Contents

- Computational sciences
- Software development
- Basic data structures
- Number representation
- Partial differential equations
- Grids
- Finite differences
- Finite elements
- Grid interpolation
- Elliptic equations
- Systems of equations
- Hyperbolic equations
- Finite volumes
- Ordinary differential equations
- Particle methods
- High performance computing

COMPUTATIONAL MODELING AND SIMULATION

Juan R. Cebral

George Mason University
Fairfax, Virginia, USA

COMPUTATIONAL SCIENCES

Sciences

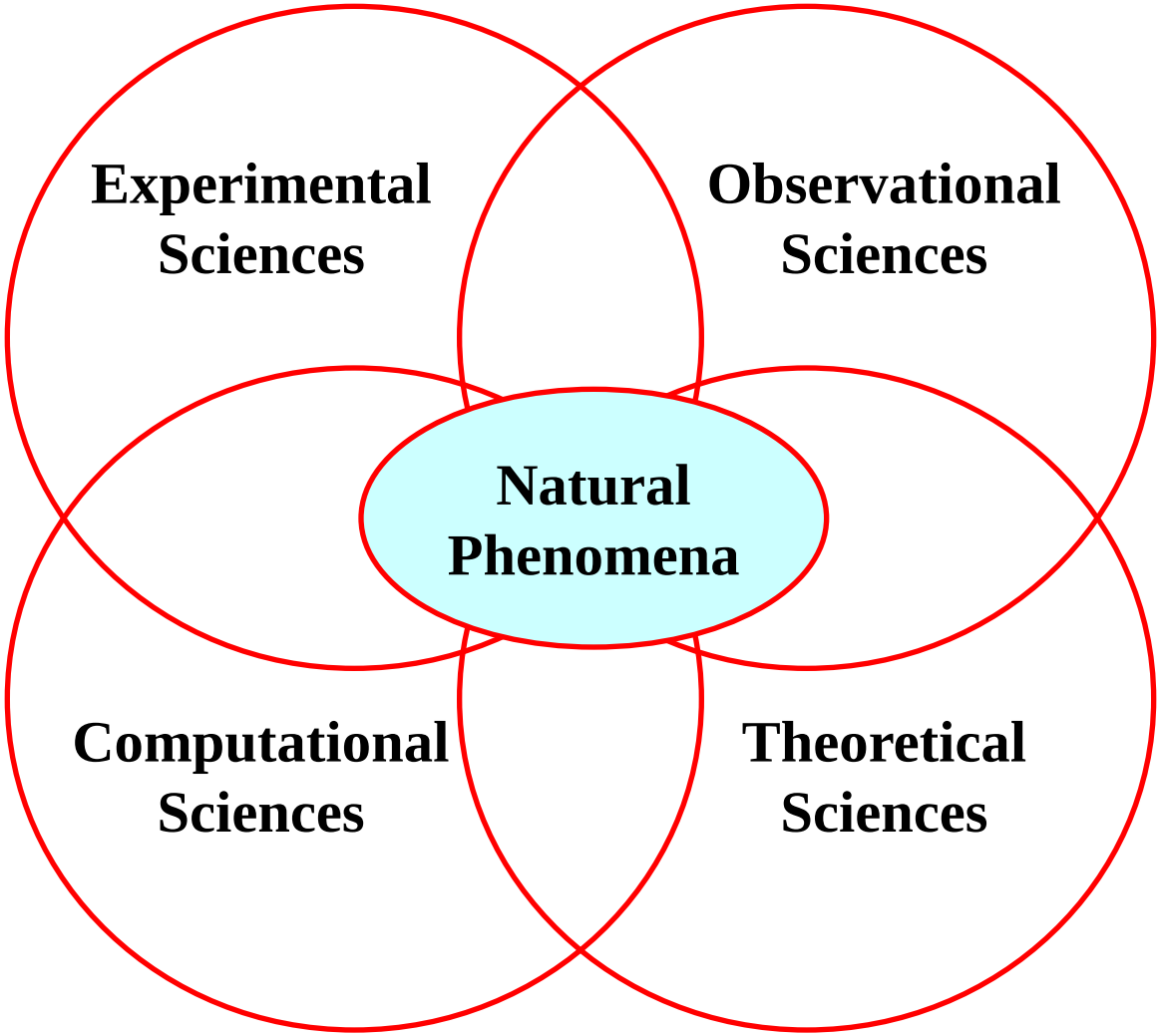
Natural sciences:

- Physics
- Biology
- Astronomy
- Chemistry
- Medicine

Abstract sciences:

- Mathematical sciences
- Data sciences
- Computer science

Natural Sciences



Numbers in Science

Greeks:

- Numbers and formulas to describe the world (geometry, algebra)

Newton & Co.:

- Functions not numbers! (differential equations)

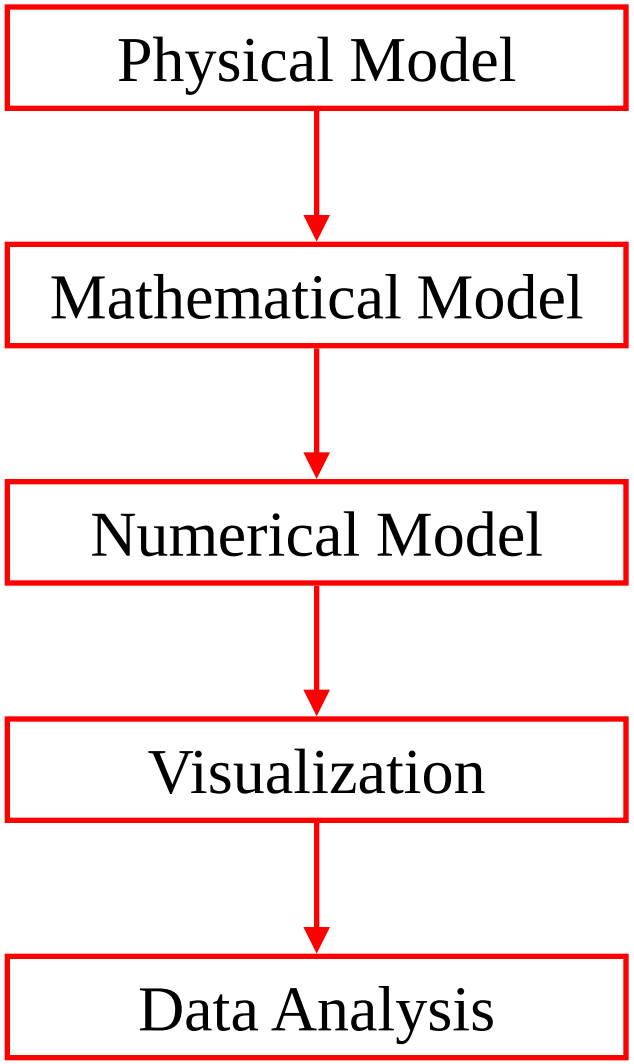
Modern sciences:

- Return of the numbers (numerical solutions)
- Computational sciences (numerical models)

Themes of Computational Sciences

- Modeling and Simulation
- Numerical Methods
- Data Analysis
- Databases / Data mining
- Visualization
- High Performance Computing

Computer simulations



Example

- Aerodynamics problem:
  need to know the forces acting on the wing of an airplane in order to create a better design

- Analytic solutions available only for simple geometries

- Experiments possible but expensive
  => would like to perform only a few

- Perform computational experiments to gain knowledge and to guide the experimental work

Physical Model

Continuous hypothesis
- Air is a continuous medium
- Valid for scales large compared to molecules mean free path

Inviscid fluid
- Air has no friction
- Valid for high Reynolds number: $Re=L.u/\upsilon$

Incompressible flow
- Propagation of information is instantaneous
- Valid for low Mach number: $Ma=v/c$

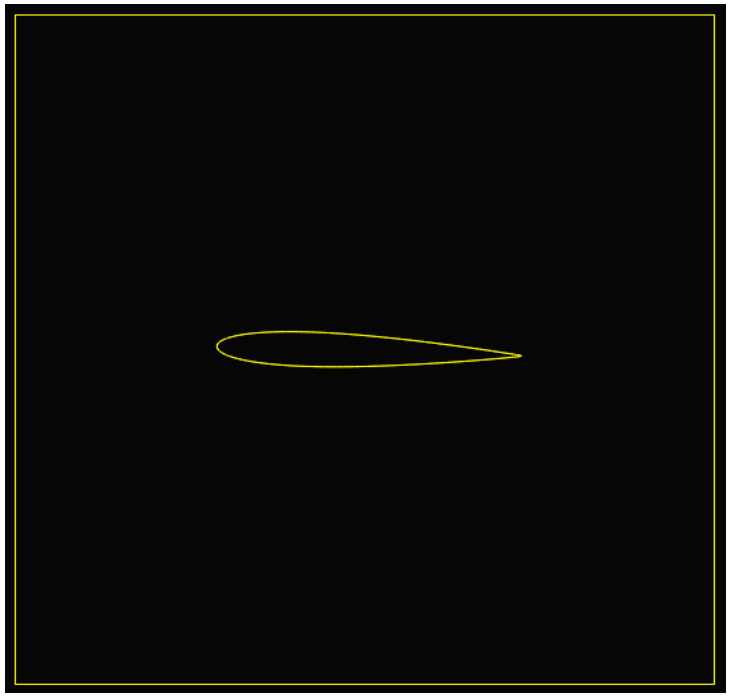
Irrotational flow
- No vorticity (rotation of fluid elements)
- Valid for inviscid fluid far from solid walls

Mathematical Model

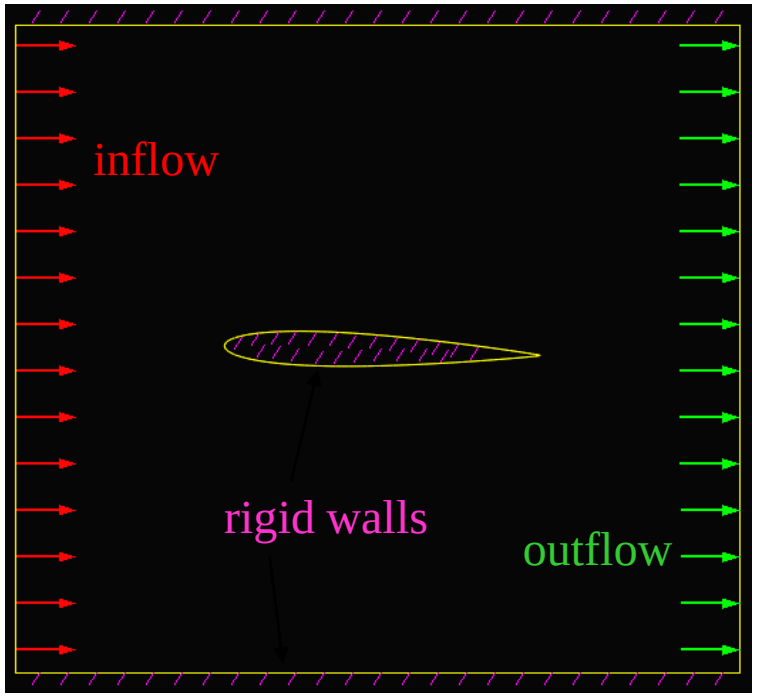
- Incompressible fluid: $\nabla \cdot \mathbf{v} = 0$
- Irrotational flow: $\nabla \times \mathbf{v} = 0 \quad \Rightarrow \quad \mathbf{v} = \nabla \phi$
- Laplace's equation : $\nabla^2 \phi = 0$
- Bernoulli's equation: $\frac{p}{\rho} + \frac{1}{2}\mathbf{v}^2 = const.$
- Two dimensions: $\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$

Problem Specification

Geometrical Model (2D)



