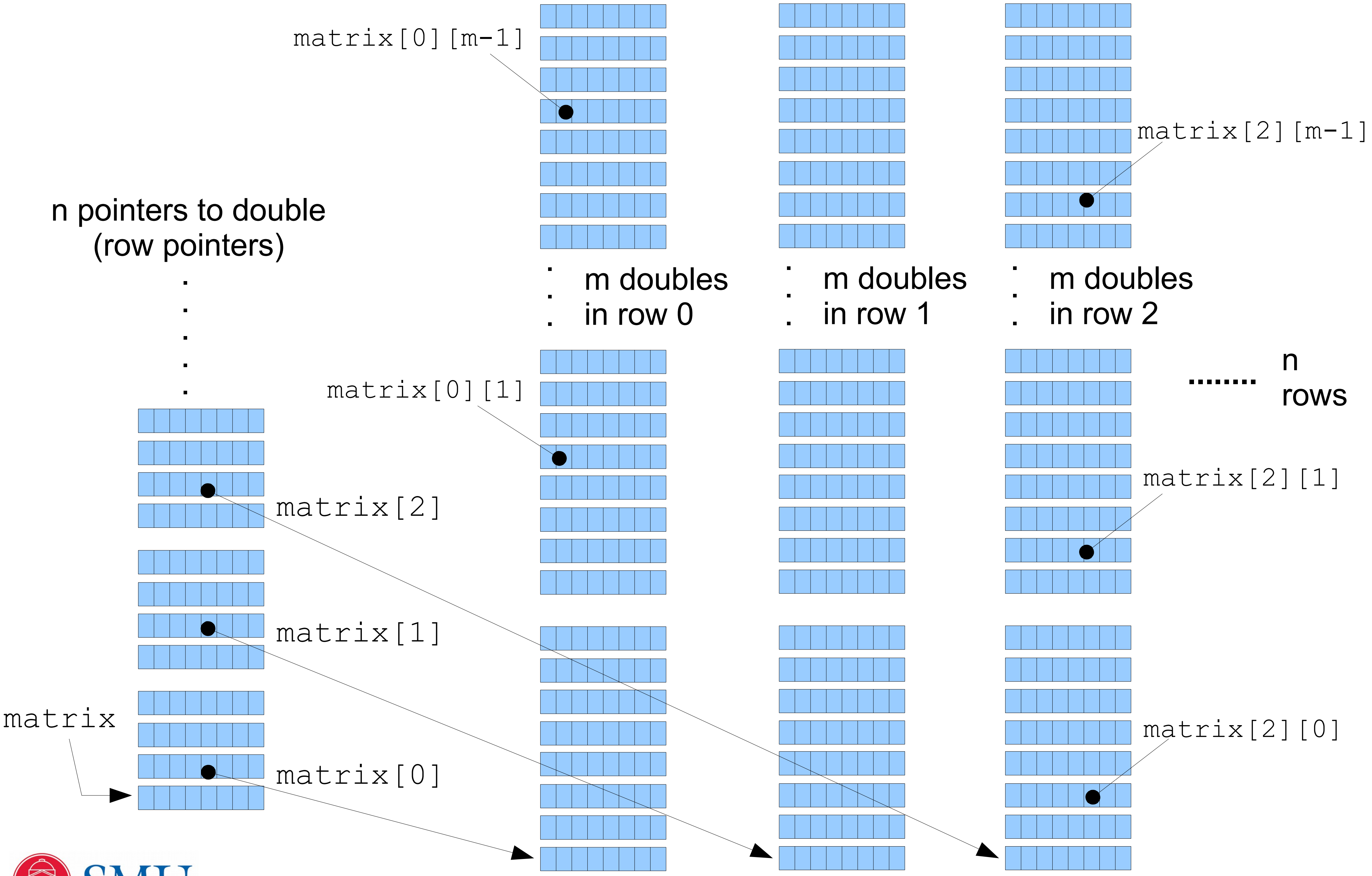
Write down the box number label for one box and stash it into another box

Then write down the box number label for the second box and stash it into another box

C Functions for Variable Arrays

```c
double **alloc_matrix(int n,int m) {
  int i;
  double **matrix;

  matrix = (double **) malloc(n * sizeof(double *));
  if (matrix == NULL) return(NULL);
  i = 0;
  while (i < n) {
    matrix[i] = (double *) malloc(m * sizeof(double));
    i++;
  }
  return(matrix);
}

void free_matrix(int n,double **matrix) {
  int i;
  i = 0;
  while (i < n) {
    if (matrix[i]) free(matrix[i]);
    i++;
  }
  free(matrix);
  return;
}
```

Variable Array Memory Allocation

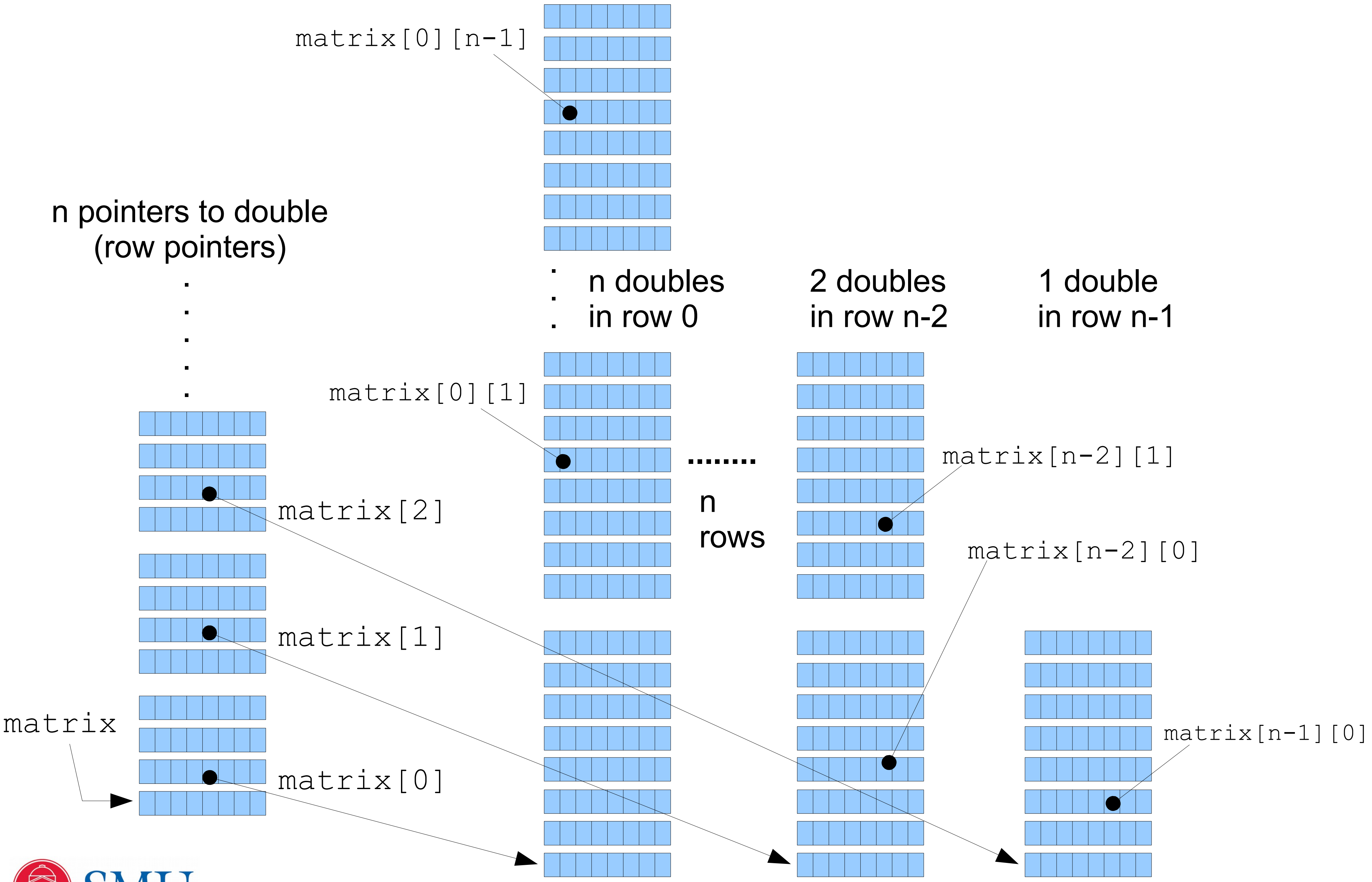


C Function for Triangular Variable Arrays

```
double **alloc_tri_upper(int n) {
  int i;
  double **matrix;

  matrix = (double **) malloc(n * sizeof(double *));
  if (matrix == NULL) return(NULL);
  i = 0;
  while (i < n) {
    matrix[i] = (double *) malloc((n-i) * sizeof(double));
    i++;
  }
  return(matrix);
}

double **alloc_tri_lower(int n) {
  int i;
  double **matrix;

  matrix = (double **) malloc(n * sizeof(double *));
  if (matrix == NULL) return(NULL);
  i = 0;
  while (i < n) {
    matrix[i] = (double *) malloc((i+1) * sizeof(double));
    i++;
  }
  return(matrix);
}
```

For triangular workspaces needed for some algorithms, the allocation function can be modified for different lengths of each row