Boundary Conditions

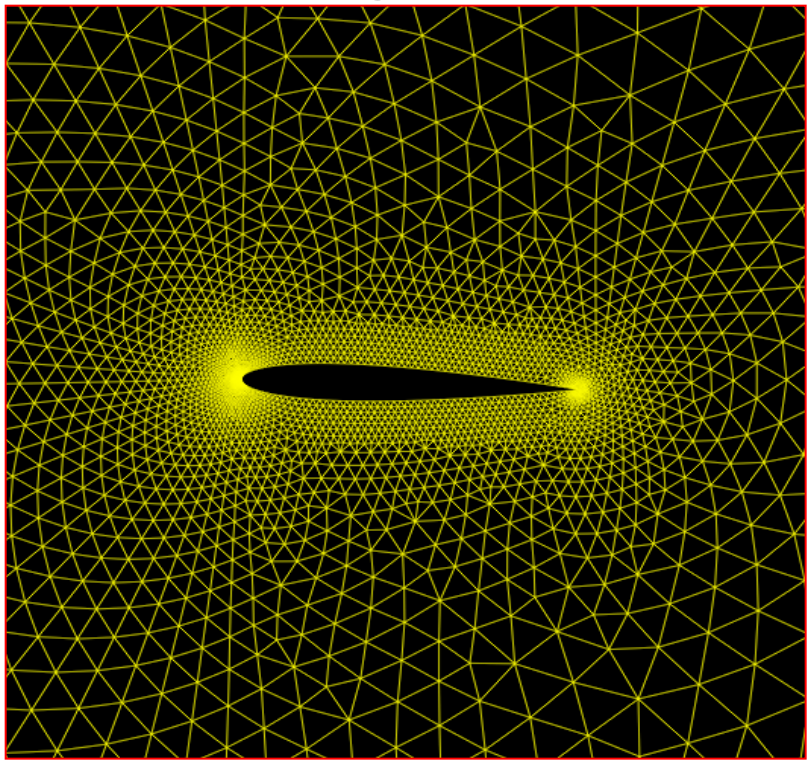


Numerical Model

Spatial discretization

- Unstructured grid
- Values at the nodes
- Grid generation

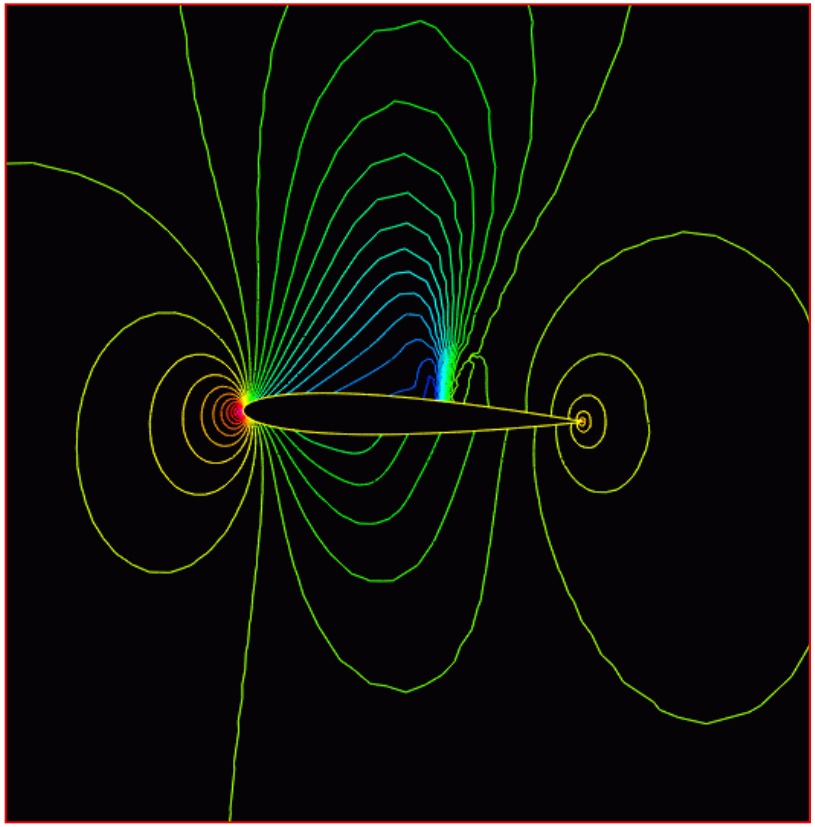
Triangular mesh



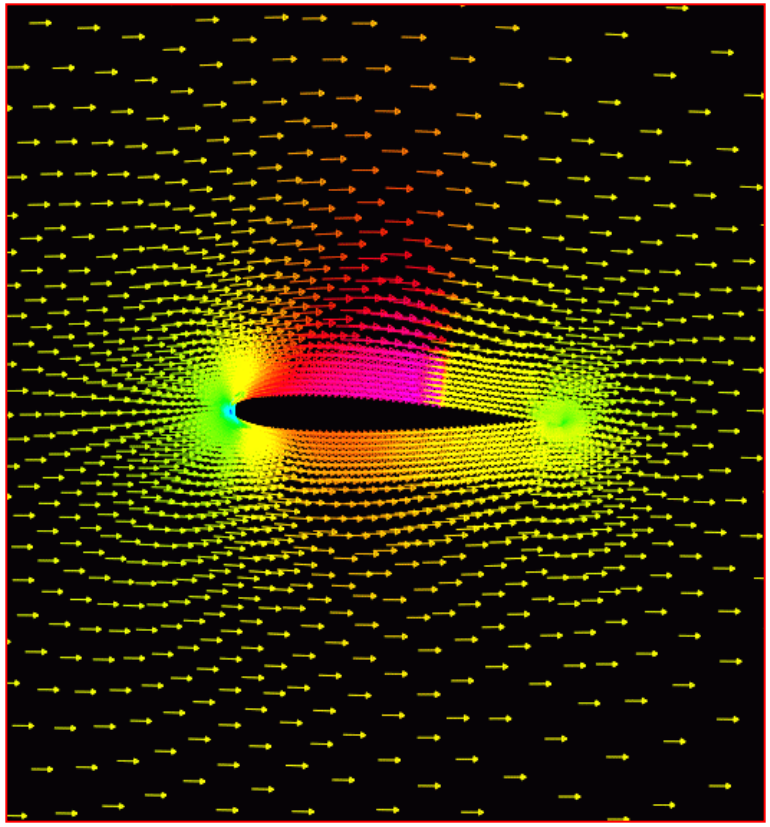
Numerical solution

- Finite elements
- Matrix solver
- Algorithm:
  - Solve Laplace's eq. $\Rightarrow \phi$
  - Calc gradient $\Rightarrow \mathbf{v}$
  - Calc pressure Bernoulli's eq. $\Rightarrow$ p
  - Calc force $\Rightarrow \mathbf{f} = \int_{body} p\,\mathbf{n}\,dS$

Visualization & Data Analysis



Pressure field



Velocity field

=> Analyze forces on wing and try new design

Transforming Observational Sciences

- Computational sciences allow to transform an observational science into a predictive science

- Example: astrophysics
  - Create models of galactic collisions
  - Perform simulations predicting shapes of merging galaxies
  - Compare with observations
  - Predict future evolution

Computational Scientists

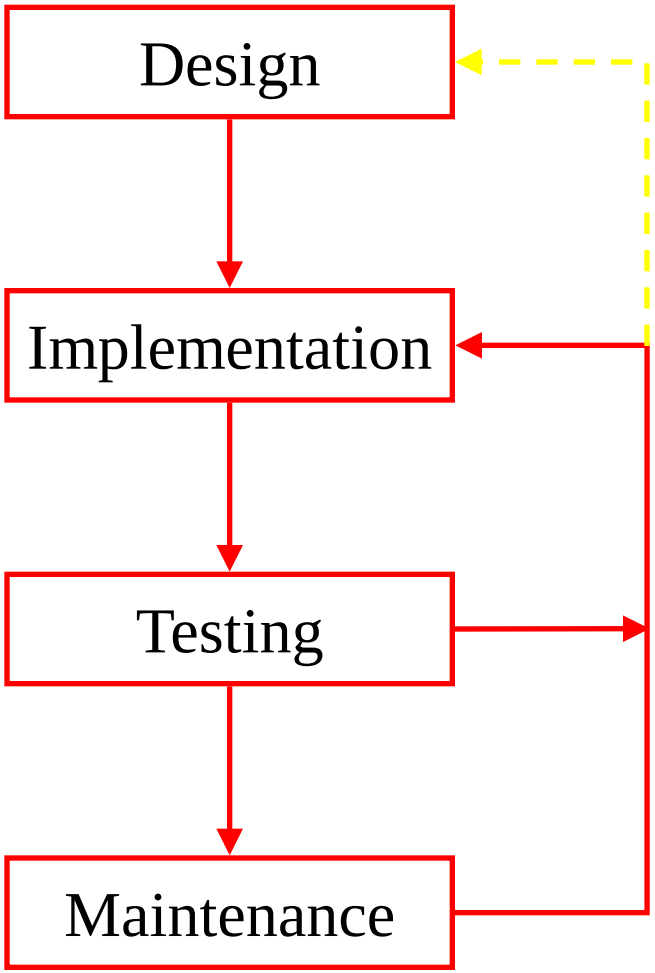
This type of research requires computational scientists:

- Requires knowledge of the science
- Requires knowledge of computational methods
- Highly multi-disciplinary: requires scientists who speak “many languages”

What is important is the science (e.g. medicine)
not just the computational technology per se

SOFTWARE DEVELOPMENT

Software Development



Some Design Criteria

- Speed: language, datastructures, optimization, parallel
- Extensibility: generality, formats
- Software reusability: language, encapsulation, libraries
- Portability: language, compilers, libraries
- Libraries: low level, high level
- Layering
- Graphical user interface: need?, language, OS

Libraries

Numerical

- Lapack / Blas
- PetSc
- IMSL
- itk
- Numerical recipes

Graphics / visualization

- OpenGL
- vtk
- Performer
- Inventor

Parallel computing

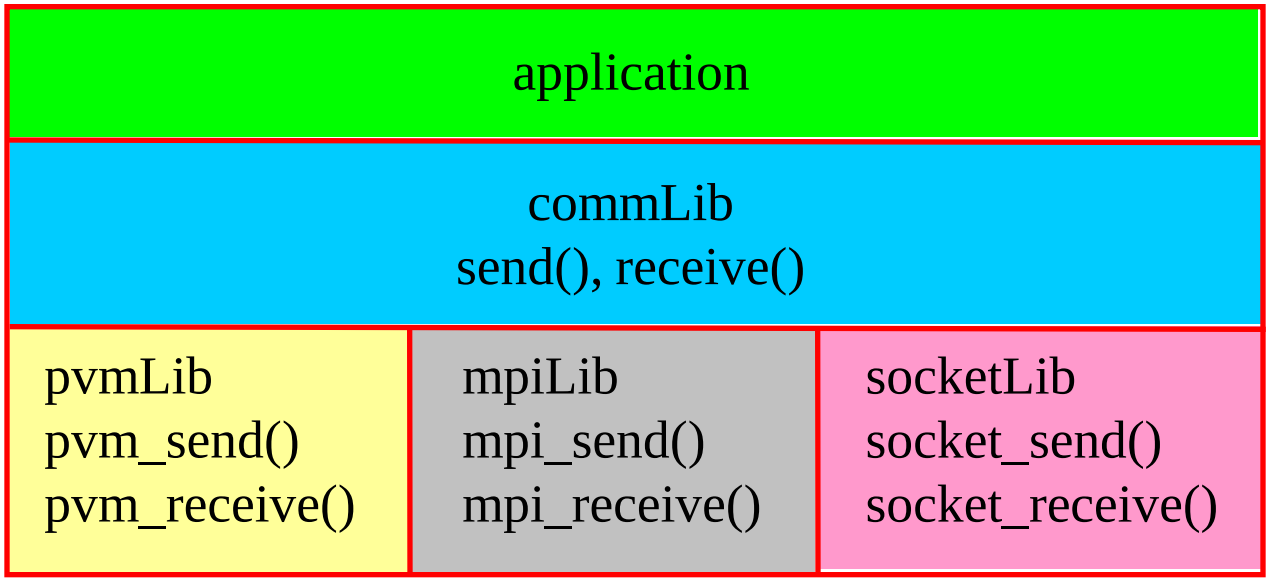
- pvm
- mpi
- pthreads
- OpenMP

GUI

- Motif
- Qt
- Gtk
- Tcl / Tk
- MFC

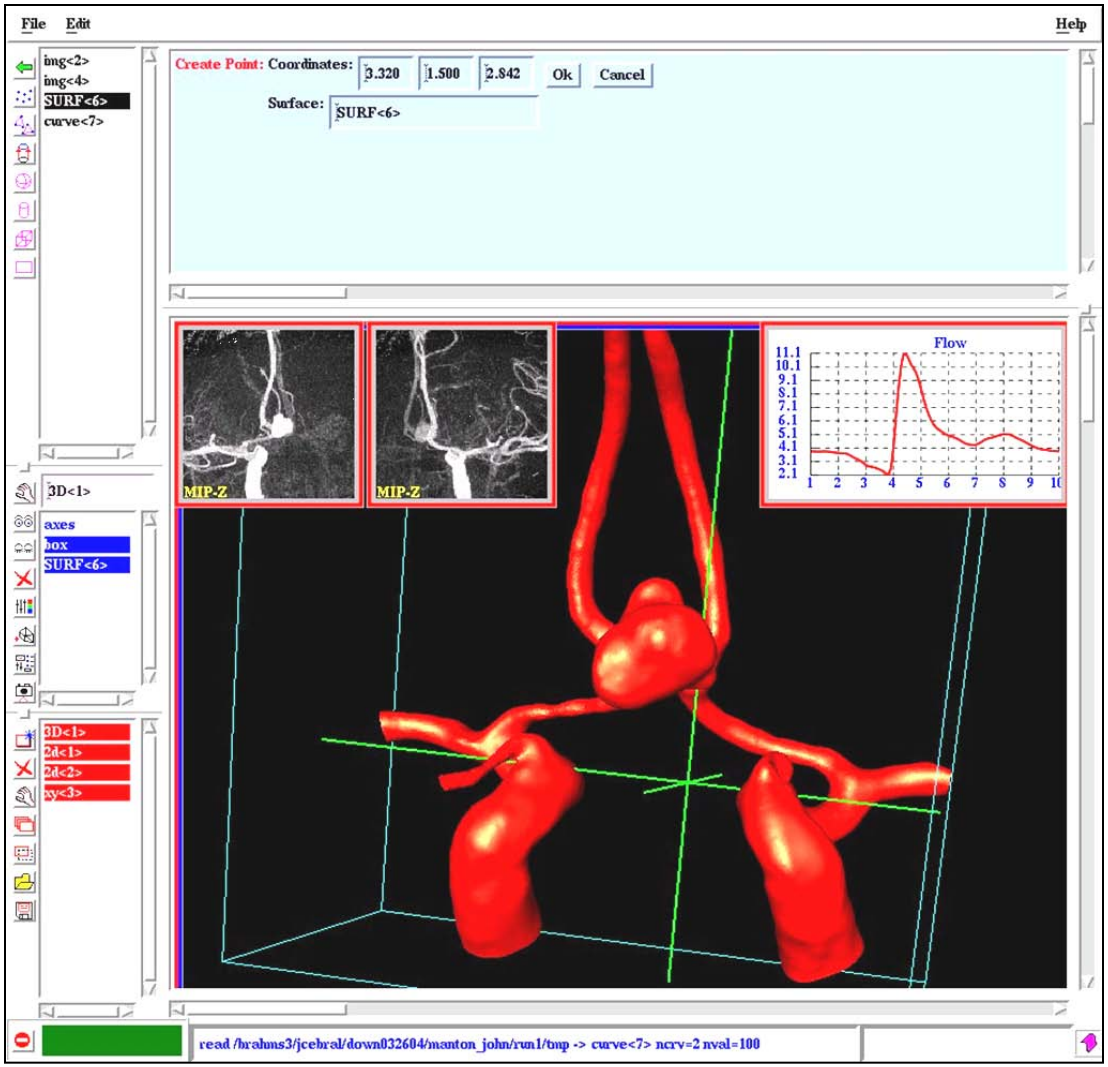
Layering

- Encapsulate library or external code dependencies
- Allows easy replacement of libraries or external codes
- Enhances extensibility
- Enhances portability
- Example:



Designing a GUI

- Simple
- Intuitive
- Few windows
- Customizable
- Fast



Code execution

- Serial / Wizard: Each stage of the code is executed sequentially

```
main() {
  stage1();
  stage2();
  stage3();
}
```

- Event driven: Execution depends on user demands

```
main() {
  loop() {
    wait_for_event();
    switch( event ) {
      1: stage1();
      2: stage2();
      3: stage3();
    }
  }
}
```

Extreme testing concept

- For each routine, write a test code
- Each time the software is updated:
  - All pieces are recompiled
  - All tests are re-run
- This is done automatically on:
  - Different architectures
  - Different compilers
  - Different targets (optimized vs debug mode)

Programming languages & paradigms

Programming Paradigms

- Generic programming
- Object-oriented programming

- Sequential execution
- Event-driven execution

- Serial execution
- Parallel execution

- Recursive codes
- Unrolled codes

Generic languages

- Loops + conditionals + jumps
- Functions

Object-oriented languages

- Abstraction
- Inheritance
- Polymorphism

Programming Languages

Compiled languages

- assembly
- C
- C++
- fortran
- fortran90
- basic
- pascal

Scripting languages

- sh / csh / bash
- perl
- python
- awk
- tcl / tk

Interpreted languages

- IDL
- Matlab
- S / R
- Lisp

Database languages

- Oracle
- SQL

Symbolic computing languages

- Mathematica
- Maple

The choice of programming language depends on the application

Developing software under linux

- Compilers:
  - C: gcc
  - C++: g++
  - Fortran: gfortran
- Editors
  - vi / gvim
  - emacs / xemacs
- Building:
  - Make
- Remote connections:
  - ssh
  - scp

- Freeware software
- Available on most unix systems
- Can be installed under Windows (cygwin)
- Can use on remote computer without graphic environment
- Documentation available through man pages or online

Some basic Linux commands

Account management

- . : current directory
- .. : parent directory
- bash / csh / tcsh: shells
- ./xxx : execute xxx command
- alias: rename command
- passwd: change password
- man: print command manual page
- xman: GUI version of man

Environment variables:

- HOME: home dir
- setenv/unsetenv: csh / tcsh
- export: bash
- PATH: command search path
- .bashrc / .csh : shell config files