Upper Triangular Array Memory Allocation



Calling gauss() Function in lineq.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "lineq.h"

int main(int argc,char *argv[]) {
  double **a,*rhs,*x;
  int i,n;

  n = ...;
  a = alloc_matrix(n,n);
/* Initialize coefficient matrix a[i][j] = ... */
  rhs = (double *) malloc(n * sizeof(double));
/* Initialize RHS vector rhs[i] = ... */
  x = (double *) malloc(n * sizeof(double));
  if (gauss(n,a,rhs,x,1.0e-18)) {
    fprintf(stderr,"Singular coefficient matrix\n");
    exit(1);
  }
/* Process solution found in x[i] */
  free(x);
  free_matrix(n,a);
  free(rhs);
  exit(0);
}
```

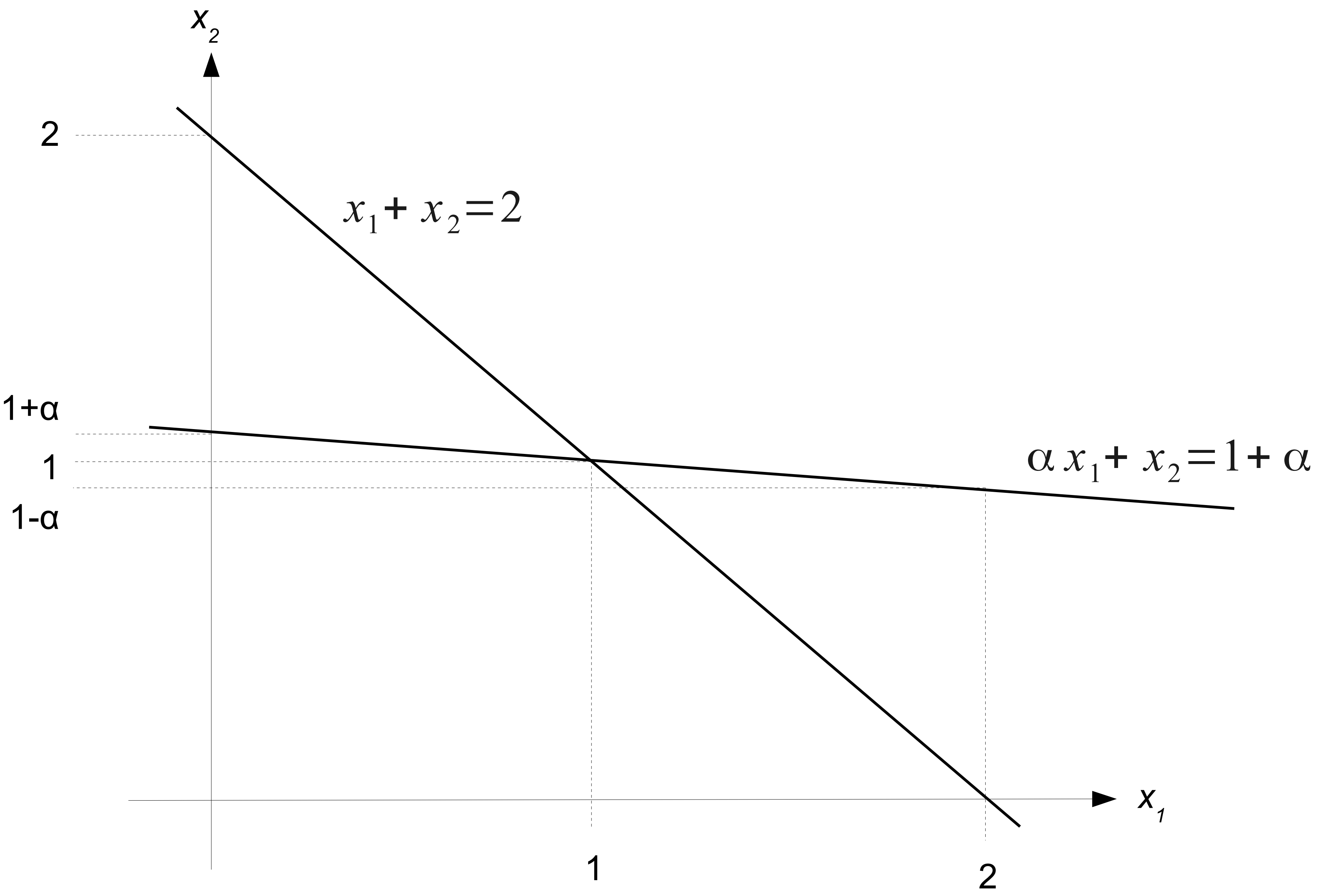
A Look at Error Susceptibility

$$\begin{bmatrix} \alpha & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad \text{for } \alpha \ll 1$$

Set desired solution $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

So system is $\begin{bmatrix} \alpha & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1+\alpha \\ 2 \end{bmatrix}$ for $\alpha \ll 1$

Graphical Look at the Trial System



Actual Computed Solutions

$$\alpha=10^{-1} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha=10^{-2} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\vdots \qquad \vdots$$

$$\alpha=10^{-7} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha=10^{-8} \quad \begin{bmatrix} 0.99999999 \\ 1 \end{bmatrix}$$

$$\alpha=10^{-9} \quad \begin{bmatrix} 0.99999986 \\ 1 \end{bmatrix}$$

$$\alpha=10^{-10} \quad \begin{bmatrix} 1.0000001 \\ 1 \end{bmatrix}$$

$$\alpha=10^{-11} \quad \begin{bmatrix} 1.0000001 \\ 1 \end{bmatrix}$$

$$\alpha=10^{-12} \quad \begin{bmatrix} 0.99986686 \\ 1 \end{bmatrix}$$

$$\alpha=10^{-13} \quad \begin{bmatrix} 0.99920072 \\ 1 \end{bmatrix}$$

$$\alpha=10^{-14} \quad \begin{bmatrix} 0.99920072 \\ 1 \end{bmatrix}$$

$$\alpha=10^{-15} \quad \begin{bmatrix} 0.88817842 \\ 1 \end{bmatrix}$$

Reorder Equations in System

System with reordered rows is $\begin{bmatrix} 1 & 1 \\ \alpha & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1+\alpha \end{bmatrix}$ for $\alpha \ll 1$

Known solution is still $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

Computed Solutions with Reordered Rows

$$\alpha = 10^{-1} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-2} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\vdots \qquad \vdots$$

$$\alpha = 10^{-7} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-8} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-9} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-10} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-11} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-12} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-13} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-14} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-15} \quad \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

A Look at Error Susceptibility

Set desired solution of a forth order system to $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$

So one such system is $\begin{bmatrix} \alpha & 1 & 1 & 1 \\ 2 & 0.5 & -1.5 & 1 \\ -1.5 & -0.5 & 1.5 & -0.5 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 3+\alpha \\ 2 \\ -1 \\ 4 \end{bmatrix}$ for $\alpha \ll 1$

Actual Computed Solutions

$$\alpha = 10^{-1} \quad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-2} \quad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\vdots \qquad \vdots$$

$$\alpha = 10^{-7} \quad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-8} \quad \begin{bmatrix} 0.99999999 \\ 0.99999997 \\ 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-9} \quad \begin{bmatrix} 1.0000001 \\ 0.99999965 \\ 1.0000001 \\ 1.0000002 \end{bmatrix}$$

$$\alpha = 10^{-10} \quad \begin{bmatrix} 1.0000001 \\ 0.99999996 \\ 0.99999996 \\ 1.0000001 \end{bmatrix}$$

$$\alpha = 10^{-11} \quad \begin{bmatrix} 1.0000001 \\ 1.0000278 \\ 0.9999944 \\ 0.99997783 \end{bmatrix}$$

Reorder Rows in System

$$\text{So one such system is} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 0.5 & -1.5 & 1 \\ -1.5 & -0.5 & 1.5 & -0.5 \\ \alpha & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ -1 \\ 3+\alpha \end{bmatrix} \text{ for } \alpha \ll 1$$

$$\text{Solution is still} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Computed Solutions with Reordered Matrix

$$\alpha = 10^{-1} \quad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-2} \quad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\vdots \qquad \vdots$$

$$\alpha = 10^{-7} \quad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-8} \quad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-9} \quad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-10} \quad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\alpha = 10^{-11} \quad \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Reordering Rows in Software

Use an additional array of integers named rindex to store the index of the actual sequence of rows to use in the elimination process.

Actual row index into matrix is rindex[i] instead of i

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n-1} & a_{2,n} \\ \xrightarrow{} a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n-1} & a_{3,n} \\ & & \vdots & & & \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n-1} & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

Reorder rows by simply swapping elements of rindex[ ] array

Save significant computing time that would be needed to swap stored coefficients with large matrices

Don't forget to use rindex[ ] when extracting solution vector too!

Code with Pivoting

```
scale = (double *) malloc(n * sizeof(double));
i = 0;
while (i < n) {
  rindex[i] = i;   /*initialize pivot row indices*/
  scale[i] = fabs(a[i][0]);   /*find maximum row coefficient*/
  j = 1;
  while (j < n) {
    coeffabs = fabs(a[i][j]);
    if (coeffabs > scale[i]) scale[i] = coeffabs;
    j++;
  }
  if (scale[i] < min) {
    free(scale);
    return(1);
  }
  i++;
}
```

Code with Pivoting (continued)

```
  k = 0;
  while (k < n - 1) {   /*loop over pivot columns*/
    imax = k; /*initialize search for maximum candidate pivot elements*/
    coeffmax = fabs(a[rindex[k]][k]) / scale[rindex[k]];
    i = k + 1;
    while (i < n) {   /*loop over remaining unpivoted rows to find
maximum remaining coefficient for pivot*/
      coeffabs = fabs(a[rindex[i]][k]) / scale[rindex[i]];
      if (coeffabs > coeffmax) {
        coeffmax = coeffabs;
        imax = i;
      }
      i++;
    }
    if (coeffmax < min) {
      free(scale);
      return(1);
    }
```

Code with Pivoting (continued)

```
  if (imax != k) {   /*swap in best pivot row if necessary*/
    temp = rindex[imax];
    rindex[imax] = rindex[k];
    rindex[k] = temp;
  }
  i = k + 1;
  while (i < n) { /*loop down rows under pivot row*/
    factor = a[rindex[i]][k] / a[rindex[k]][k];
    j = k + 1;
    while (j < n) { /*loop over columns to the right of pivot column*/
      a[rindex[i]][j] -= factor * a[rindex[k]][j];
      j++;
    }
    b[rindex[i]] -= factor * b[rindex[k]];
    i++;
  }
  k++;
}
```

Code with Pivoting (continued)

```
i = n - 1;
while (i >= 0) {    /*loop up rows for back substitution*/
  if (fabs(a[rindex[i]][i]) < min) {
    free(scale);
    return(1);
  }
  sum = 0.0;
  j = i + 1;
  while (j < n) {    /*loop over columns for summation*/
    sum += a[rindex[i]][j] * x[rindex[j]];
    j++;
  }
  x[rindex[i]] = (b[rindex[i]] - sum)/ a[rindex[i]][i];
  i--;
}
free(scale);
```

Code with Pivoting (continued)

```
if (gauss_pivotmax(n,a,rhs,x,rindex,1.0e-30)) {
  fprintf(stderr,"Singular coefficient matrix\n");
  exit(1);
}
i = 0;
while (i < n) {
  printf("%.12g\n",x[rindex[i]]);
  i++;
}
```

Don't forget to use the rindex[ ] array when extracting the solution vector as well

Calling gauss_pivotmax() Function in lineq.c

```
#include <stdio.h>
#include <stdlib.h>
#include "lineq.h"

int main(int argc,char *argv[]) {
  double **a,*rhs,*x;
  int i,n,*rindex;

  n = ...;
  a = alloc_matrix(n,n);
/* Initialize coefficient matrix a[i][j] = ... */
  rhs = (double *) malloc(n * sizeof(double));
/* Initialize RHS vector rhs[i] = ... */
  x = (double *) malloc(n * sizeof(double));
  rindex = (int *) malloc(n * sizeof(int));
  if (gauss_pivotmax(n,a,rhs,x,rindex,1.0e-18)) {
    fprintf(stderr,"Singular coefficient matrix\n");
    exit(1);
  }
/* Process solution found in x[rindex[i]] */
  free(x);
  free_matrix(n,a);
  free(rhs);
  exit(0);
}
```

Add rindex allocation,
but gauss_pivotmax()
will initialize it

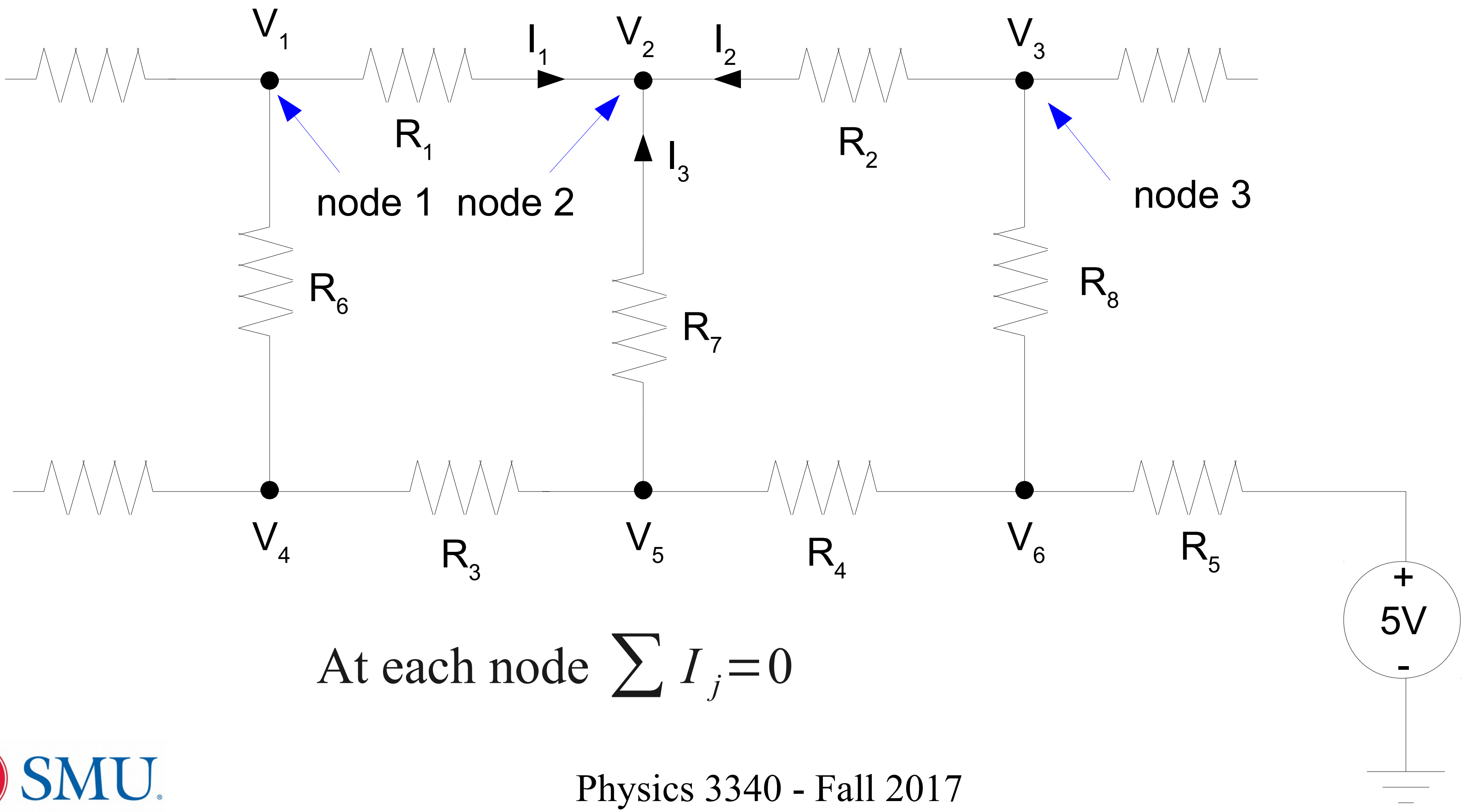
Kirchoff's Current Law

Electrical potential at node *i* is V_i relative to a reference node (ground)

The sum of all currents flowing into each network node = 0

The *j*th element has current I_j flowing through it and V_j potential drop across it, where V_j is the difference between two node potentials, or between a source voltage and a node voltage

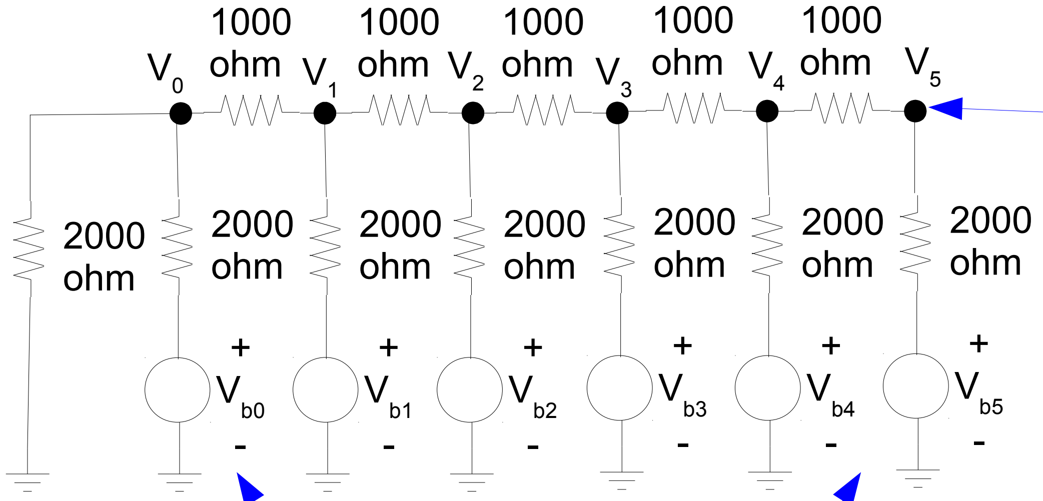
If the relationships between I_j and V_j for each circuit element are linear (the elements are linear resistors), the node voltages V_i can be found by solving a system of linear equations



At each node $\sum I_j = 0$

Six Bit Digital to Analog Converter

Solve the circuit by setting up a system of six linear KCL equations in the six unknown voltages V_0 through V_5 at the nodes shown



Analog output voltage is V_5

$V_5 = D_{in} \cdot 5V / 64$

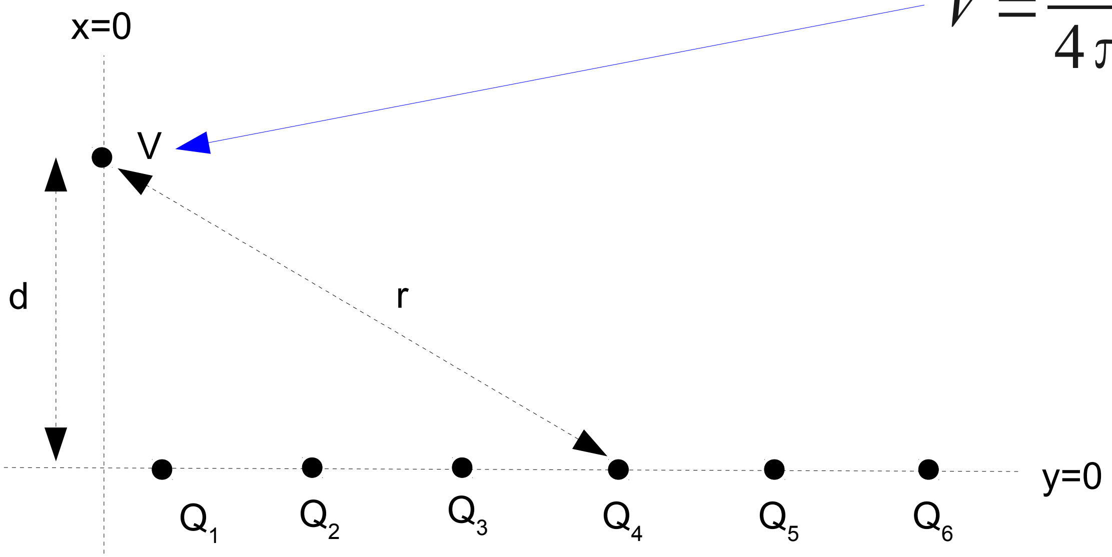
Assume a reference potential of 0V at the “ground” node

The digital input bits for 2^0 through 2^5 are represented by the six voltage sources V_{b0} through V_{b5}

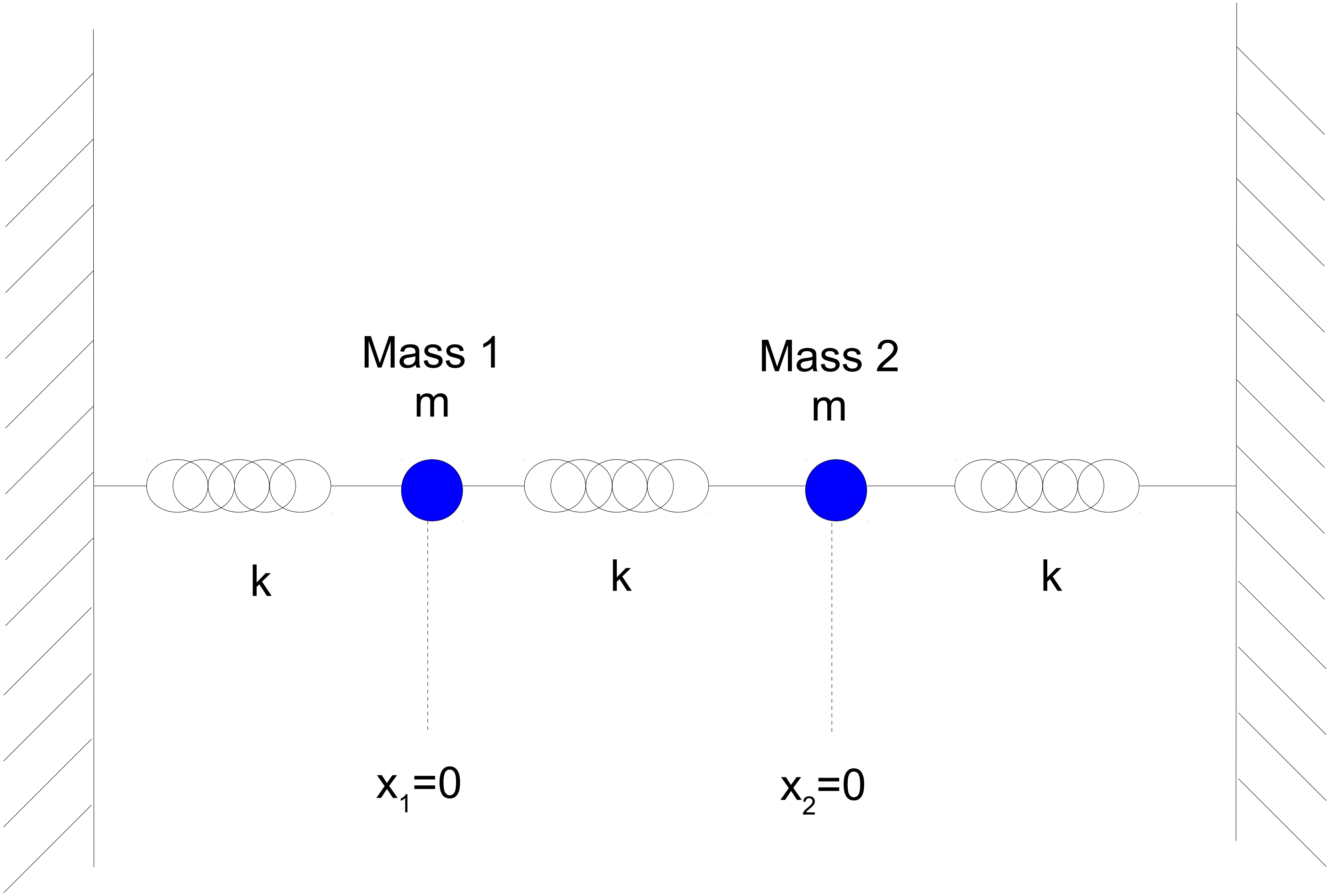
5V → bit 1, 0V → bit 0