File manipulation

- cp: copy file
- mv: move / rename file / dir
- rm: remove file / dir
- scp: secure shell copy (transfer)
- ssh: secure shell connection
- find: fine file / directory
- tar: archive (combine) files
- gzip/gunzip: compress files
- zip/unzip: archive & compress files
- chmod: change file/dir permissions
- chown: change file/dir ownership
- ln –s : make soft link

More Linux Commands

Directory manipulation

- cd: change directory
- pwd: print current (working) dir
- mkdir: make directoy
- ls: list directory contents
- ~: home directory

ASCII file manipulation

- cat: print file contents
- more/less: browse file contents
- head: print beginning of file
- tail: print ending of file
- wc: count lines/words in file
- grep: search expression in file
- awk/gawk: search/process files
- diff: print differences between 2 files
- mgdiff: GUI version of diff
- vi/vim: edit file with vi/vim
- gvim: GUI version of vim
- emacs: edit file with emacs
- xemacs: GUI version of emacs
- zcat / zgrep / zless / zdiff / zmore: operations on compressed files

More Linux Commands

Compiling

- gcc: C compiler
- g++: C++ compiler
- gfortran: fortran / fortran90 compiler
- make: build code using Makefile

Process control

- ps: list processes
- jobs: list running or stopped jobs
- top: display running processes
- kill: terminate process
- ctrl+C: kill process
- ctrl+Z: suspend process
- bg: send process to background
- fg: send process to foreground
- time: time a command

Commands

- ./xxx : execute xxx command
- xxx ; yyy: execute xxx then yyy
- xxx & : execute xxx in background
- xxx > out : redirect stdout
- xxx >> out: redirect stdout (append)
- xxx >& err : redirect stderr
- ((xxx < inp) > out) >& err & : redirect and exec in background
- xxx < inp : redirect stdin
- xxx | yyy : pipe xxx into yyy
- x* : file names starting with x
- x?: file names starting with x and one extra character
- x[0-5]: file names x0, x1, …, x5

Unix programming

Script files:

- Set variables
- Set environment variables
- Execute shell commands
- Manipulate files
- Manipulate processes
- Loops / conditional statements
- …

Syntaxis depends on shell:

- sh
- ksh
- csh / tcsh
- bash

Other scripting languages:

- python
- perl
- awk / gawk
- tcl / tk
- …

Example Makefile

```
EXE = run
OBJ = main.o  file.o   calc.o
LIB   = -llapack –lblas –lm –lgfortran
CMP = gcc –g –c --pedantic
LNK  = gcc

$(EXE): $(OBJ)
        $(LNK) –o $@ $(OBJ) $(LIB)

clean:
        rm –f *.o

%.o: %.c
        $(CMP) $?
```

BASIC DATA STRUCTURES

Data structures

- Data structure are containers for storing data into the computer’s memory in an organized manner

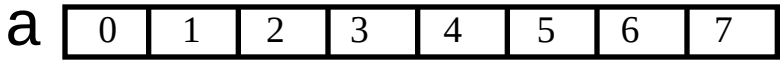
- Data structures allow logical and efficient archival and retrieval of data items

Algorithms

- Algorithms are procedures to perform a specific task
- Data structures usually have associated algorithms for efficient access to the data stored in the data structures

Arrays

- An array is a fixed number of data items that are stored contiguously and that are accessible by an index
- Arrays are defined as primitive in most programming languages (C, C++, Fortran, …)
- Arrays can be statically (size given when declaring the variable) or dynamically allocated (malloc…)
- We refer to the ith element of an array a as:
  C/C++: **`a[i]`** Fortran: **`a(i)`**



Arrays

- Traversing and array:

```
for(i=0; i<n; i++) a[i]=b[i];
```

```
do i=1,n
  a(i)=b(i)
enddo
```

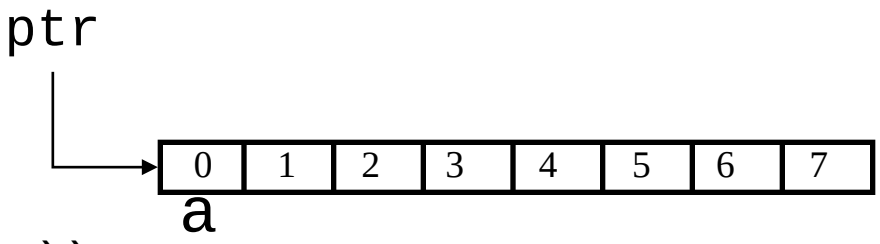
- The basic operation of an array is to access a given element in the array: **`a[i]`** **`a(i)`**

- Arrays are static data structures (they must to be reallocated if one needs to expand or shrink an array)

Pointers in C/C++

- A pointer is a variable that contains a memory location
- A pointer can be made to point to different positions in the computer's memory
- Through a pointer one can access the data stored at the location pointed to by the pointer

```
double a[100];
double *ptr;

ptr=a;
ptr=&a[0];
ptr=malloc(100*sizeof(double));

ptr+=10;
ptr=&a[10];

a[10]=5.0;
*(ptr+10)=5.0;
ptr[10]=5.0;
```

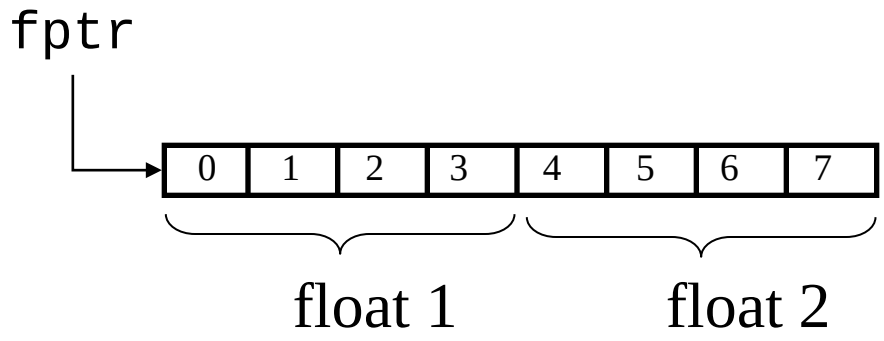


Casting Pointers

- The pointer arithmetics takes into account the type of data being pointed to (jumps in steps of sizeof(*ptr))

- Pointers can be ‘cast’ to access the memory in a different manner:

```
float  fptr[1000];
char  *cptr;

cptr=(char*) dptr;
```



Linked lists

- A linked list is a set of items organized sequentially
- In arrays the sequential ordering is provided implicitly by the position in the array (index)
- Linked lists use an explicit arrangement in which each item is part of a “node” that also contains a “link” to the next node

Linked Lists

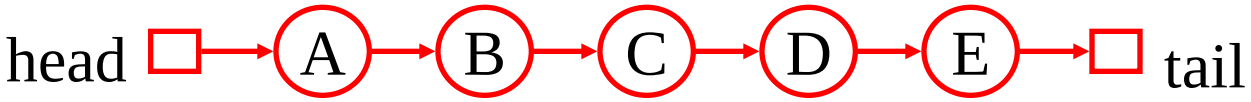
- Linked lists are defined as a primitive in some languages (Lisp, Logo) but not in most commonly used languages

- Main advantages of linked lists:
  - Dynamic data structure: can grow and shrink in size
  - Flexible in allowing items to be rearranged efficiently

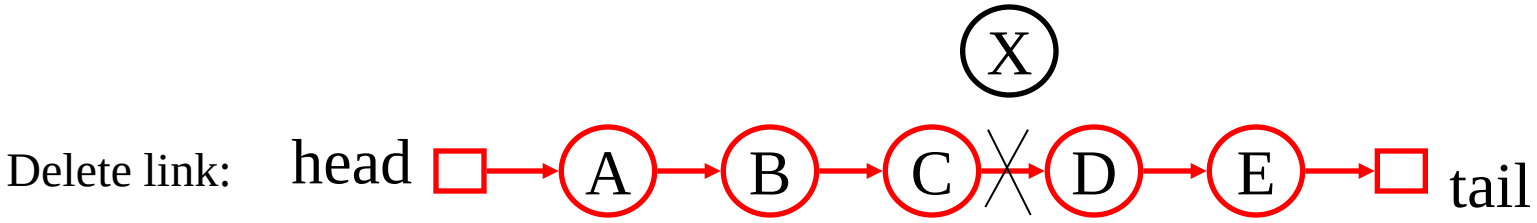
- Disadvantages:
  - Slower than arrays
  - Need to traverse the list to find an item

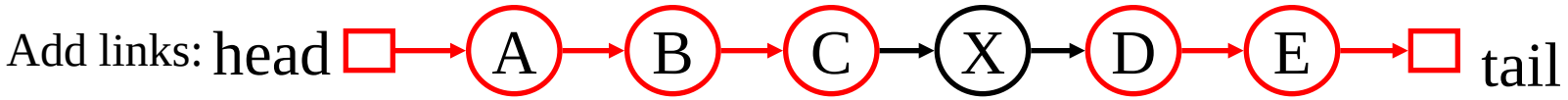
Linked lists: basic operations

- Linked list with dummy nodes

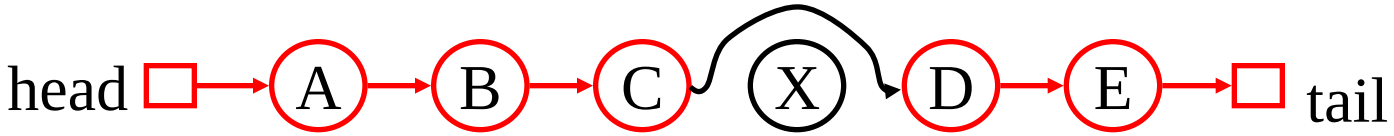


- Adding an item





- Deleting an item



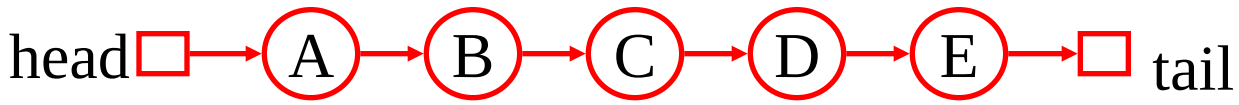
Linked lists: pointer implementation (C)

Node data structure:

```
typedef struct {
   int    item;
   nodeT *next;
} nodeT;
nodeT *head, *tail;
```

Initialization function:

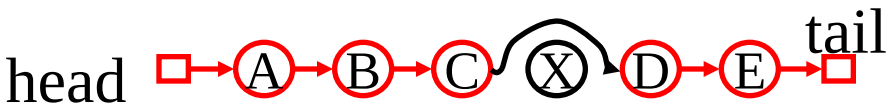
```
void init() {
  head=(nodeT*)malloc(sizeof(nodeT));
  tail=(nodeT*)malloc(sizeof(nodeT));
  head->next=tail; tail->next=tail;
}
```



Linked lists: pointer implementation (C)

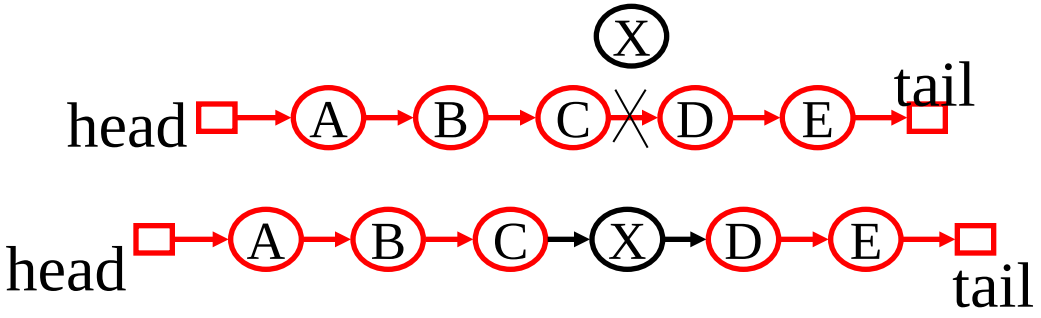
Deletion operation:

```
void deleteNext(struct node *n) {
   n->next=n->next->next;
}
```



Insertion operation:

```
nodeT *insertAfter(int item,struct node *n) {
   nodeT *x=(nodeT*)malloc(sizeof(*x));
   x->item=item;
   x->next=n->next;
   n->next=x;
   return x;
}
```



Linked lists: array implementation (C)

Data structure:

```
int item[max+2], next[max+2]; // parallel arrays
int head=0, tail=1, x=2;  // x: next unused position
```

Initialization function:

```
void init() {
   next[head]=tail; next[tail]=tail;
}
```

Linked lists: array implementation (C)

Deletion operation:

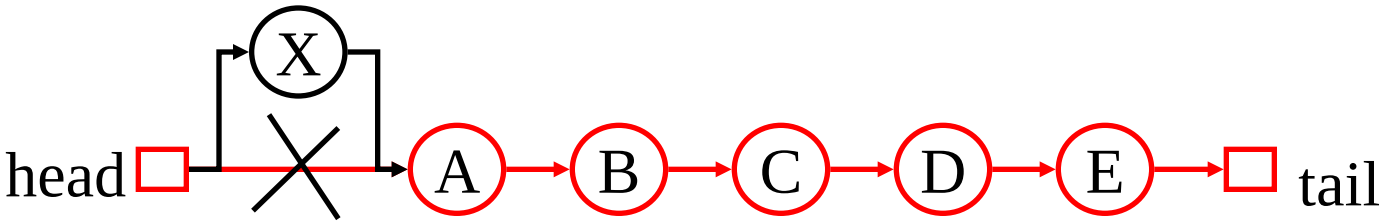
```
void deleteNext(int n) {
   next[n]=next[next[n]];
}
```

Insertion operation:

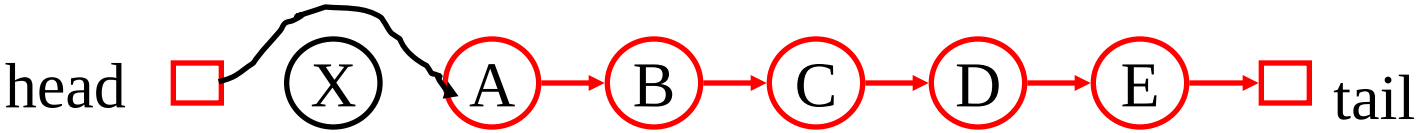
```
int insertAfter(int i,int n) {
   item[x]=i; next[x]=next[n];
   next[n]=x;
   return x++;
}
```

Stacks

- Restricted access data structure: LIFO (last in first out)
- Can push an element to the top of the stack:



- Can pop an element from the top of the stack:



Stacks: pointer implementation (C)

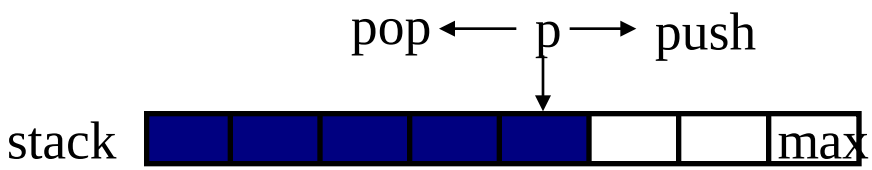
```
typedef struct {
   int    item;
   nodeT *next;
} nodeT;
nodeT *head,*tail;


void init() {
  head=(nodeT*)malloc(sizeof(nodeT));
  tail=(nodeT*)malloc(sizeof(nodeT));
  head->next=tail; head->item=0;
  tail->next=tail;
}
```