Approximation of Continuous Charge Distributions

$$V = \frac{1}{4\pi\epsilon_0}\sum_i \frac{Q_i}{r_i}$$



Mass/Spring Coupled Oscillator



Equations of Motion for Two Mass Oscillator

Let $\omega^2 = \frac{k}{m}$ so $\frac{d^2 x_1}{dt^2} = -2\omega^2 x_1 + \omega^2 x_2$ $\frac{d^2 x_2}{dt^2} = \omega^2 x_1 - 2\omega^2 x_2$

Assume solutions in the form: $x_1 = a_1 e^{i\alpha t}$ $x_2 = a_2 e^{i\alpha t}$

$$-\alpha^2 a_1 e^{i\alpha t} = -2\omega^2 a_1 e^{i\alpha t} + \omega^2 a_2 e^{i\alpha t} \qquad -\alpha^2 a_2 e^{i\alpha t} = \omega^2 a_1 e^{i\alpha t} - 2\omega^2 a_2 e^{i\alpha t}$$

Then cancel the time dependent factors:

$$-\alpha^2 a_1 = -2\omega^2 a_1 + \omega^2 a_2 \qquad -\alpha^2 a_2 = \omega^2 a_1 - 2\omega^2 a_2$$

$$\begin{bmatrix} 2\omega^2 & -\omega^2 \\ -\omega^2 & 2\omega^2 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \alpha^2 \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

$$\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \lambda \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \text{ where } \lambda = \frac{\alpha^2}{\omega^2}$$

Solutions to Eigenvalue Problem

Solution to this eigenvalue and eigenvector problem is

$$\lambda_1 = 3: \begin{bmatrix} 1 \\ -1 \end{bmatrix} \qquad \lambda_2 = 1: \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Any linear combination of eigenvectors are valid solutions, where the linear combination coefficients are to be determined by the initial conditions

$$\lambda = \frac{\alpha^2}{\omega^2} \quad \text{so} \quad \alpha = \pm\sqrt{\lambda}\,\omega$$

The general solution is

$$x_1(t) = b_1 e^{i\sqrt{\lambda_1}\omega t} + b_2 e^{-i\sqrt{\lambda_1}\omega t} + b_3 e^{i\sqrt{\lambda_2}\omega t} + b_4 e^{-i\sqrt{\lambda_2}\omega t}$$
$$x_2(t) = -b_1 e^{i\sqrt{\lambda_1}\omega t} - b_2 e^{-i\sqrt{\lambda_1}\omega t} + b_3 e^{i\sqrt{\lambda_2}\omega t} + b_4 e^{-i\sqrt{\lambda_2}\omega t}$$

Or in a more convenient form

$$x_1(t) = c_1 \cos(\sqrt{3}\,\omega t) + c_2 \sin(\sqrt{3}\,\omega t) + c_3 \cos(\omega t) + c_4 \sin(\omega t)$$
$$x_2(t) = -c_1 \cos(\sqrt{3}\,\omega t) - c_2 \sin(\sqrt{3}\,\omega t) + c_3 \cos(\omega t) + c_4 \sin(\omega t)$$

Applying Initial Conditions for Complete Solution

Example:

$$\text{Initial } x_1(t)=x_{10} \quad \text{Initial } \frac{dx_1(t)}{dt}=0 \quad \text{Initial } x_2(t)=x_{20} \quad \text{Initial } \frac{dx_2(t)}{dt}=0$$

Then only the cosine terms contribute to solution:

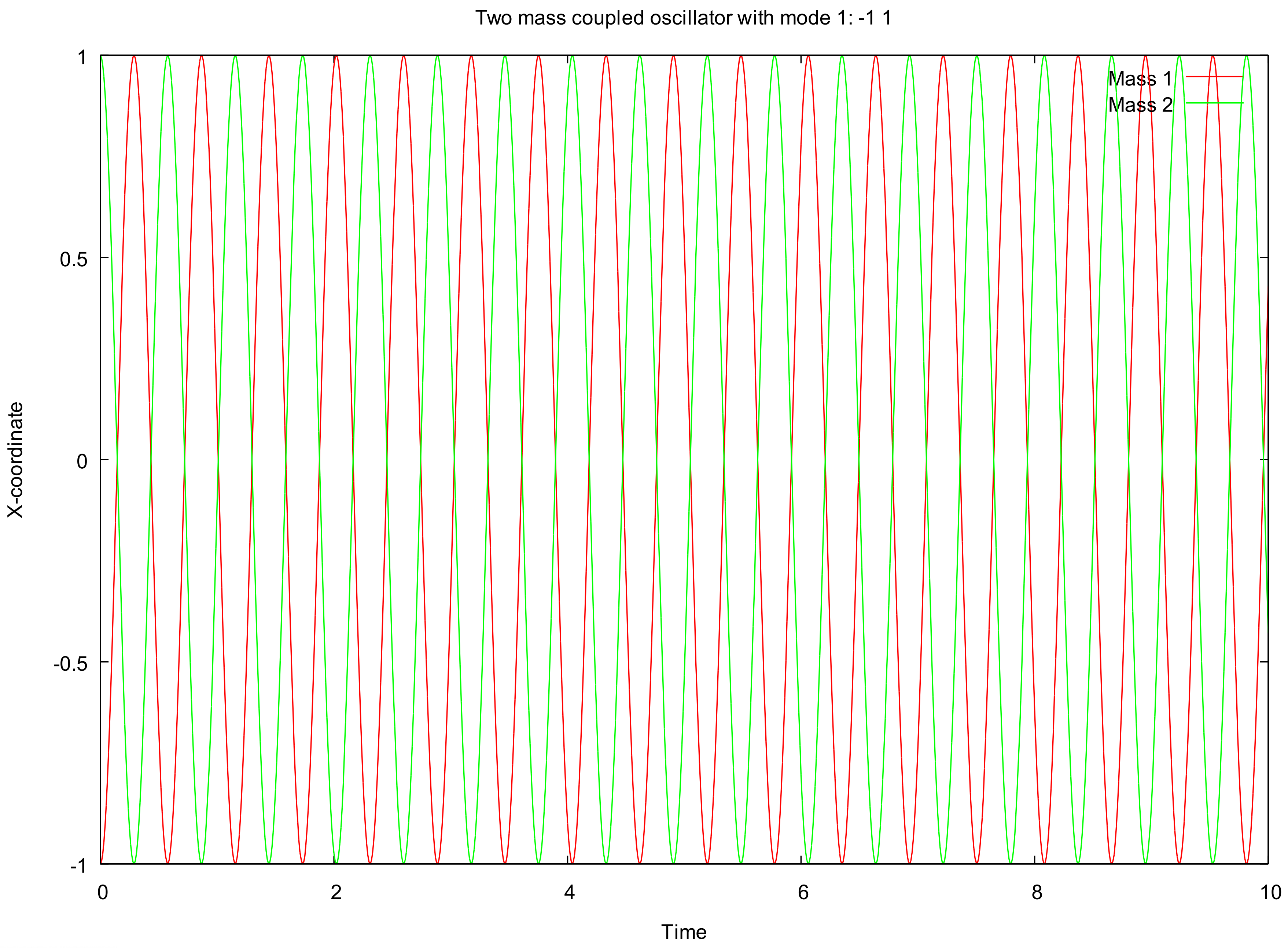
$$x_1(t)=c_1\cos(\sqrt{3}\,\omega t)+c_3\cos(\omega t)$$
$$x_2(t)=-c_1\cos(\sqrt{3}\,\omega t)+c_3\cos(\omega t)$$

Find the coefficients c_1 and c_3 by solving the system of linear equations:

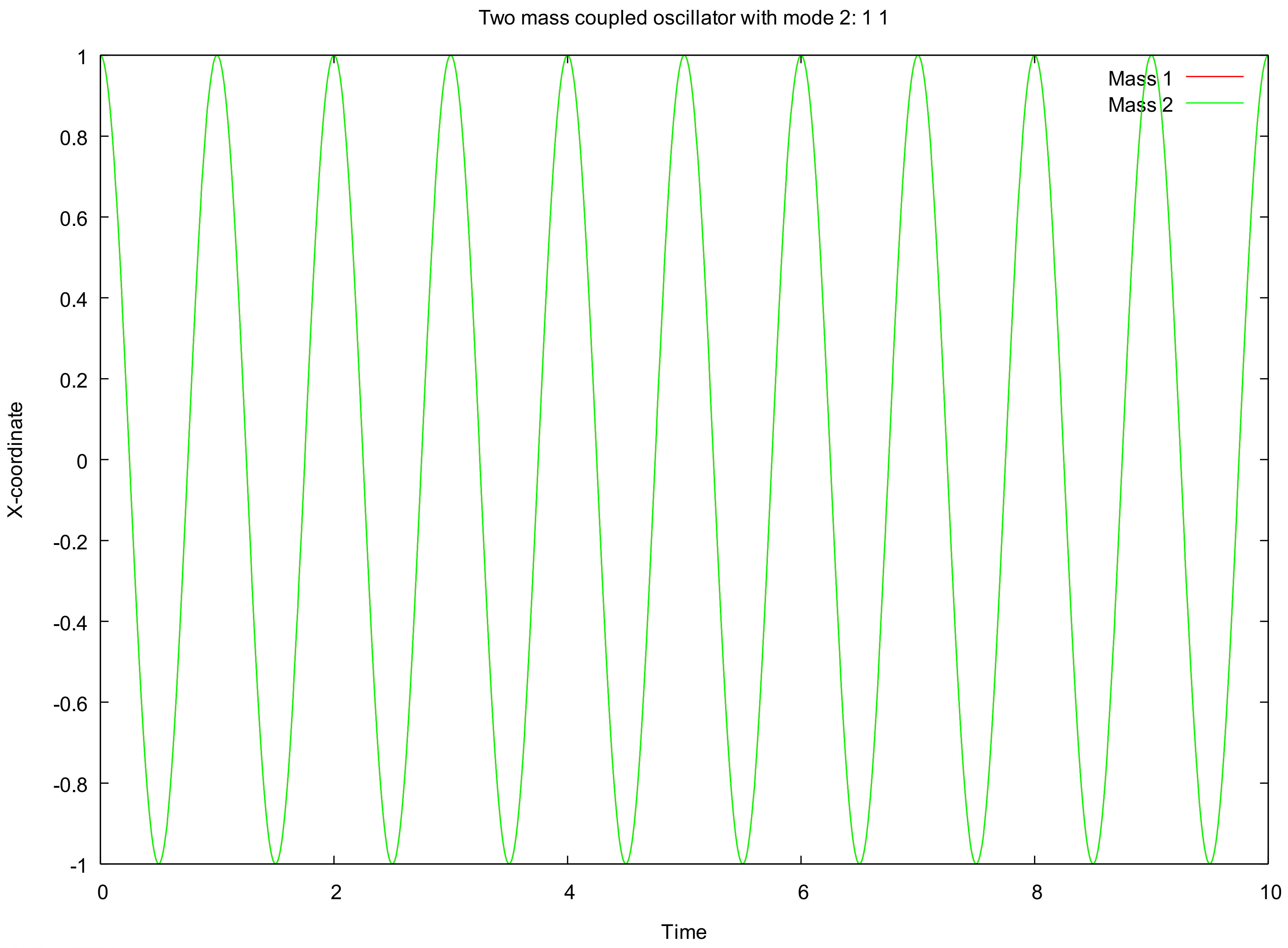
$$\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} c_1 \\ c_3 \end{bmatrix}=\begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix}$$

Note eigenvectors as column vectors in coefficient matrix

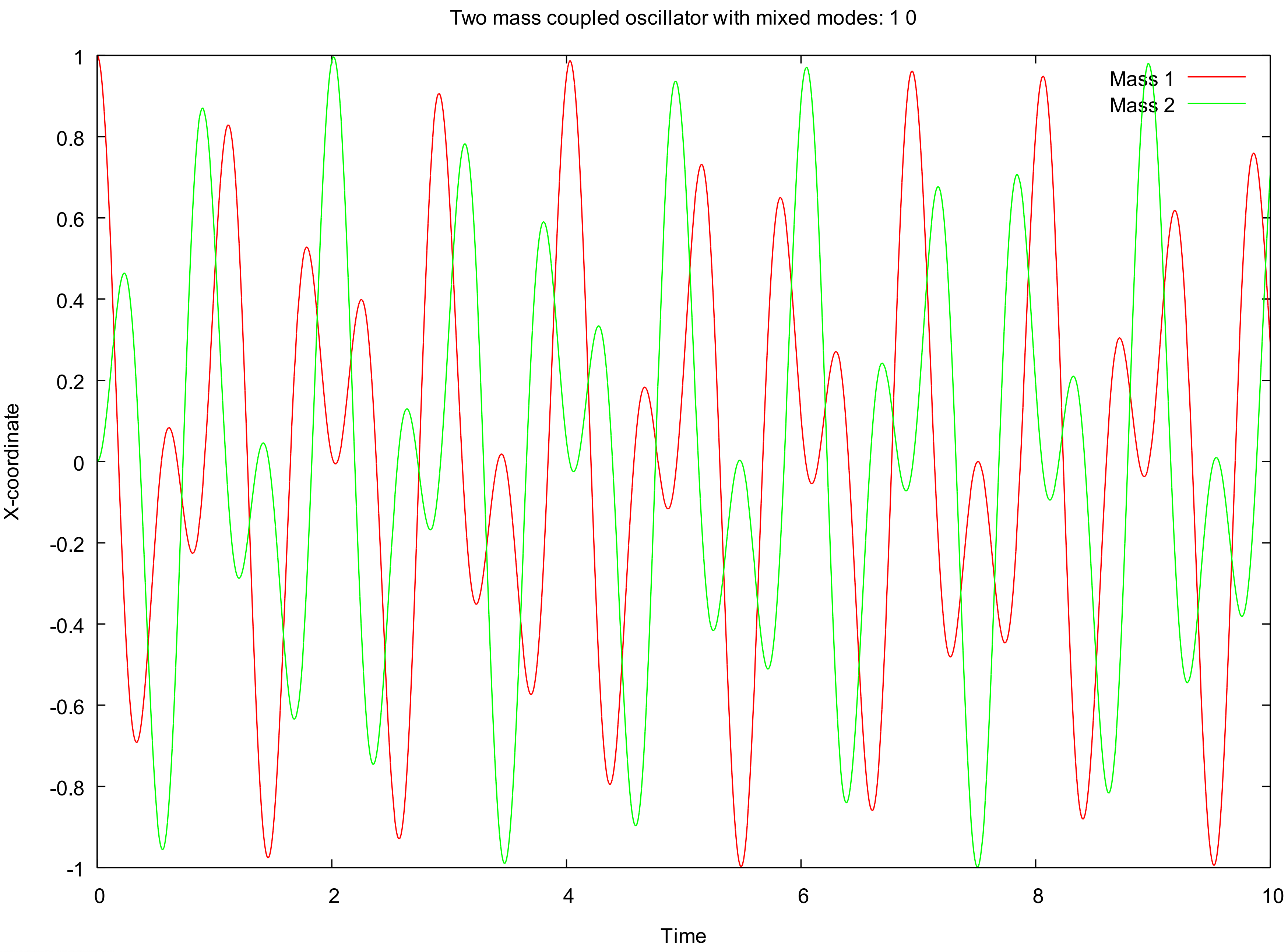
Mode 1 in Two Mass Coupled Oscillator



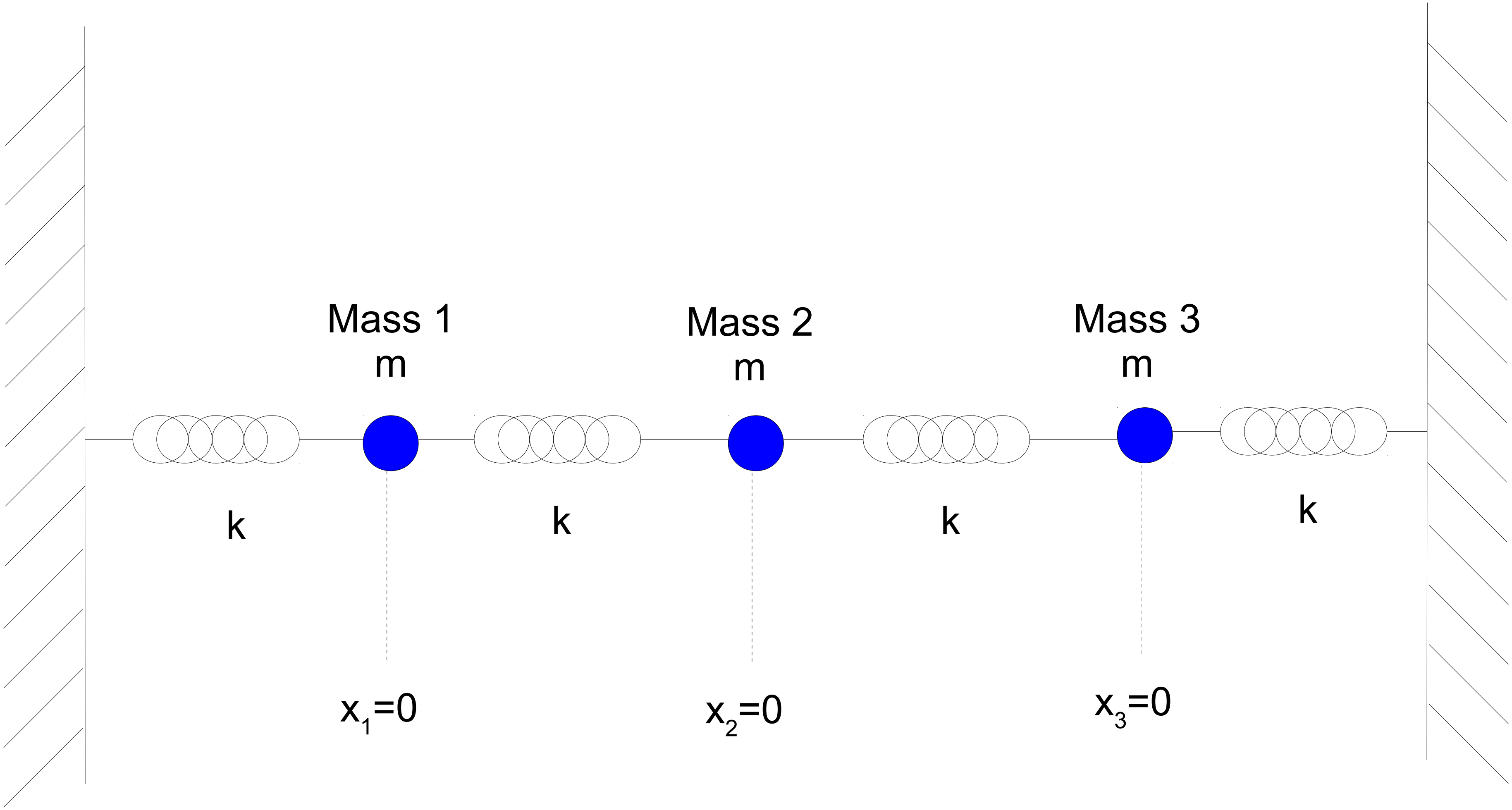
Mode 2 in Two Mass Coupled Oscillator



Mixed Modes in Two Mass Coupled Oscillator



Mass/Spring Coupled Oscillator



Equations of Motion for Three Mass Oscillator

Let $\omega^2 = \frac{k}{m}$

$$\frac{d^2 x_1}{dt^2} = -2\omega^2 x_1 + \omega^2 x_2 \qquad \frac{d^2 x_2}{dt^2} = \omega^2 x_1 - 2\omega^2 x_2 + \omega^2 x_3 \qquad \frac{d^2 x_3}{dt^2} = \omega^2 x_2 - 2\omega^2 x_3$$

Assume solutions in the form $x_1 = a_1 e^{i\alpha t} \qquad x_2 = a_2 e^{i\alpha t} \qquad x_3 = a_3 e^{i\alpha t}$

$$-\alpha^2 a_1 = -2\omega^2 a_1 + \omega^2 a_2 \qquad -\alpha^2 a_2 = \omega^2 a_1 - 2\omega^2 a_2 + \omega^2 a_3 \qquad -\alpha^2 a_3 = \omega^2 a_2 - 2\omega^2 a_3$$