Stacks: pointer implementation (C)

```
void push(int i) {
   nodeT *n=(nodeT*)malloc(sizeof(nodeT));
   n->item=i; n->next=head->next;
   head->next=n;
}

int pop() {
   int x;
   nodeT *first=head->next;
   head->next=first->next;
   x=first->item;
   free(first);
   return x;
}
```

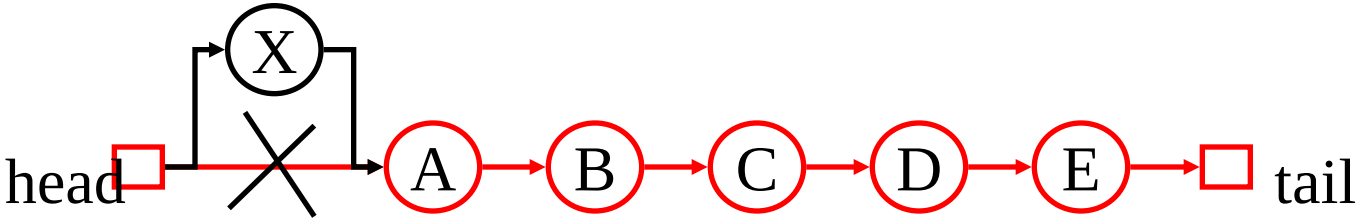
Stacks: array implementation (C)

```
int stack[max+1],p=0;

void init(int i) {
   stack[p++]=i;
}

void push(int i) {
   stack[p++]=i;
}

int pop() {
   return stack[--p];
}
```

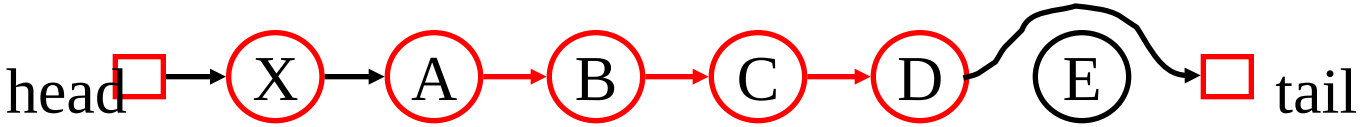


Queues

- Restricted access data structure: FIFO (first in first out)
- Can put or insert an element at the beginning of the queue:

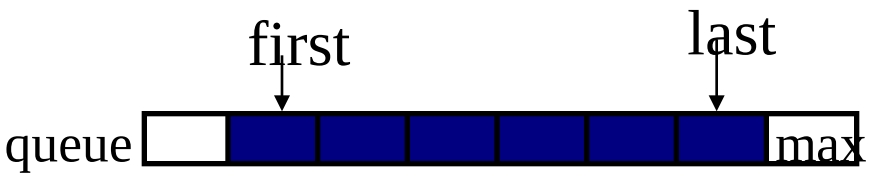


- Can get or extract and element from the end of the queue:



Queues: array implementation

```
int queue[max+1],first=0,last=0;

void put(int i) {
   queue[last++]=i;
   if( last>max ) tail=0;
}

int get() {
   int i=queue[first++];
   if( first>max ) first=0;
   return i;
}

int empty() {return first==last;}
```



FLOATING POINT NUMBER REPRESENTATION

Floating point arithmetic

- Representing real numbers with the IEEE 754 Standard

	Sign	Exponent	Mantissa
Single precision	1 bit	8 bits	23 bits
Double precision	1 bit	10 bits	53 bits

- Sign bit: 0=positive, 1=negative
- Exponent: stored in bias form, bias is usually 127
- Mantissa: an implicit 1 is included in the mantissa, which are usually between 1 and 2, with the implied 1. The implicit leading 1 is dropped for very small numbers (0 exponent)

Binary numbers

0	0.00000
1	1.00000
2	10.00000
3	11.00000
4	100.00000
5	101.00000
8	1000.00000
16	10000.00000

32	100000.00000
37	100101.00000
0.5	0.10000
0.25	0.01000
0.125	0.00100
0.0625	0.00010
0.03125	0.00001
0.65625	0.10101

More binary numbers

- We can express 83.65625 as 83 + 0.65625.
- In binary this becomes: 1010011.00 + 0.10101=1010011.10101
- We can also express this as:
  1010011.10101×2^0
  101001.110101×2^1
  …
  10.1001110101×2^5
  1.01001110101×2^6
- Of course in binary this would be: $1.01001110101 \times 10^{110}$
- In the IEEE standard, the bias is 127, so the exponent becomes: 127+6=133, and the leading 1 is dropped from the mantissa, so we have:

Number	Sign	Exponent	Mantissa
83.65625	0	1000 0101	0100 1110 1001 0000 0000 0000

Examples

number	sign	exponent	mantissa
1.00000	0	0111 1111	0000 0000 0000 0000 0000 000
-1.00000	1	0111 1111	0000 0000 0000 0000 0000 000
0.00000	0	0000 0000	0000 0000 0000 0000 0000 000
2.25000	0	1000 0000	0010 0000 0000 0000 0000 000
2.75000	0	1000 0000	0110 0000 0000 0000 0000 000
-2.75000	1	1000 0000	0110 0000 0000 0000 0000 000
127.25000	0	1000 0101	1111 1101 0000 0000 0000 000
128.25000	0	1000 0110	0000 0000 1000 0000 0000 000
-5235.25000	1	1000 1011	0100 0111 0011 0100 0000 000
1.90000	0	0111 1111	1110 0110 0110 0110 0110 011
2e-38	0	0000 0001	1011 0011 1000 1111 1011 101
1.17549421e-38	0	0000 0000	1111 1111 1111 1111 1111 111
1e-38	0	0000 0000	1101 1001 1100 0111 1101 110
1e-39	0	0000 0000	0001 0101 1100 0111 0011 000
1e-41	0	0000 0000	0000 0000 0011 0111 1100 000

Special numbers

number	sign	exponent	mantissa
+0	0	0000 0000	0000 0000 0000 0000 0000 000
-0	1	0000 0000	0000 0000 0000 0000 0000 000
denormalized	0	0000 0000	nonzero
NaN	0	1111 1111	nonzero
Inf	0	1111 1111	0000 0000 0000 0000 0000 000
-Inf	1	1111 1111	0000 0000 0000 0000 0000 000

Errors in representation

- 0.78000= 0 0111 1110 1000 1111 0101 1100 0010 100
- 0.78=0.7799999713898

- 0.95000= 0 0111 1110 1110 0110 0110 0110 0110 0110 011
- 0.95=0.9499999880791

Floating Point Math

1e6: 0 1001 0011 0010 0100 1111 1000 0000 000
1.0: 0 0111 1111 0000 0000 0000 0000 0000 000

Addition:

1e6 = $1.0010\ 0100\ 1111\ 1000\ 0000 \times 2^{20}$
+ 1.0 = $1.0000\ 0000\ 0000\ 0000\ 0000 \times 2^{0}$

Alignment / round-off:

1e6 = 1 0010 0100 1111 1000 0000. 000
+ 1.0 = 0 0000 0000 0000 0000 0001. 000

=> 1e6+1.0 = 1 0010 0100 1111 1000 0001. 000
= 1,000,001.000

Smaller numbers

Addition:

1e6 = $1.0010\ 0100\ 1111\ 1000\ 0000 \times 2^{20}$

+ 0.125 = $1.0000\ 0000\ 0000\ 0000\ 0000 \times 2^{-3}$

Alignment / round-off:

1e6 = 1 0010 0100 1111 1000 0000. 000

+ 0.125 = 0 0000 0000 0000 0000 0000. 001

=> 1e6+0.125 = 1 0010 0100 1111 1000 0000. 001

= 1,000,000.125

Even smaller numbers

Addition:

1e6 = $1.0010\ 0100\ 1111\ 1000\ 0000 \times 2^{20}$

+ 0.0625 = $1.0000\ 0000\ 0000\ 0000\ 0000 \times 2^{-4}$

Alignment / round-off:

1e6 = 1 0010 0100 1111 1000 0000. 000

+ 0.0625 = 0 0000 0000 0000 0000 0000. 000

=> 1e6+0.0625 = 1 0010 0100 1111 1000 0000. 000

= 1,000,000.000

- There is no difference before and after the addition !

Other fractions

Addition:

1e6 = $1.0010\ 0100\ 1111\ 1000\ 0000 \times 2^{20}$

+ 0.9 = $1.100\ 1100\ 1100\ 1100\ 1100 \times 2^{-1}$

Alignment / round-off:

1e6 = 1 0010 0100 1111 1000 0000. 000

+ 0.9 = 0 0000 0000 0000 0000 0000. 110 (011 001100)

=> 1e6+0.9 = 1 0010 0100 1111 1000 0000. 110

= 1,000,000.750

Storing multi-byte data in memory

Big Endian

- Most significant byte has lowest memory address
- Used in SGI (IRIX) / SUN processors

Little Endian

- Least significant byte has lowest memory address
- Used in Intel processors

Possible problems

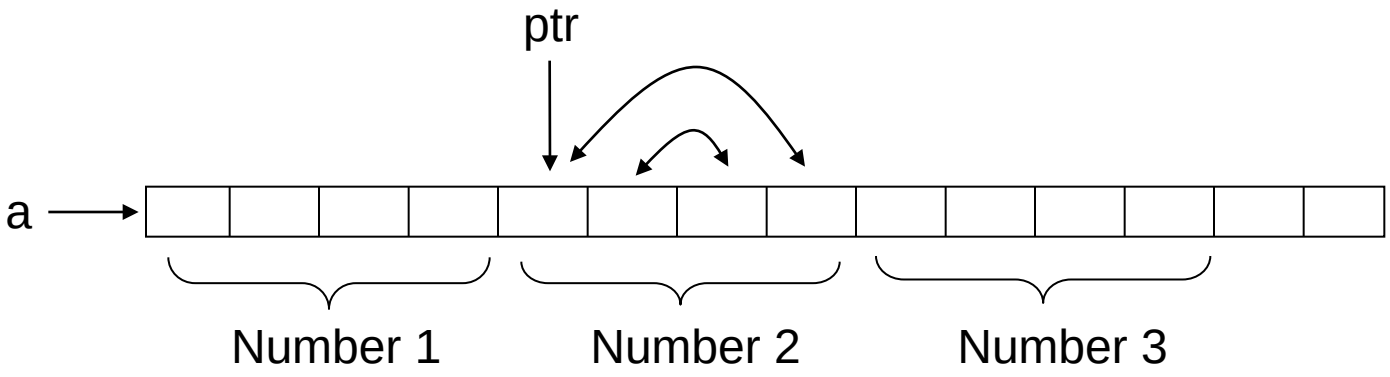
- Data written in binary format to a file may be scrambled if it is read from another computer
- Need to swap bytes

Swapping Bytes

- Use pointers to access individual bytes of each number and swap the byte order

```
float a[100];
char *ptr,c;
ptr=&a[1];
```

```
c=ptr[0]; ptr[0]=ptr[3]; ptr[3]=c;
c=ptr[1]; ptr[1]=ptr[2]; ptr[2]=c;
```



PARTIAL DIFFERENTIAL EQUATIONS

Partial differential equations

- Non-linear $\Rightarrow$ closed form solutions usually impossible
- PDE's are generally more difficult to solve than ODE's
- PDE's can be broken into three *physical* classes:
  - Equilibrium problems
  - Eigenvalue problems
  - Propagation problems
- There are different techniques for each class of problem

Classification of PDE's by characteristics

- We classify PDE's on the basis of their characteristic equations. Consider

$$a\,u_{xx} + b\,u_{xy} + c\,u_{yy} = f$$

- Under what conditions does u, u_x and u_y uniquely determine u_{xx}, u_{xy}, u_{yy} such that our equation is satisfied ?

- We can derive from this, two equations for the derivatives of u with respect to x and y:

$$d(u_x) = u_{xx}\,dx + u_{xy}\,dy$$
$$d(u_y) = u_{xy}\,dx + u_{yy}\,dy$$

- We are seeking characteristic directions along which the equations only involve total differentials (i.e. *characteristics*)

Characteristic equations

- Writing the previous three equations in matrix form:

$$\begin{bmatrix} a & b & c \\ dx & dy & 0 \\ 0 & dx & dy \end{bmatrix} \begin{bmatrix} u_{xx} \\ u_{xy} \\ u_{yy} \end{bmatrix} = \begin{bmatrix} f \\ d(u_x) \\ d(u_y) \end{bmatrix}$$

- The determinant of this matrix is

$$\det = a(dy)^2 - b\,dx\,dy + c(dx)^2$$

- The solution of the PDE exists and is unique if the determinant of this matrix is non-zero
- If the determinant is zero, multiple solutions exist

Classification of PDE's

- The behavior of the quadratic equation (det=0) is closely related to the character of the PDE

- Three types of PDE's are defined:
  - $b^2-4ac>0$: hyperbolic equation (2 real characteristics exist)
  - $b^2-4ac=0$: parabolic equation (1 real characteristic exist)
  - $b^2-4ac<0$: elliptic equation (characteristics are complex)

Classification of PDE systems

- For systems of PDE's the classification is much more involved. One way to proceed is to write the system of equations in matrix form: **Lu**=**f**

  with **L** the differential operator and **u** the vector of unknowns

- The system of equations is characterized by its eigenvalues:

1. Hyperbolic: if *n* real roots of the characteristic equation
2. Parabolic: if $1 \leq m \leq n-1$ real roots, and no complex roots
3. Elliptic: no real roots

- The eigenvalues are determined from the characteristic equation: det(**L**)=0

Classification by Fourier analysis

- The same characteristic equation can be obtained from a Fourier analysis of the system of equations
- In this case, the roots have a different physical interpretation, but the PDE classification remains the same
- The Fourier approach is useful for systems of PDE's of order higher than first-order
- The Fourier approach indicates the expected from of the solution: oscillatory, exponential growth, etc. (useful for stability analysis)

Example

- Suppose the solution of the homogeneous 2nd order equation

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} = 0$$

  is sought of the form

$$u(x,y) = \frac{1}{4\pi^2}\sum_{j=-\infty}^{\infty}\sum_{k=-\infty}^{\infty} a_{jk}\,\exp[i\alpha_j x]\exp[i\beta_j y]$$

  substituting into the equation gives

$$-A\alpha^2 - B\alpha\beta - C\beta^2 = 0 \quad \Rightarrow \quad A(\alpha/\beta)^2 + B(\alpha/\beta) + C = 0$$

- This is a characteristic polynomial equivalent to the eigenvalue analysis, and the nature of the PDE depends on the nature of the roots of this characteristic equation

PDE types

- Hyperbolic equations: describe time-dependent, conservative physical processes that are *not* evolving towards a steady state

$$\frac{\partial^2 u}{\partial t^2} = \alpha \frac{\partial^2 u}{\partial x^2} + \beta \frac{\partial^2 u}{\partial y^2} + \gamma \frac{\partial^2 u}{\partial z^2}$$