$$\begin{bmatrix} 2\omega^2 & -\omega^2 & 0 \\ -\omega^2 & 2\omega^2 & -\omega^2 \\ 0 & -\omega^2 & 2\omega^2 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \alpha^2 \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \lambda \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad \text{where} \quad \lambda = \frac{\alpha^2}{\omega^2}$$

SMU

Solutions to Eigenvalue Problem

Solution to this eigenvalue and eigenvector problem is

$$\lambda_1=2:\ \begin{bmatrix}-1\\0\\1\end{bmatrix}\qquad \lambda_2=2-\sqrt{2}:\ \begin{bmatrix}1\\\sqrt{2}\\1\end{bmatrix}\qquad \lambda_3=2+\sqrt{2}:\ \begin{bmatrix}1\\-\sqrt{2}\\1\end{bmatrix}$$

Any linear combination of eigenvectors are valid solutions, where the linear combination coefficients are to be determined by the initial conditions

$$\lambda=\frac{\alpha^2}{\omega^2}\quad \text{so}\quad \alpha=\pm\sqrt{\lambda}\,\omega$$

The general solution is

$$\begin{aligned}
x_1(t)&=-b_1e^{i\sqrt{\lambda_1}\omega t}-b_2e^{-i\sqrt{\lambda_1}\omega t}+b_3e^{i\sqrt{\lambda_2}\omega t}+b_4e^{-i\sqrt{\lambda_2}\omega t}+b_5e^{i\sqrt{\lambda_3}\omega t}+b_6e^{-i\sqrt{\lambda_3}\omega t}\\
x_2(t)&=\sqrt{2}b_3e^{i\sqrt{\lambda_2}\omega t}+\sqrt{2}b_4e^{-i\sqrt{\lambda_2}\omega t}-\sqrt{2}b_5e^{i\sqrt{\lambda_3}\omega t}-\sqrt{2}b_6e^{-i\sqrt{\lambda_3}\omega t}\\
x_3(t)&=b_1e^{i\sqrt{\lambda_1}\omega t}+b_2e^{-i\sqrt{\lambda_1}\omega t}+b_3e^{i\sqrt{\lambda_2}\omega t}+b_4e^{-i\sqrt{\lambda_2}\omega t}+b_5e^{i\sqrt{\lambda_3}\omega t}+b_6e^{-i\sqrt{\lambda_3}\omega t}
\end{aligned}$$

Or in a more convenient form

$$\begin{aligned}
x_1(t)&=-c_1\cos(\sqrt{\lambda_1}\omega t)-c_2\sin(\sqrt{\lambda_1}\omega t)+c_3\cos(\sqrt{\lambda_2}\omega t)+c_4\sin(\sqrt{\lambda_2}\omega t)+c_5\cos(\sqrt{\lambda_3}\omega t)+c_6\sin(\sqrt{\lambda_3}\omega t)\\
x_2(t)&=\sqrt{2}c_3\cos(\sqrt{\lambda_2}\omega t)+\sqrt{2}c_4\sin(\sqrt{\lambda_2}\omega t)-\sqrt{2}c_5\cos(\sqrt{\lambda_3}\omega t)-\sqrt{2}c_6\sin(\sqrt{\lambda_3}\omega t)\\