- Parabolic equations: describe time-dependent, dissipative physical processes that are evolving towards a steady state

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + \beta \frac{\partial^2 u}{\partial y^2} + \gamma \frac{\partial^2 u}{\partial z^2}$$

- Elliptic equations: describe systems that have already reached a steady state or equilibrium and thus are time-independent

$$\alpha \frac{\partial^2 u}{\partial x^2} + \beta \frac{\partial^2 u}{\partial y^2} + \gamma \frac{\partial^2 u}{\partial z^2} = g$$

Hyperbolic PDE's

- *Advection equation*:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \quad \Rightarrow \quad u(x,t) = u_0(x - ct)$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{c}\cdot\nabla\mathbf{u} = 0 \quad \Rightarrow \quad \mathbf{u}(\mathbf{x},t) = \mathbf{u}_0(\mathbf{x} - \mathbf{c}t)$$

Propagation with velocity c, no damping

- *Wave equation*:

$$\frac{\partial^2 u}{\partial t^2} - c^2\frac{\partial^2 u}{\partial x^2} = 0 \quad \Rightarrow \quad u(x,t) = u_0 e^{\pm i(kx - \omega t)}$$

$$\frac{\partial^2 \mathbf{u}}{\partial t^2} - c^2\nabla^2\mathbf{u} = 0 \quad \Rightarrow \quad \mathbf{u}(\mathbf{x},t) = \mathbf{u}_0 e^{\pm i(\mathbf{k}\cdot\mathbf{x} - \omega t)}$$

- Hyperbolic PDE's are associated with *propagation* problems when *no dissipation* is present (no attenuation of the wave amplitudes). Discontinuities are transported or advected

Parabolic PDE's

- *Diffusion equation*:

$$\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2} \quad \Rightarrow \quad u(x,t) = u_0\, e^{-\lambda t}\, e^{\pm ikx}$$

$$\frac{\partial \mathbf{u}}{\partial t} = k\nabla^2 \mathbf{u} \quad \Rightarrow \quad \mathbf{u}(\mathbf{x},t) = \mathbf{u}_0 e^{-\lambda t}\, e^{\pm ikx}$$

Propagation and decay in time => damping

- Parabolic PDE's are associated with *propagation* problems that *include dissipative mechanisms*
- Parabolic PDE's are characterized by solutions that march forward in time but diffuse in space
- Discontinuities are diffused => solution becomes continuous

Elliptic PDE's

- *Laplace's equation*:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0 \quad \Rightarrow \quad \phi(x, y) = e^{\pm i\alpha x}\, e^{\pm i\beta y}$$

$$\nabla^2 \phi = 0 \quad \Rightarrow \quad \phi(\mathbf{x}) = e^{\pm i\mathbf{k}\cdot\mathbf{x}}$$

- Elliptic PDE's are associated with *steady state* problems
- The most important feature is that disturbances introduced in the interior of the domain influence all other points. In contrast, hyperbolic and parabolic PDE's can be solved by marching progressively the initial conditions

GRIDS

Discretization

- The solution of PDE's are continuous functions of the independent variables
- In order to represent continuous functions on a computer (an inherently discrete device) the computational domain as well as the equation operators must be discretized
- Usually, temporal and spatial discretization are performed independently
- Functions and operators are discretized in space using *computational grids*

Computational grids types

Grid topology:

- Structured grids
  - Cartesian grids
  - Curvilinear grids
- Unstructured grids
  - Triangles, tetrahedra
  - Quadrilaterals, hexahedra
  - Mixed elements
- Mixed
  - Mini structured, macro unstructured

Boundary representation:

- Body conforming grids
- Embedded grids
- Overlapping grids

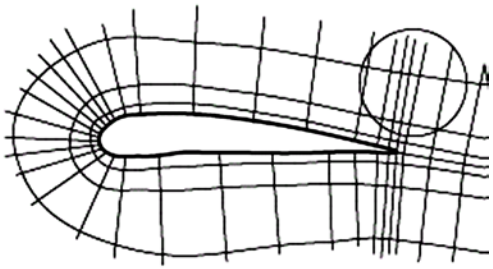
Element topology:

- Conforming
- Non-conforming

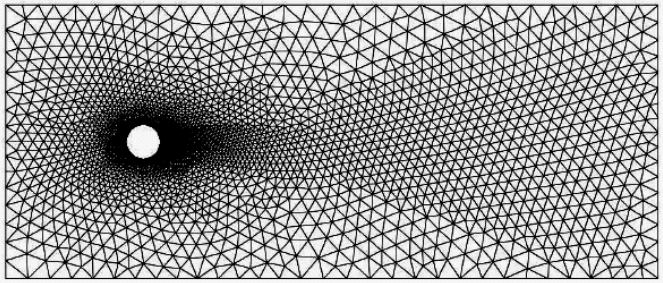
Global topology:

- Multi-block grids
- Overlapping grids

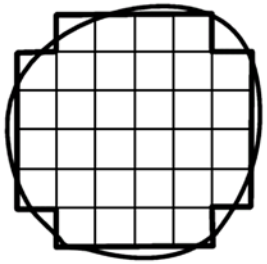
Grid examples



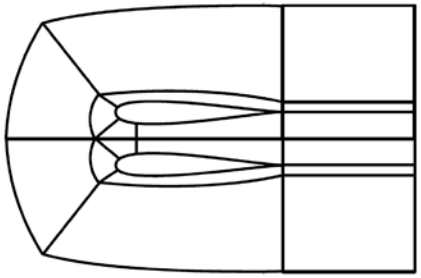
Body-conforming,
curvilinear structured grid



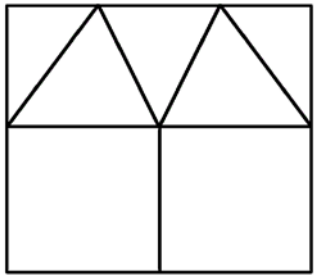
Body conforming unstructured grid



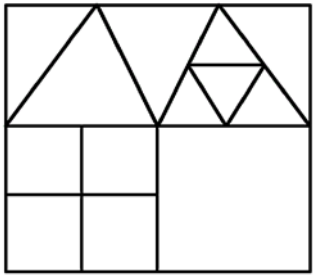
Non-body conforming,
Cartesian grid (structured)



Multi-block grid



Conforming elements



Non-conforming elements

Discretization techniques

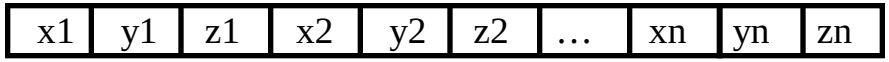
- Finite difference methods
- Finite element methods
- Finite volume methods
- Particle methods
- Meshless or finite points methods

Geometry Representation

- Representation of points
  - Coordinate arrays
- Representation of segments
  - List of end points
- Representation of polygons
  - List of vertices
- Representation of surfaces
  - List of triangles / quads
- Representation of solids
  - List of tetrahedra / hexahedra

Storing point coordinates in memory

- Storage order:
  - Contiguous points



  - Contiguous coordinates

- The choice of storage order depends on how you will access the point coordinates in your algorithms
  => try to minimize cache misses

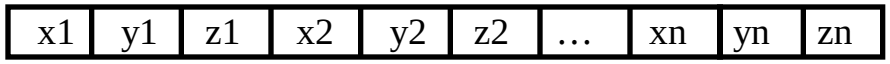
Using linear arrays

C:

```
int npoin;
double xyz[3*npoin];

xi=xyz[i*3];
yi=xyz[i*3+1];
zi=xyz[i*3+2];
```

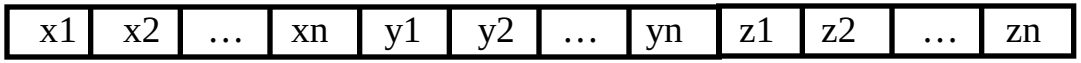
Fortran:

```
integer npoin
real*8 xyz(3*npoin)

xi=xyz(i*3)
yi=xyz(i*3+1)
zi=xyz(i*3+2)
```




```
xi=xyz[i];
yi=xyz[n+i];
zi=xyz[2*n+i];
```

```
xi=xyz(i)
yi=xyz(n+i)
zi=xyz(2*n+i)
```



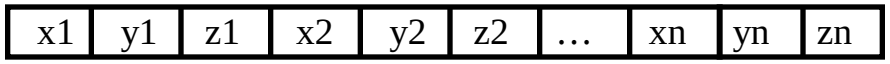
Using multidimensional arrays

C: stores by rows

```
double xyz[npoin][3];
xi=xyz[i][0];
yi=xyz[i][1];
zi=xyz[i][2];
```

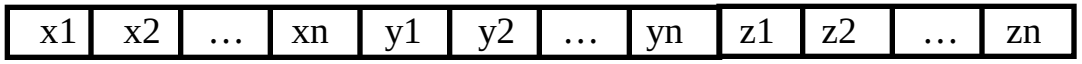
Fortran: stores by columns

```
real*8 xyz(3,npoin)
xi=xyz(1,i)
yi=xyz(2,i)
zi=xyz(3,i)
```




```
double xyz[3][npoin];
xi=xyz[0][i];
yi=xyz[1][i];
zi=xyz[2][i];
```

```
real*8 xyz(npoin,3)
xi=xyz(i,1)
yi=xyz(i,2)
zi=xyz(i,3)
```



Using structures & classes

C:

```
typedef struct {
   double x,y,z;
} ptT;

int npoin;
ptT pts[npoin];
xi=pts[i].x;
yi=pts[i].y;
zi=pts[i].z;
```

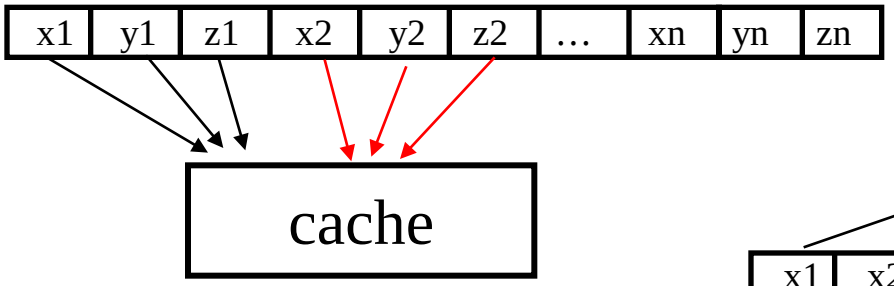
C++:

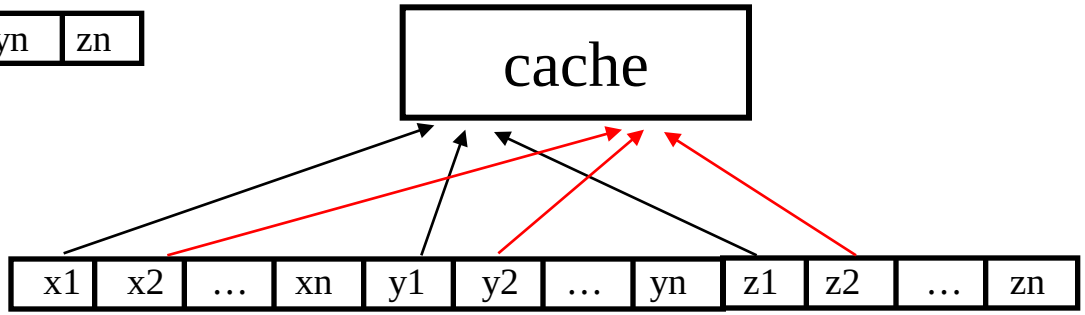
```
class ptT {
  double x,y,z;
  ptT(){x=0;y=0;z=0;}
};

ptT pts[npoin];
xi=pts[i].x;
yi=pts[i].y;
zi=pts[i].z;
```

Example: memory access

```
double xyz[npoin*3];

for(i=0; i<npoin; i++) {
  x=xyz[i*3  ];
  y=xyz[i*3+1];
  z=xyz[i*3+2];
  d[i]=sqrt(x*x+y*y+z*z);
}
```

```
double xyz[npoin*3];

for(i=0; i<npoin; i++) {
  x=xyz[i];
  y=xyz[i+npoin];
  z=xyz[i+2*npoin];
  d[i]=sqrt(x*x+y*y+z*z);
}
```





Representing segments

C:

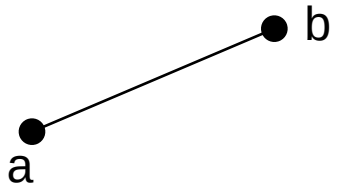
```
int nseg;
int lnode[2*nseg];
ai=lnode[i*2];
bi=lnode[i*2+1];
```

Fortran:

```
integer nseg
integer lnode(2,nseg)
ai=lnode(1,i)
bi=lnode(2,i)
```

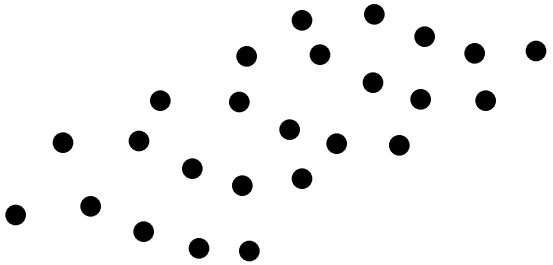
lnode: segment connectivity array
(list of the two nodes of a segment)

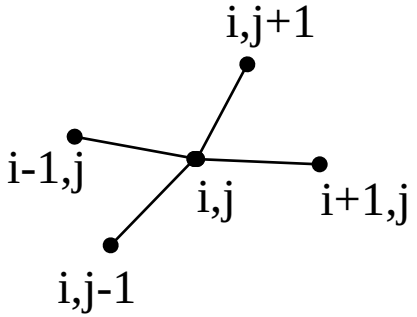
Must start at 0 in C
Must start at 1 in fortran

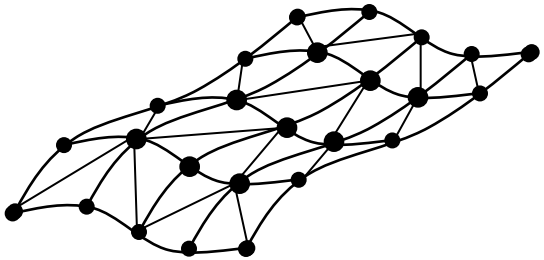


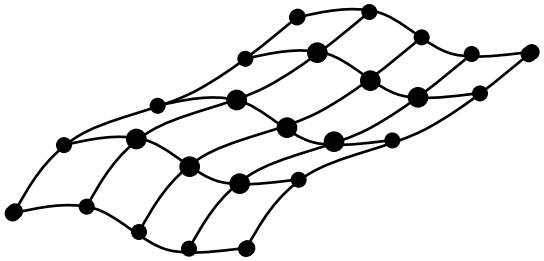
Representing Surfaces in 3D

- Cloud of points
- Structured curvilinear grids
- Triangular grids
- Quadrilateral grids



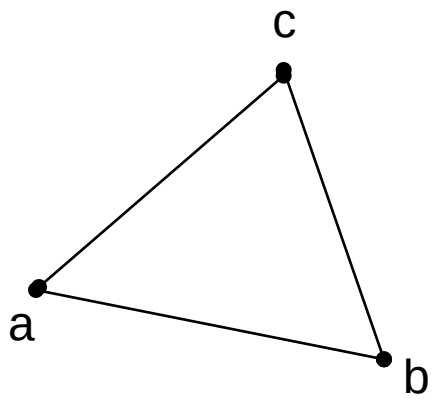


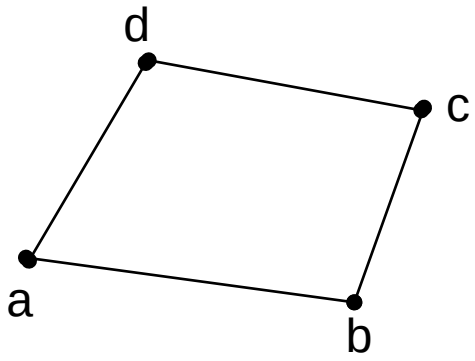




Representing surfaces (triangles and quads)

```
int npoin;                  // nr of points
double xyz[3*npoin];        // point coords
int nelem;                  // nr of elements
int nnode;                  // nr of nodes per element
int lnode[nnode*nelem]; // connectivity array
ai=lnode[nnode*i];          // nodes of an element
bi=lnode[nnode*i+1];
ci=lnode[nnode*i+2];
```



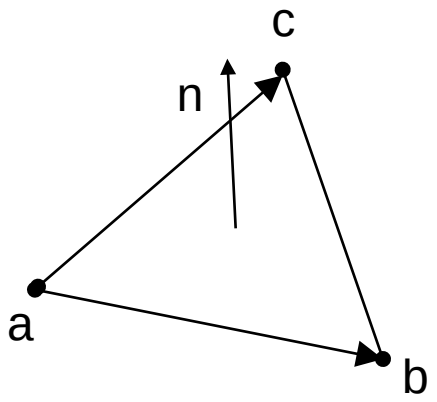


Element normal and area

Area weighted normal

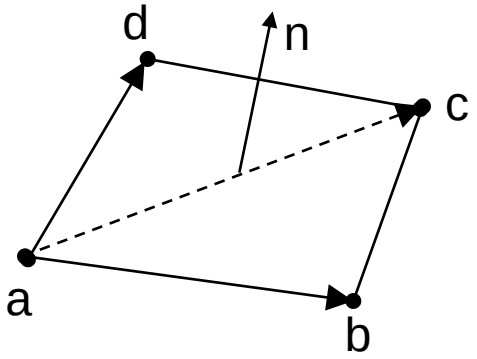
Triangle: $\mathbf{n} = \frac{1}{2}\mathbf{x}_{ba} \times \mathbf{x}_{ca}$

Quad: $\mathbf{n} = \frac{1}{4}\left(\mathbf{x}_{ba} \times \mathbf{x}_{ca} + \mathbf{x}_{ca} \times \mathbf{x}_{da}\right)$



Element area

$$A = |\mathbf{n}|$$



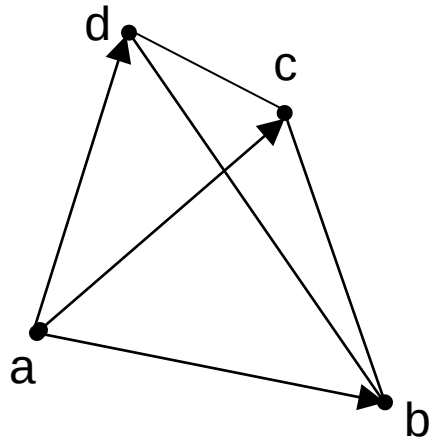
Unit normal

$$\mathbf{n}_0 = \frac{\mathbf{n}}{A}$$

Representing Solids: Tetrahedral Grids

Same data structures and algorithms for:

- Element connectivity (nnode=4)
- Elements surrounding points
- Points surrounding points
- Elements surrounding elements



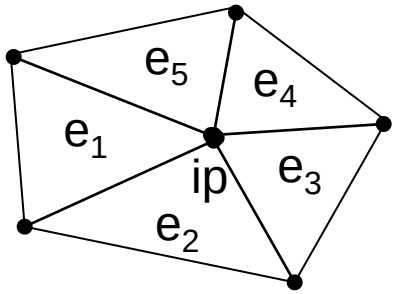
Element volume

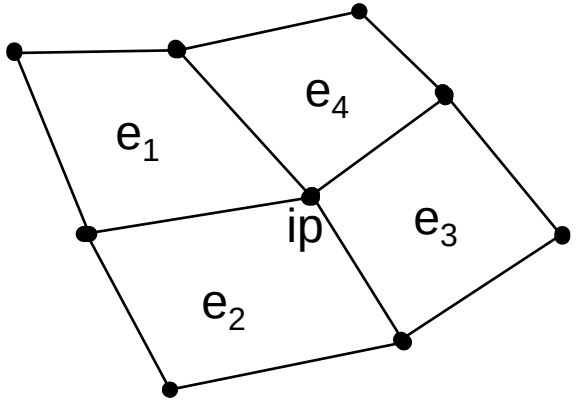
$$v = \frac{1}{4}\left(\mathbf{x}_{\mathbf{ba}} \times \mathbf{x}_{\mathbf{ca}}\right) \cdot \mathbf{x}_{da}$$

Unstructured grids

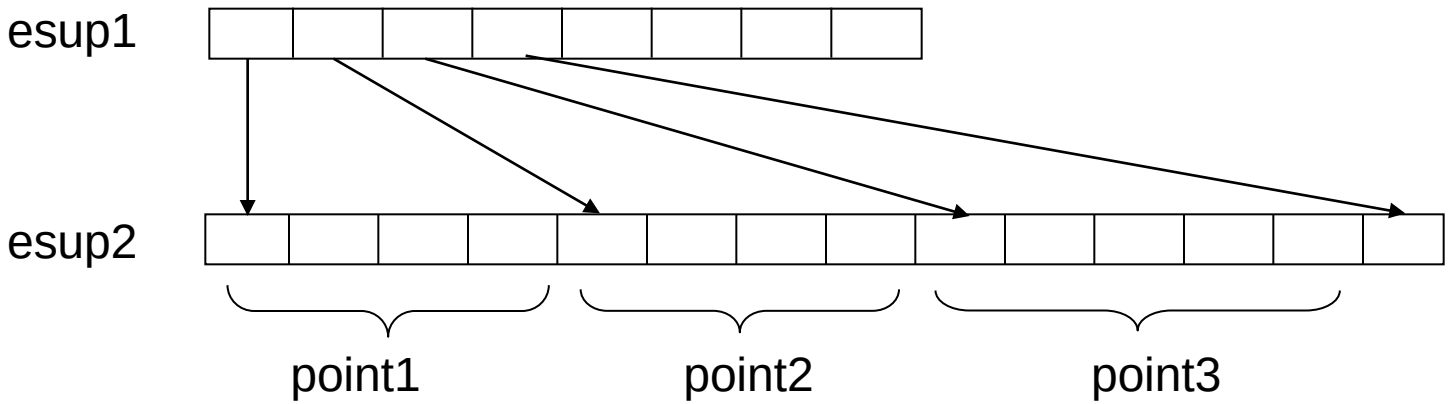
- Mesh representation
  - List of points + list of elements
- Derived data structures
  - List of elements surrounding each point
  - List of points connected to each point
  - List of neighbor elements to each element
  - List of boundary elements / points
- These derived data structures are many times useful for developing efficient algorithms on unstructured grids

Elements surrounding points






```
int esup1[npoin+1];
int esup2[max_store];
```



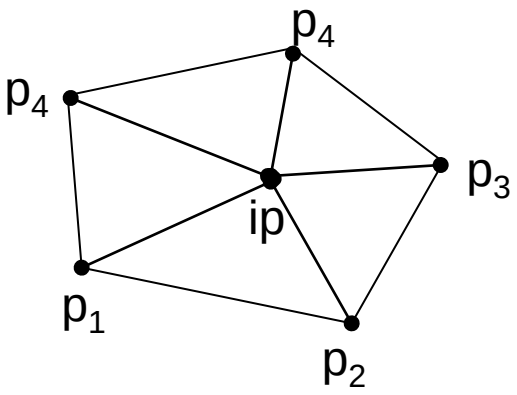
Elements surrounding points: building

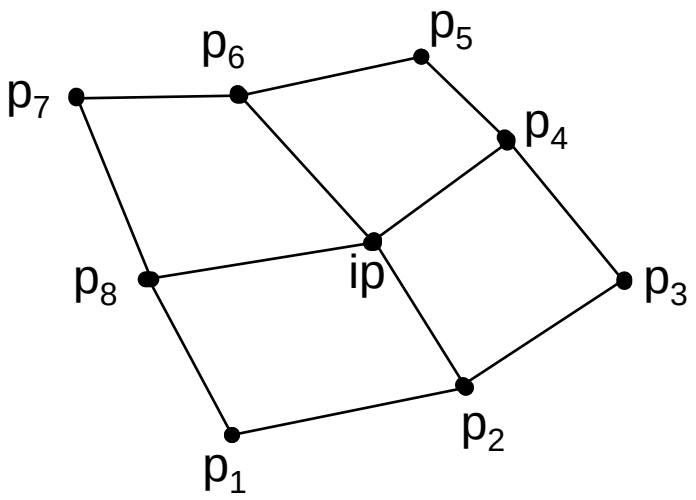
```
// count nr of elements surrounding each point
int *esup1=malloc(npoin*sizeof(int));
for(i=0; i<nelem; i++)
   for(j=0; j<nnode; j++) esup1[lnode[3*i+j]+1]++;

// initialize pointers and allocate storage
for(i=1; i<npoin-1; i++) esup1[i]+=esup1[i-1];
int *esup2=malloc(esup1[npoin]*sizeof(int));

// store elements surrounding each point
for(i=0; i<nelem; i++)
   for(j=0; j<nnode; j++)
      esup2[esup1[lnode[3*i+j]]++]=i;
for(i=npoin; i>0; i--) esup1[i]=esup1[i-1];
esup1[0]=0;
```

Elements surrounding points: usage

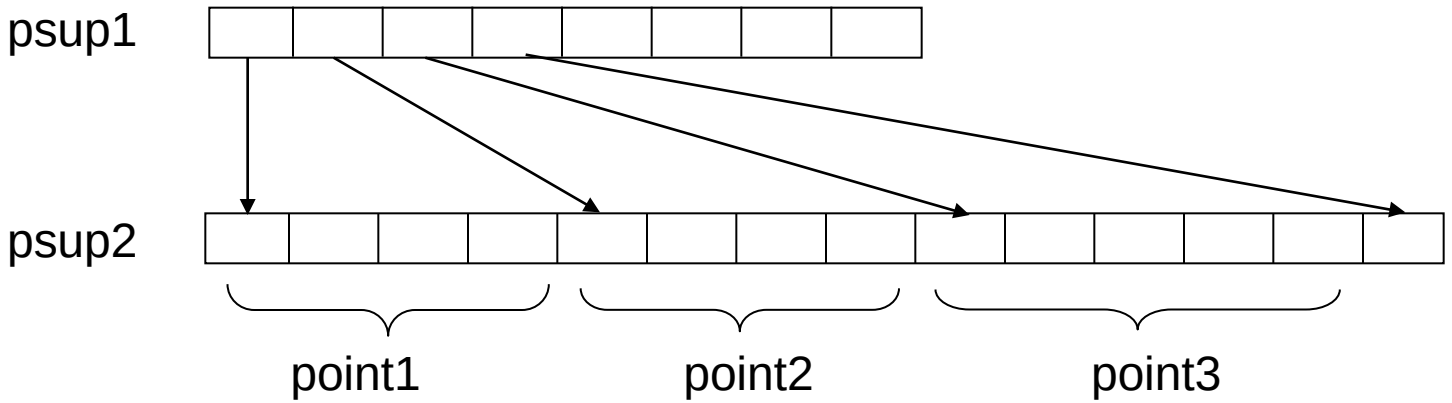
```
for(i=0; i<npoin; i++) {      // loop over points (i)
   nel=esup1[i+1]-esup1[i];  // num  elem surr i
   lel=&esup2[esup1[i]];     // list elem surr i
   for(j=0; j<ne1; j++) {     // loop over surr elem
      iel=lel[j];            // get next elem surr I
      do_work...
   }
}
```

Points surrounding points






```
int psup1[npoin+1];
int psup2[max_store];
```



Points surrounding points: building

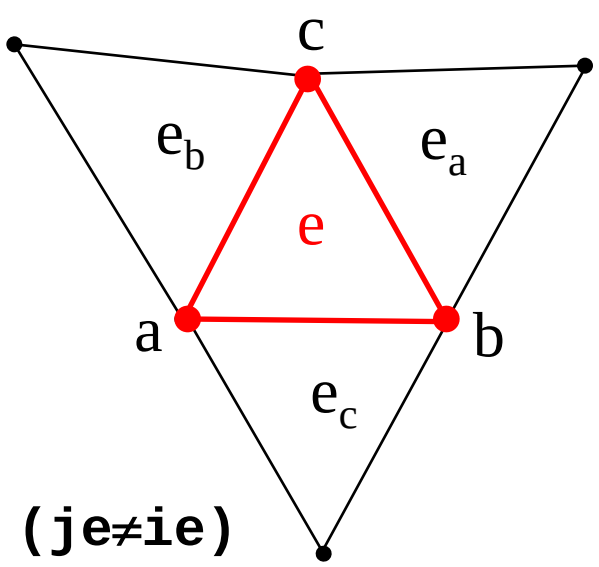
```
integer psup1(npoin+1),psup2(max) ! psup arrays
integer esup1(npoin+1),esup2(max) ! assumed done
integer ltmp(npoin)               ! temp array
initialize ltmp(1:npoin)=0
initialize psup1(1:npoin+1)=0
istor=0
do ipoin=1,npoin                            ! loop over pts
  do iesup=esup1(ipoin)+1,esup2(ipoin+1)    ! loop el surr pt
    ielem=esup2(iesup)                      ! get el surr ip
    do inode=1,nnode                        ! Loop over nodes
      jpoin=lnode(inode,ielem)              ! get point
      if(jpoin.ne.ipoin. and .ltmp(jpoin).ne.ipoin) then
        istor=istor+1                       ! updt strg count
        psup2(istor)=jpoin                  ! store point
        ltmp(jpoin)=ipoin                   ! mark point
      endif
    enddo
  enddo
  psup2(ipoin+1)=istor                      ! Updt strg count
enddo
```

Points surrounding points: usage

```
Do ipoin=1,npoin          ! loop over points
  jp1=psup1(ipoin)+1      ! first point surr ipoin
  jp2=psup1(ipoin+1)      ! last  point surr ipoin
  do jp=jp1,jp2           ! loop over pts surr ipoin
    jpoin=psup2(jp)       ! gather point
    do_work...
  enddo
enddo
```