x_3(t)&=c_1\cos(\sqrt{\lambda_1}\omega t)+c_2\sin(\sqrt{\lambda_1}\omega t)+c_3\cos(\sqrt{\lambda_2}\omega t)+c_4\sin(\sqrt{\lambda_2}\omega t)+c_5\cos(\sqrt{\lambda_3}\omega t)+c_6\sin(\sqrt{\lambda_3}\omega t)
\end{aligned}$$

Applying Initial Conditions for Complete Solution

Example:

$$\text{Initial } x_1(t) = x_{10} \quad \text{Initial } \frac{dx_1(t)}{dt} = 0 \quad \text{Initial } x_2(t) = x_{20} \quad \text{Initial } \frac{dx_2(t)}{dt} = 0$$

$$\text{Initial } x_3(t) = x_{30} \quad \text{Initial } \frac{dx_3(t)}{dt} = 0$$

Then only the cosine terms contribute to solution:

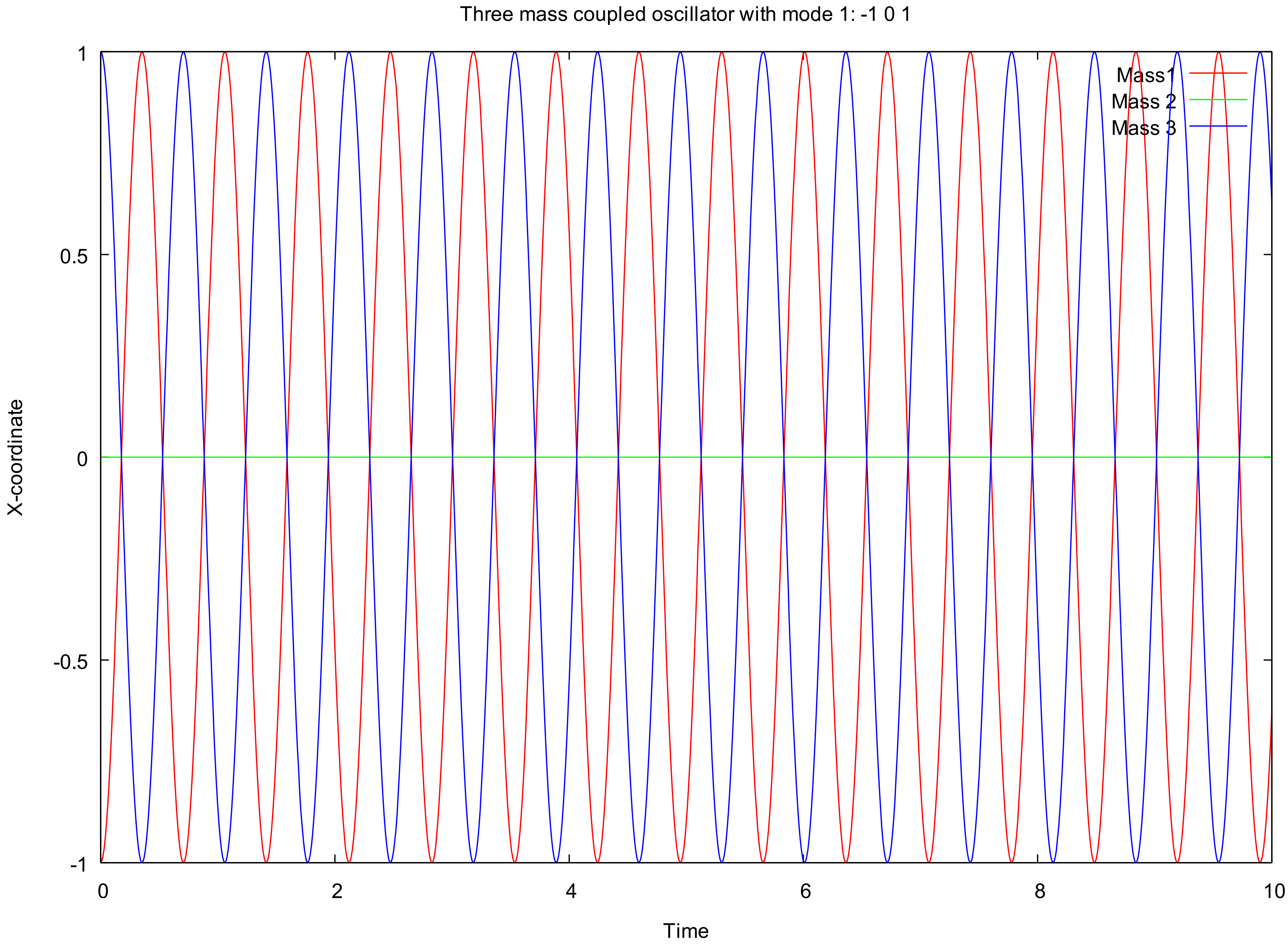
$$\begin{aligned}
x_1(t) &= -c_1 \cos\left(\sqrt{2}\,\omega t\right) + c_3 \cos\left(\sqrt{2-\sqrt{2}}\,\omega t\right) + c_5 \cos\left(\sqrt{2+\sqrt{2}}\,\omega t\right) \\
x_2(t) &= \sqrt{2}\,c_3 \cos\left(\sqrt{2-\sqrt{2}}\,\omega t\right) - \sqrt{2}\,c_5 \cos\left(\sqrt{2+\sqrt{2}}\,\omega t\right) \\
x_3(t) &= c_1 \cos\left(\sqrt{2}\,\omega t\right) + c_3 \cos\left(\sqrt{2-\sqrt{2}}\,\omega t\right) + c_5 \cos\left(\sqrt{2+\sqrt{2}}\,\omega t\right)
\end{aligned}$$

Find the coefficients c_1 c_3 and c_5 by solving the system of linear equations:

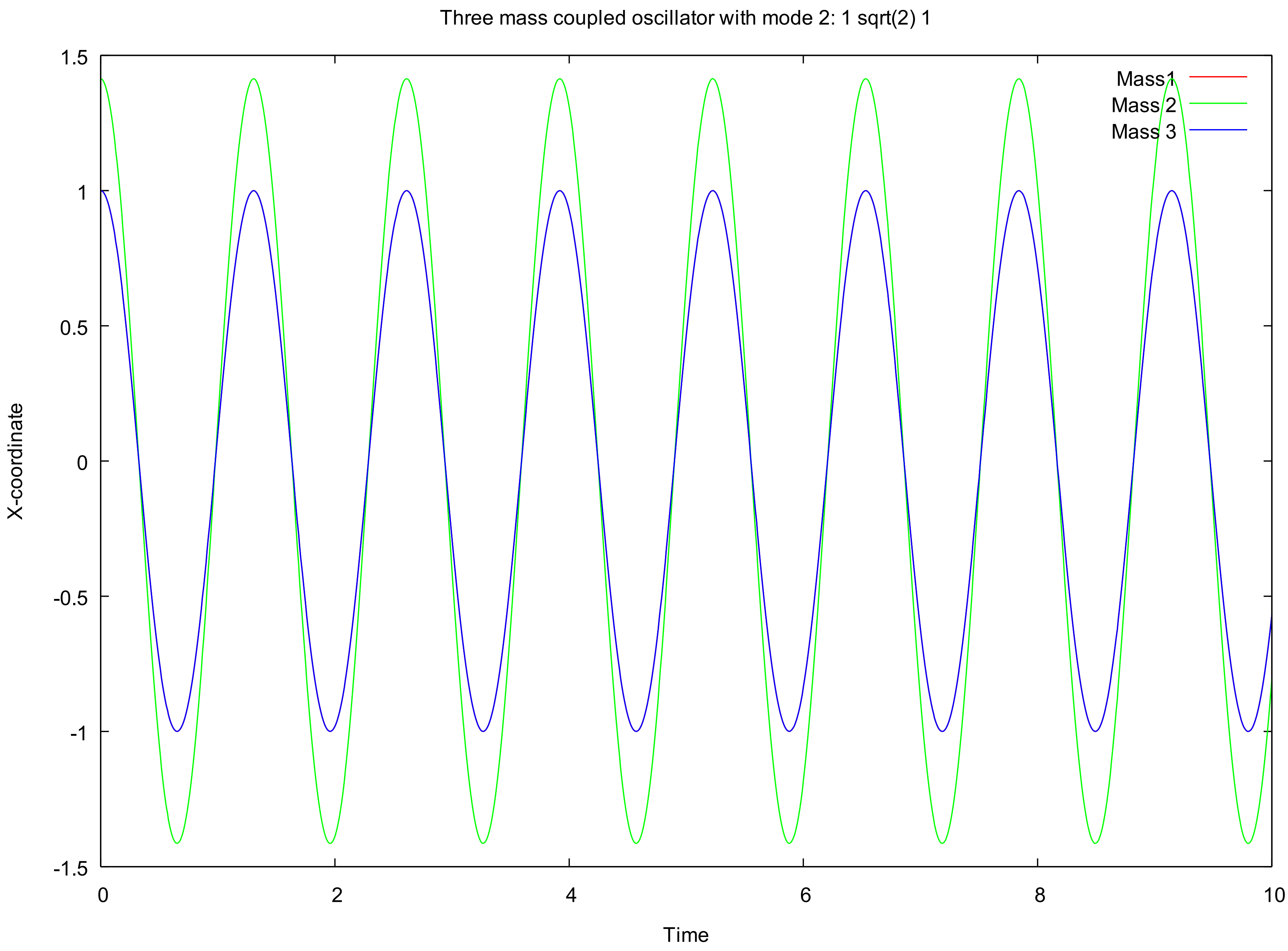
$$\begin{bmatrix} -1 & 1 & 1 \\ 0 & \sqrt{2} & -\sqrt{2} \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_3 \\ c_5 \end{bmatrix} = \begin{bmatrix} x_{10} \\ x_{20} \\ x_{30} \end{bmatrix}$$

Note eigenvectors as column vectors in coefficient matrix

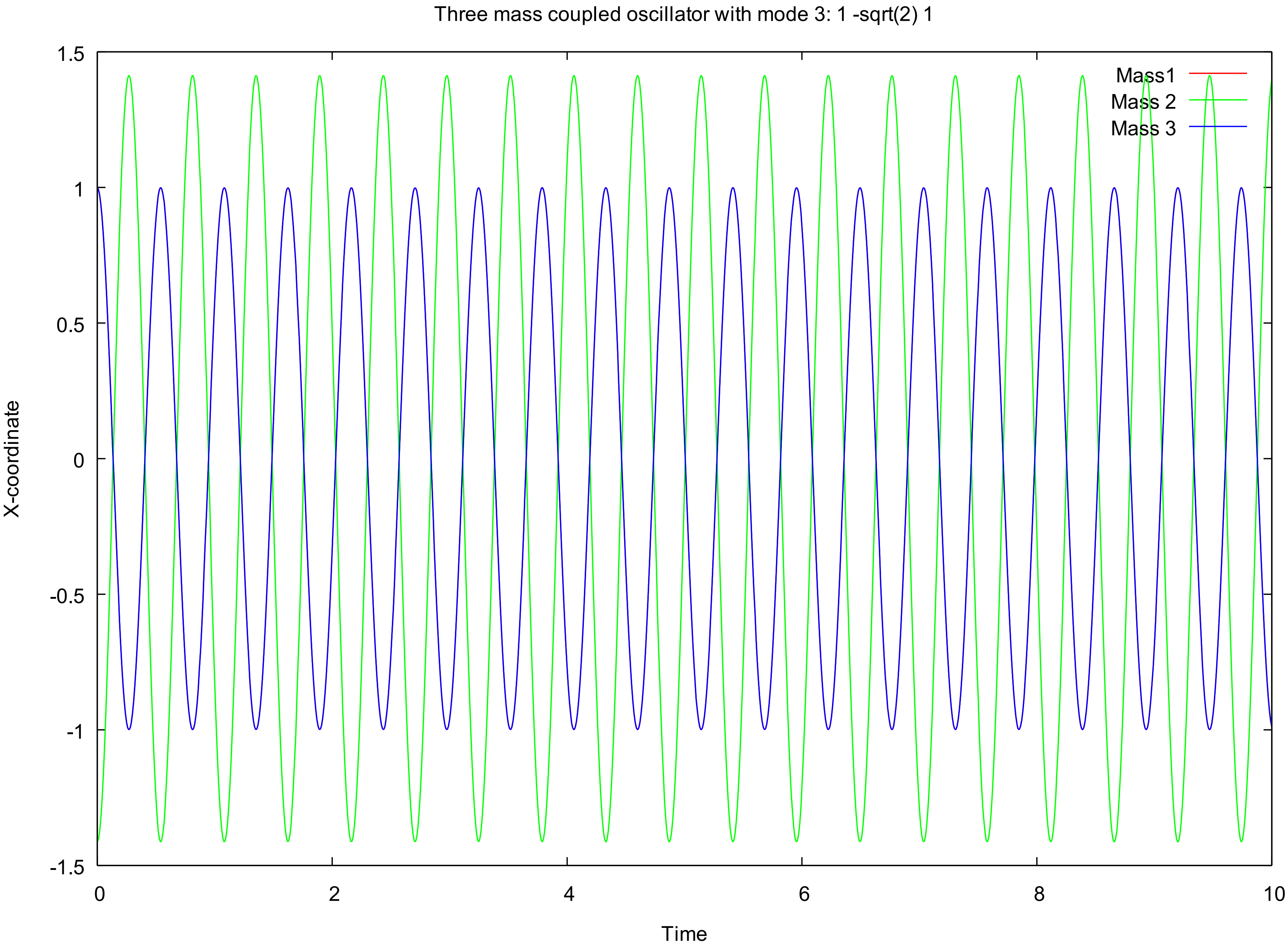
Mode 1 in Three Mass Coupled Oscillator



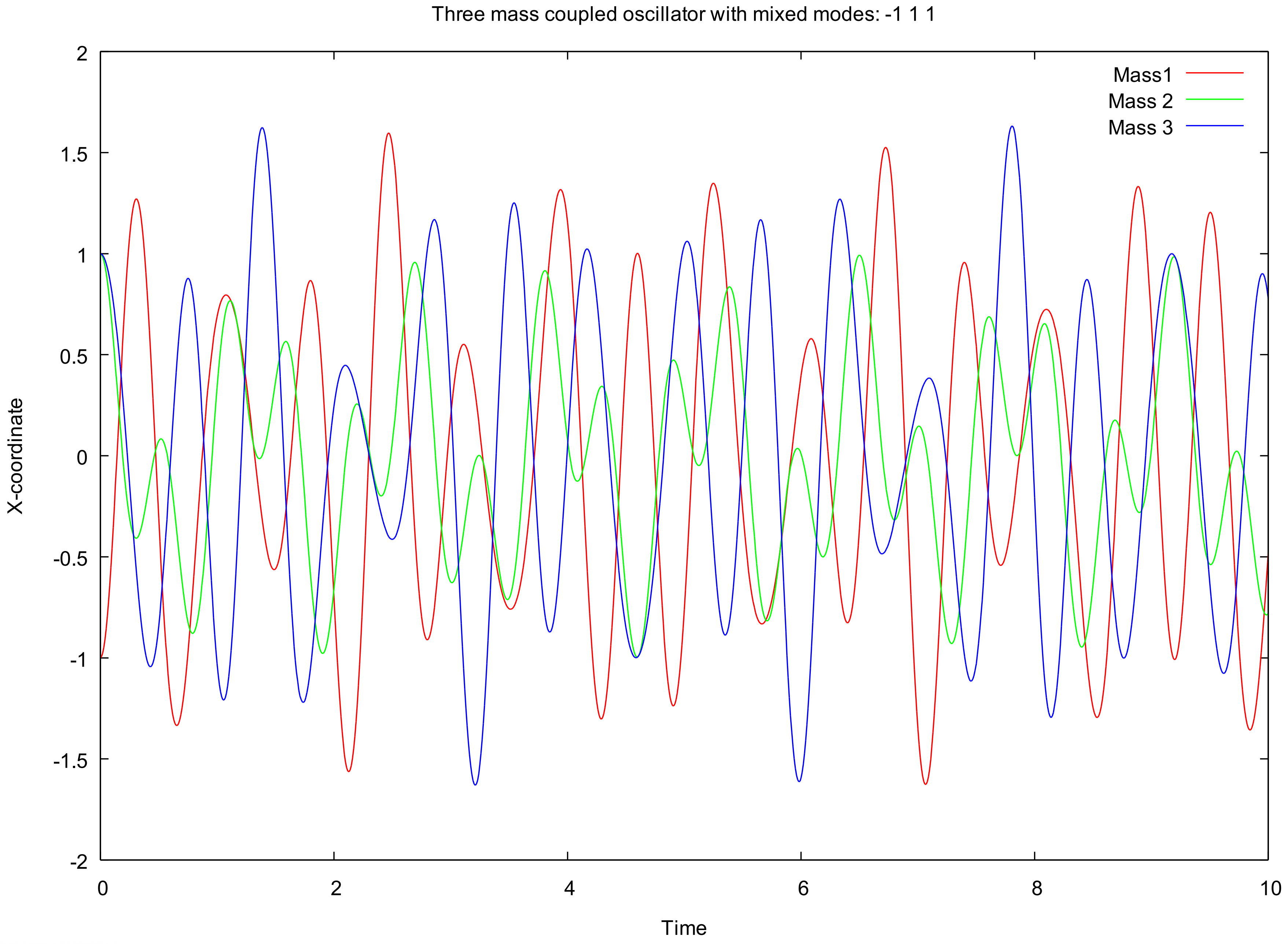
Mode 2 in Three Mass Coupled Oscillator



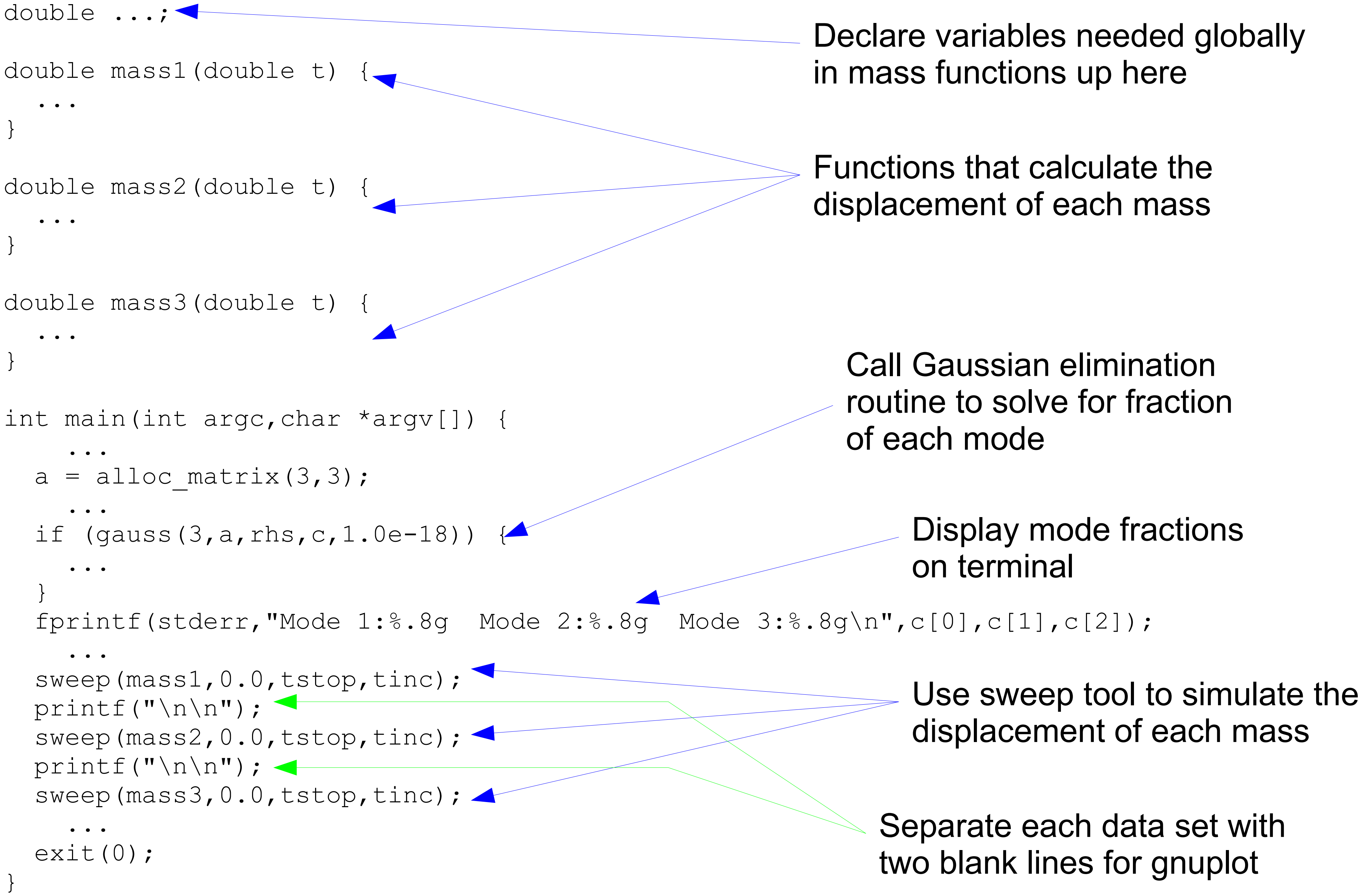
Mode 3 in Three Mass Coupled Oscillator



Mixed Modes in Three Mass Coupled Oscillator



Outline of C Code for Coupled Oscillator