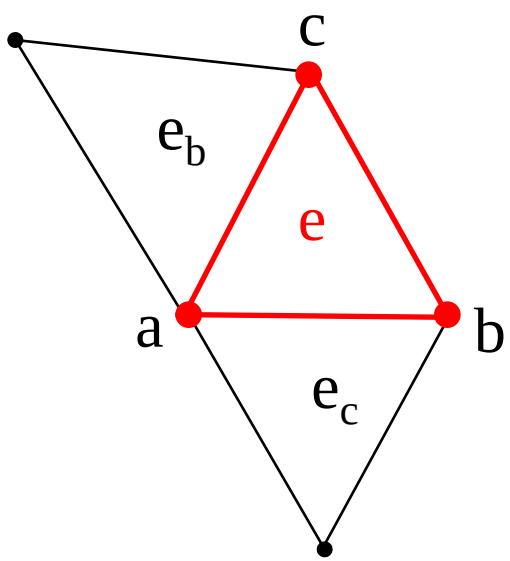
Elements surrounding elements

```
int esuel[nnode*nelem];

init esuel[1:nnode*nelem]=-1
loop over elements (ie)
 loop over nodes of ie (a)
  get b=next node of ie
  get c=next node of ie
  loop over elements surrounding b (je≠ie)
   if(je and ie have 2 nodes in common) then
    store je as neighbor opposite to a
   endif
  endloop
 endloop
endloop
```



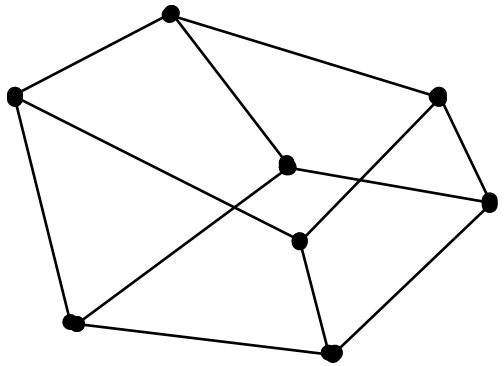
Boundary elements

```
nboun=0                      ! nr boundary elements
lboun(1:nboun)=0             ! list boundary elements
loop over elements (ie)
  loop over neighbor elements (je)
    if(je<0) then            ! neighbor does not exist
      lboun(nboun)=je        ! add element to list
      nboun=nboun+1          ! Update counter
    endif
  endloop
endloop
```



Derived data structures

- The previous data structures and algorithms can be used with no modifications for tetrahedral grids:
  - Element surrounding points
  - Points surrounding points
  - Elements surrounding elements
  - Boundary elements
- These need to be modified for Hexahedral elements



FINITE DIFFERENCES METHOD

Finite difference method

- This method is the oldest of the methods applied to PDE's (Euler 1768).
- It is based on the properties of the Taylor expansions and on the straightforward application of the definition of derivatives
- It is the simplest method to apply but requires a high degree of regularity of the grids, and in particular, the grid must be structured
- The basic idea of this method is quite simple: estimate the derivatives by the ratio of two differences according to the definition of the derivative

Basics of the FD method

- For a function *u(x)* the derivative at a point *x* is defined as:

$$u_x \equiv \frac{\partial u}{\partial x} = \lim_{\Delta x \to 0} \frac{u(x + \Delta x) - u(x)}{\Delta x}$$

- If Δx is small but finite, the expression on the right hand side is an approximation to the exact value of u_x

- The approximation improves as we reduce Δx, but for any finite value of Δx a *truncation* error is introduced that tends to zero when Δx goes to zero

- The power of Δx with which the error goes to zero is called the *order of the difference approximation*

Difference approximations

- The whole concept of finite difference approximation is based on the properties of the Taylor expansions
- Developing $u(x + \Delta x)$ we obtain

$$u(x+\Delta x) = u(x) + \Delta x\, u_x(x) + \frac{\Delta x^2}{2!} u_{xx}(x) + \cdots$$

and therefore

$$\frac{u(x+\Delta x) - u(x)}{\Delta x} = u_x(x) + \frac{\Delta x}{2} u_{xx}(x) + \cdots$$

- This approximation of u_x is of first order in Δx , indicating that the truncation error goes to zero like the first power in Δx

Finite difference formulas

- Consider a 1D space, discretized with *N* uniformly distributed mesh points x_i with $i=1,\ldots,N$.
- We denote by $u_i=u(x_i)$ the value of the function at point *i* and Δx the spacing between mesh points
- The following formulas can be derived for the first derivative:

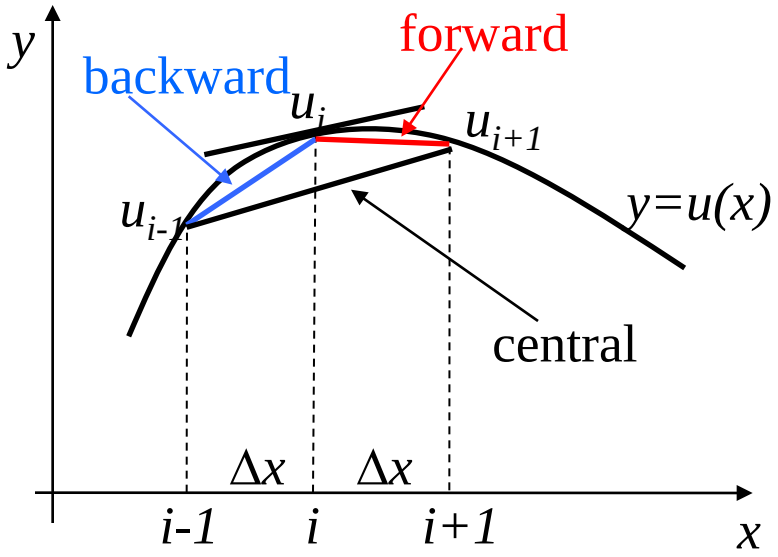
$$forward\ difference: \quad (u_x)_i = \frac{u_{i+1}-u_i}{\Delta x} + O(\Delta x)$$

$$backward\ difference: \quad (u_x)_i = \frac{u_i-u_{i-1}}{\Delta x} + O(\Delta x)$$

$$central\ difference: \quad (u_x)_i = \frac{u_{i+1}-u_{i-1}}{2\Delta x} + O(\Delta x^2)$$

Finite difference

- It can be easily verified from the Taylor expansion of u_{i+1} and u_{i-1} that the *one sided formulas* are of first order while the *central* difference formula is of second order in Δx



Difference formulas with an arbitrary number of points

- Difference formulas for the first derivative can be constructed with any number of adjacent points, with the order of the approximation increasing with the number of points
- Example: one-sided, 2nd order formula containing *upstream* points *i-2, i-1, i* can be obtained by an expression of the form

$$(u_x)_i = \frac{au_i + bu_{i-1} + cu_{i-2}}{\Delta x} + O(\Delta x^2)$$

- The coefficients (*a,b,c*) are obtained from the Taylor expansions of u_{i-2} and u_{i-1} around u_i:

$$u_{i-2} = u_i - 2\Delta x(u_x)_i + 2\Delta x^2 (u_{xx})_i - \frac{(2\Delta x^3)}{6}(u_{xxx})_i + \cdots$$

$$u_{i-1} = u_i - \Delta x(u_x)_i + \frac{\Delta x^2}{2}(u_{xx})_i - \frac{\Delta x^3}{6}(u_{xxx})_i + \cdots$$

Difference formulas with an arbitrary number of points

- From these expansions, we get:

$$c\,u_{i-2} + b\,u_{i-1} + a\,u_i =$$

$$(a+b+c)u_i - \Delta x(2c+b)(u_x)_i + \frac{\Delta x^2}{2}(4c+b)(u_{xx})_i + O(\Delta x^3)$$

- Identifying terms in the proposed difference formula, we get:

$$a+b+c=0$$

$$(2c+b)=-1$$

$$(4c+b)=0$$

- The 2nd order one sided formula is

$$(u_x)_i = \frac{3u_i - 4u_{i-1} + u_{i-2}}{2\Delta x} + O(\Delta x^2)$$

- This is a general procedure for obtaining FD formulas with an arbitrary number of points and an given order of accuracy

2nd order formulas for the first derivative

- Backward differences

$$(u_x)_i = \frac{3u_i - 4u_{i-1} + u_{i-2}}{2\Delta x} + O(\Delta x^2)$$

- Forward differences

$$(u_x)_i = \frac{-3u_i + 4u_{i+1} - u_{i+2}}{2\Delta x} + O(\Delta x^2)$$

- Central differences

$$(u_x)_i = \frac{u_{i+1} - u_{i-1}}{2\Delta x} + O(\Delta x^2)$$

$$(u_x)_i = \frac{-u_{i+2} + 8u_{i+1} - 8u_{i-1} + u_{i-2}}{2\Delta x} + O(\Delta x^4)$$

Higher order derivatives

- Finite difference formulas for higher order derivatives can be obtained by applying the same technique to first derivatives, then to second, etc:

- First order formulas for the second derivative:
- Forward differences:

$$(u_{xx})_i = \frac{u_{i+2} - 2u_{i+1} + u_i}{\Delta x^2} + O(\Delta x)$$

- Backward differences:

$$(u_{xx})_i = \frac{u_i - 2u_{i-1} + u_{i-2}}{\Delta x^2} + O(\Delta x)$$

2nd order formulas for the 2nd derivative

- Forward differences

$$(u_{xx})_i = \frac{2u_i - 5u_{i+1} + 4u_{i+2} - u_{i+3}}{\Delta x^2} + O(\Delta x^2)$$

- Backward differences

$$(u_{xx})_i = \frac{2u_i - 5u_{i-1} + 4u_{i-2} - u_{i-3}}{\Delta x^2} + O(\Delta x^2)$$

- Central differences

$$(u_{xx})_i = \frac{-u_{i+2} + 16u_{i+1} - 30u_i + 16u_{i-1} - u_{-2}}{12\Delta x^2} + O(\Delta x^2)$$

$$(u_{xx})_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + O(\Delta x^2)$$

Multi-dimensional FD formulas

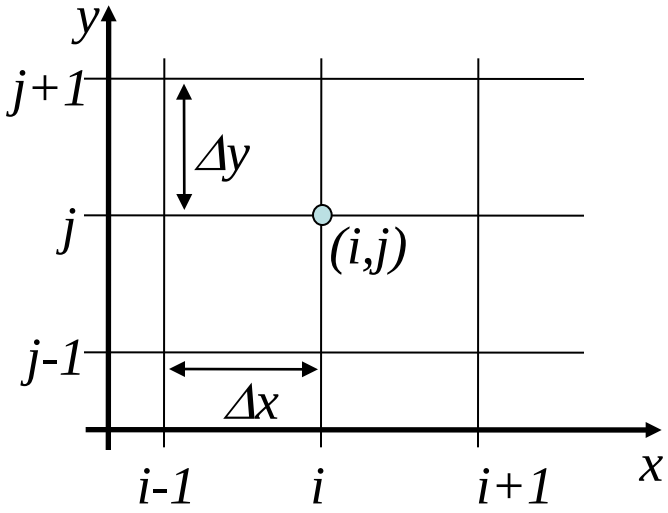
- The partial derivatives of functions of several variables can be approximated by the previously derived formulas considering each variable separately
- Examples:

$$\left(\frac{\partial u}{\partial x}\right)_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + O(\Delta x)$$

$$\left(\frac{\partial u}{\partial y}\right)_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{\Delta y} + O(\Delta y)$$

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + O(\Delta x^2)$$

$$\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} + O(\Delta y^2)$$

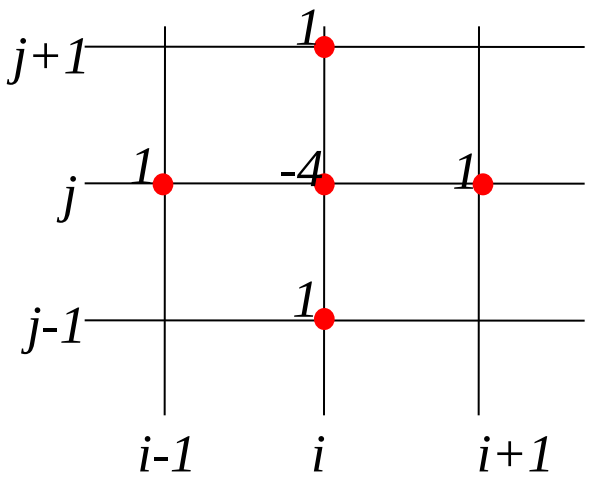


Difference formulas for the Laplace operator

- 2nd order central difference formula for the Laplace operator:

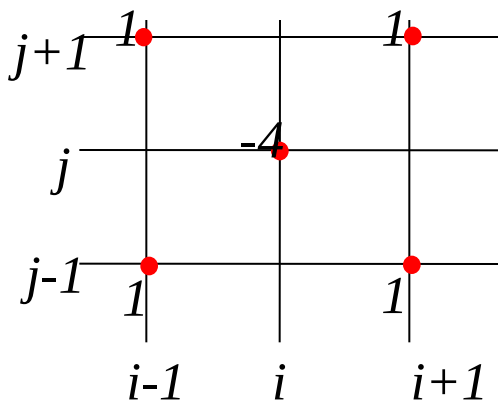
$$(\nabla^2 u)_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} + O(\Delta x^2, \Delta y^2)$$

- This is the most widely applied 2nd order scheme for the Laplace operator, and can be represented by the following *computational molecule:*



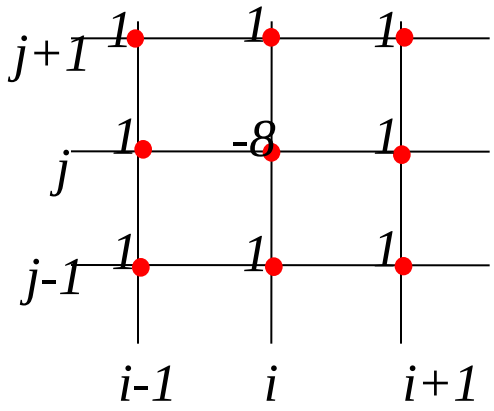
Other forms of the Laplace operator

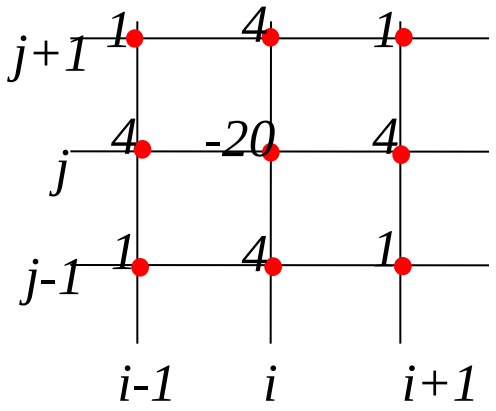
- 2nd order 5-point formula:



This formula is not recommended because odd- and even-numbered points are detached and oscillatory solutions can be obtained