```
double ...;

double mass1(double t) {
  ...
}

double mass2(double t) {
  ...
}

double mass3(double t) {
  ...
}

int main(int argc,char *argv[]) {
    ...
  a = alloc_matrix(3,3);
    ...
  if (gauss(3,a,rhs,c,1.0e-18)) {
    ...
  }
  fprintf(stderr,"Mode 1:%.8g  Mode 2:%.8g  Mode 3:%.8g\n",c[0],c[1],c[2]);
    ...
  sweep(mass1,0.0,tstop,tinc);
  printf("\n\n");
  sweep(mass2,0.0,tstop,tinc);
  printf("\n\n");
  sweep(mass3,0.0,tstop,tinc);
    ...
  exit(0);
}
```

Declare variables needed globally in mass functions up here

Functions that calculate the displacement of each mass

Call Gaussian elimination routine to solve for fraction of each mode

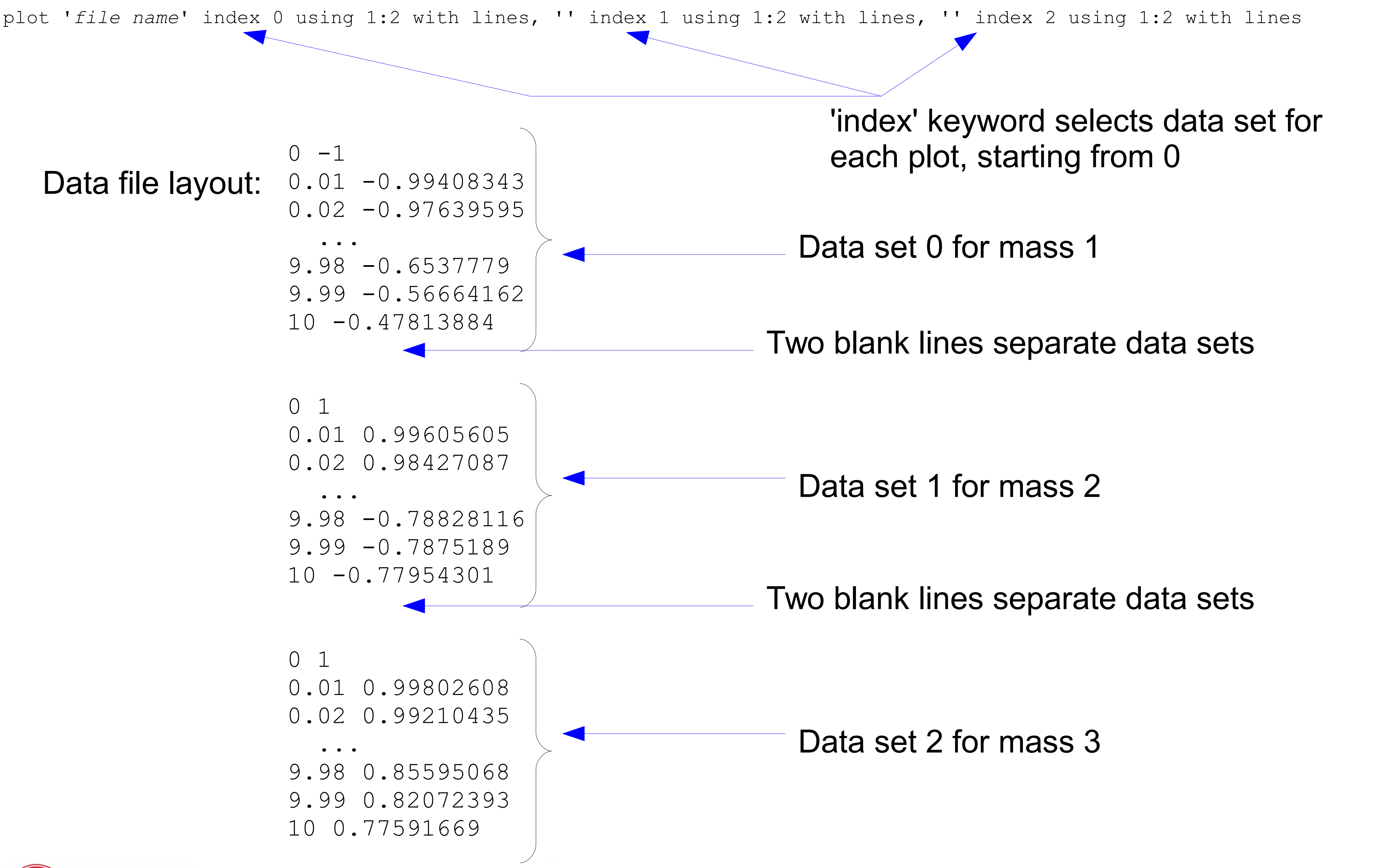
Display mode fractions on terminal

Use sweep tool to simulate the displacement of each mass

Separate each data set with two blank lines for gnuplot

Plotting Multiple “Data Sets” in gnuplot

```
plot 'file name' index 0 using 1:2 with lines, '' index 1 using 1:2 with lines, '' index 2 using 1:2 with lines
```



'index' keyword selects data set for each plot, starting from 0

Data file layout:

```
0 -1
0.01 -0.99408343
0.02 -0.97639595
  ...
9.98 -0.6537779
9.99 -0.56664162
10 -0.47813884


0 1
0.01 0.99605605
0.02 0.98427087
  ...
9.98 -0.78828116
9.99 -0.7875189
10 -0.77954301


0 1
0.01 0.99802608
0.02 0.99210435
  ...
9.98 0.85595068
9.99 0.82072393
10 0.77591669
```

Data set 0 for mass 1

Two blank lines separate data sets

Data set 1 for mass 2

Two blank lines separate data sets

Data set 2 for mass 3