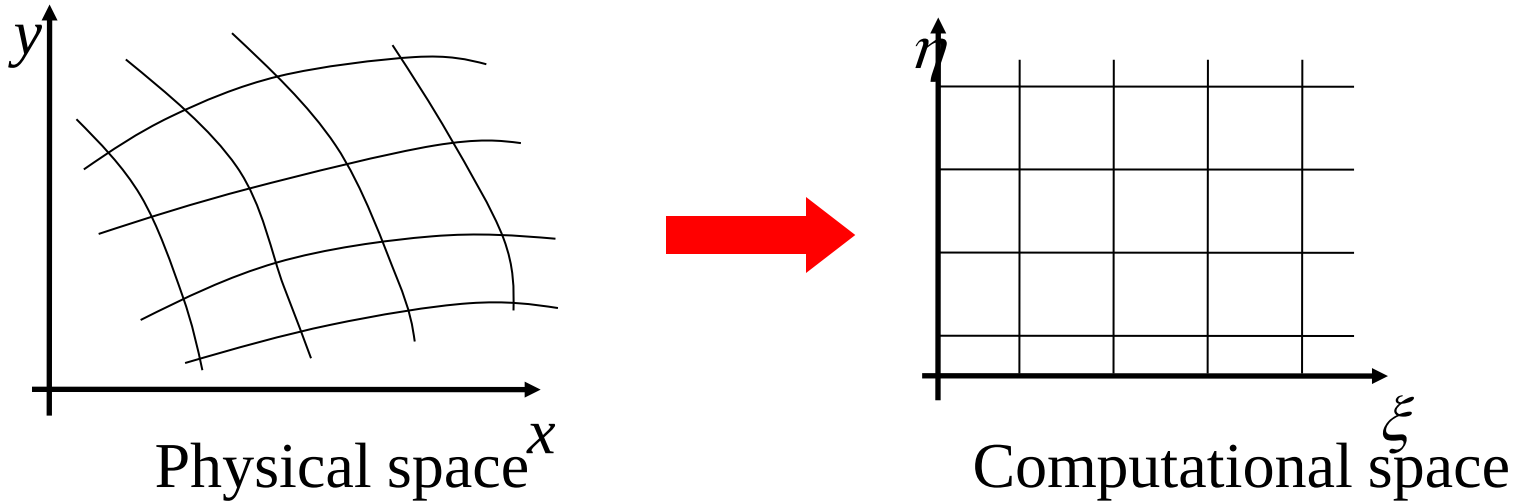
- Other 2nd order 9-point formulas:





FD formulas on non-uniform Cartesian meshes

- For non-uniform or curvilinear meshes the discretization can be performed after a transformation from the physical space (x, y, z) to a Cartesian computational space (ξ, η, ζ)
- The relations between the two spaces are defined by a coordinate transformation or mapping:
  $\xi = \xi(x,y,z)$, $\eta = \eta(x,y,z)$, $\zeta = \zeta(x,y,z)$
- All the FD formulas previously derived can be applied in the computational space (ξ, η, ζ)



FD formulas in 1D non-uniform space

- Consider an arbitrary mesh point distribution in a one dimensional space:

- FD formulas can be derived from Taylor expansions. For the first derivatives we have:

- Forward difference: $$(u_x)_i = \frac{u_{i+1} - u_i}{\Delta x_{i+1}} - \frac{\Delta x_{i+1}}{2} u_{xx}$$

$$\Delta x_i = x_i - x_{i-1}$$

- Backward difference: $$(u_x)_i = \frac{u_i - u_{i-1}}{\Delta x_i} + \frac{\Delta x_i}{2} u_{xx}$$

- Central difference:

$$(u_x)_i = \frac{1}{\Delta x_i + \Delta x_{i+1}} \left[\frac{\Delta x_i}{\Delta x_{i+1}} \left(u_{i+1} - u_i \right) + \frac{\Delta x_{i+1}}{\Delta x_i} \left(u_i - u_{i-1} \right) \right] - \frac{\Delta x_i \, \Delta x_{i+1}}{6} u_{xxx}$$

2nd derivatives

- The 3-point central difference formula for the second derivative is

$$(u_{xx})_i = \frac{2}{\Delta x_i + \Delta x_{i+1}}\left[\frac{u_{i+1}-u_i}{\Delta x_{i+1}} + \frac{u_i - u_{i-1}}{\Delta x_i}\right] + \frac{1}{3}\left(\Delta x_{i+1} - \Delta x_i\right)u_{xxx} + O(\Delta x^2)$$

- Note that the truncation error is proportional to the difference of two consecutive mesh lengths. If the mesh varies abruptly, this formula is only first order accurate.

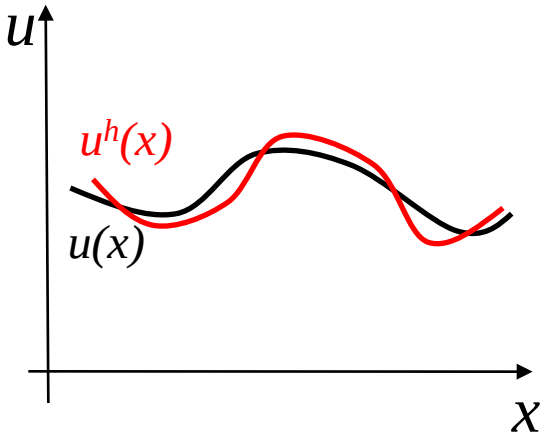
- This is a general property of finite difference approximations on non-uniform meshes. If the mesh size does not vary smoothly, a loss of accuracy is unavoidable.

FINITE ELEMENT METHOD

Approximation of functions

- Given a function *u(x)* in a domain Ω, we want to approximate this function as a linear combination of known functions:

$$u(x) \approx u^h(x) = N^i(x)\hat{u}_i$$



$u^h(x)$ *is* the approximation of *u(x)* ,
(Einstein's summation convention)

$N^i(x)$ are called trial functions, $\hat{u}_i$ are the *nodal values* of $u(x_i)$

- In general, we choose a complete set of trial functions

Examples

- Truncated Taylor series:

$$u(x) \approx u^h(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

$$a_i = \frac{1}{i!} \frac{d^i u}{dx^i}\bigg|_{x=0}$$

- Truncated Sine series:

$$u(x) \approx u^h(x) = a_i \sin \frac{i \pi x}{L}$$

$$a_i = \frac{2}{L} \int_0^L u(x) \sin \frac{i \pi x}{L} dx$$

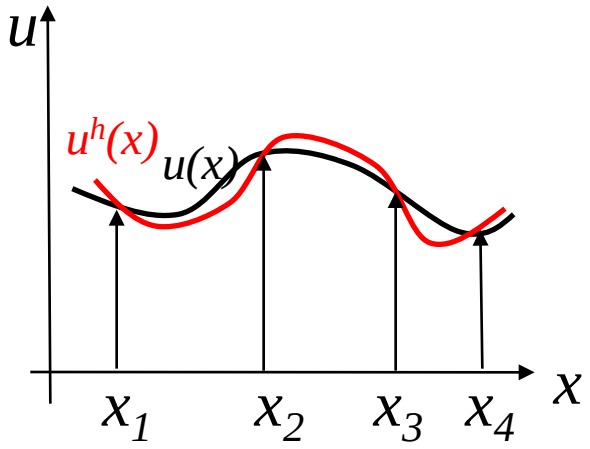
Determination of constants

Point Fitting Methods:

- Set $u^h = u$ at M selected points:

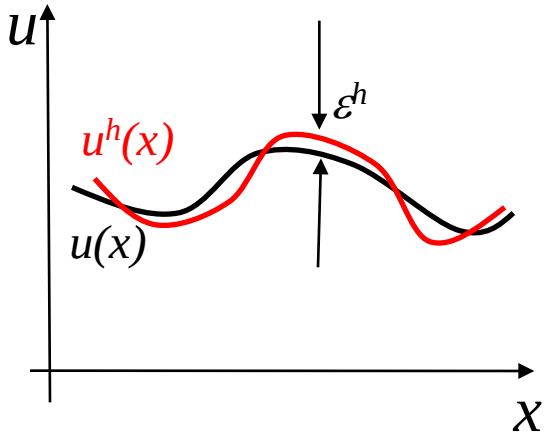
$$u^h(x_i) = u(x_i) \quad i = 1,2,\ldots,m$$

$$\Rightarrow \quad \mathbf{K} \cdot \mathbf{a} = \mathbf{f}$$



Weighted Residual Methods:

- Find the constants that minimize the “error” between the functions



Weighted Residual Methods

- We want $\varepsilon^h = u - u^h \to 0$ in Ω.

- This can be satisfied by requiring that

$$\int_\Omega \varepsilon^h d\Omega = 0 \quad \Rightarrow \quad \int_\Omega W \varepsilon^h d\Omega = 0 \quad \forall W$$

- WRM approximate this integral statement by considering only a finite set of weighting functions:

$$\int_\Omega W^i \varepsilon^h d\Omega = 0 \quad i = 1 \ldots m$$

- Then as $m \to \infty$, $\varepsilon^h \to 0$ at all points in Ω

WRM

- Inserting the approximation for u:

$$u^h(x) = N^i(x)\,\hat{u}_i$$

$$\int_\Omega W^i \varepsilon^h d\Omega = 0 \quad \Rightarrow \quad \int_\Omega W^i (u - N^j \hat{u}_j) d\Omega \quad i = 1...m$$

- This leads to a system of algebraic equations:

$$\mathbf{K} \cdot \hat{\mathbf{u}} = \hat{\mathbf{r}}, \quad K^{ij} = \int_\Omega W^i N^j \, d\Omega, \quad r^i = \int_\Omega W^i u \, d\Omega$$

- How we choose W^i defines the method

Point collocation method

- Choosing $W^i = \delta(x - x_i)$ for a set of points $x_i \in \Omega$, the weighted residual statement becomes:

$$\int_\Omega W^i \varepsilon^h d\Omega = \int_\Omega \delta(x - x_i)\, \varepsilon^h d\Omega = \varepsilon^h(x_i) = 0, \quad i = 1,2,\ldots,M$$

- Therefore, we obtain the following system of equations:

$$N^j(x_i)\hat{u}_j = u(x_i)$$

- This is equivalent to a Point Fitting Method

Galerkin method

- Choosing $W^i = N^i$, the weighted residual statement becomes:

$$\int_\Omega N^i \, \varepsilon^h d\Omega = \int_\Omega N^i \, (u - N^j \hat{u}_j) \, d\Omega = 0, \quad i = 1,2,\ldots,M$$

- Therefore, we obtain the following system of equations:

$$\left[\int_\Omega N^i N^j \, d\Omega \right] \hat{u}_j = \int_\Omega N^i u \, d\Omega \quad i = 1,2,\ldots,M$$

- In matrix form:

$$\mathbf{M_c} \cdot \hat{\mathbf{u}} = \mathbf{r}$$

$\mathbf{M_c}$ is called the consistent mass-matrix

Least squares problem

- In the least squares problem we want to minimize the square of the point-wise errors:

$$I = \int_{\Omega} (\varepsilon^h)^2 \, d\Omega = \int_{\Omega} (u^h - u)^2 \, d\Omega = \int_{\Omega} (N^j \hat{u}_j - u)^2 \, d\Omega \rightarrow \min$$

- The variation of this integral must be zero at the minimum:

$$\delta I = \delta \hat{u}_i \int_{\Omega} N^i \, (N^j \hat{u}_j - u) \, d\Omega = 0 \quad \Rightarrow \quad \int_{\Omega} N^i \, N^j \, d\Omega \, \hat{u}_j = \int_{\Omega} N^i u \, d\Omega$$

- This is equivalent to the Galerkin WRM

- This shows that the choice of the weighting functions equal to the trial functions is an optimal choice

Local vs global trial functions

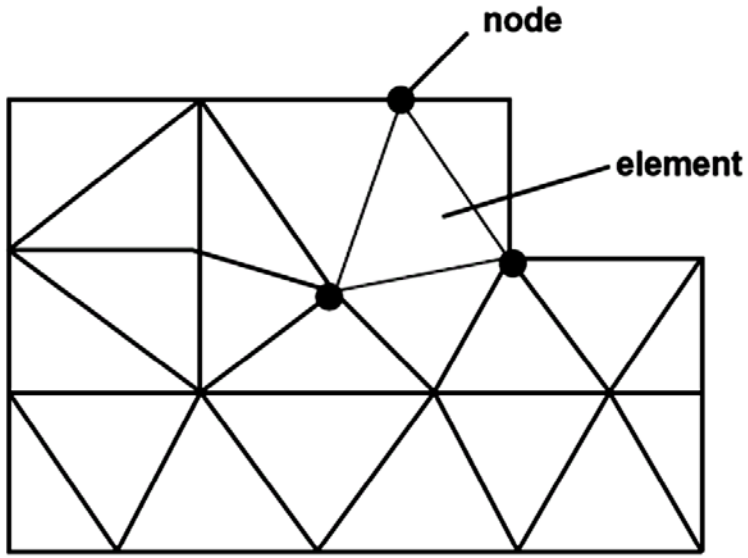
- There are several drawbacks in using *global trial functions*:
  - Determining N^j is difficult for all but a the simplest geometries in 2D or 3D
  - The matrix is full and can become ill-conditioned, even for simple problems
  - The constants $\hat{u}_i$ have no physical meaning

- Therefore, the use of *local trial functions* is usually preferred

Local trial functions

Given $u(x)$ for $x \in \Omega$:

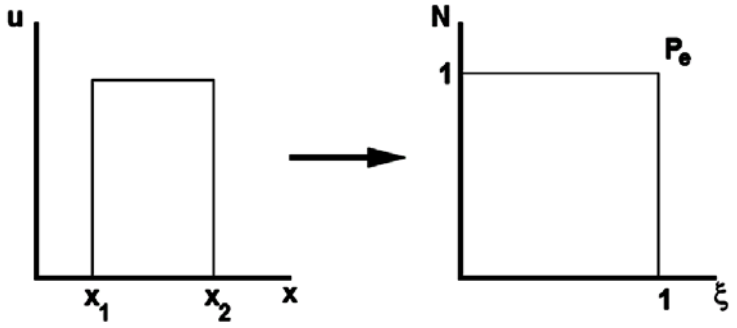
- Divide the computational domain Ω into a set of non-overlapping sub-intervals Ω_{el}
- Then define u^h in each sub-interval
- The sub-intervals are called *elements* and the points x_i are called *nodes*



Constant trial functions

- Define a piecewise constant function:

$$N^E(x) = \begin{cases} 1 & x \in E \\ 0 & x \notin E \end{cases}$$

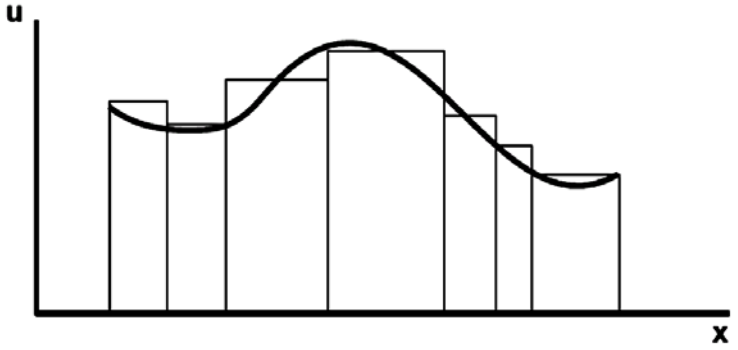


- Then, globally:

$$u(x) \approx u^h(x) = N^E(x) u_E$$

- And locally on element *E*:

$$u(x) \approx u^h(\xi) = u_E$$



Linear trial functions

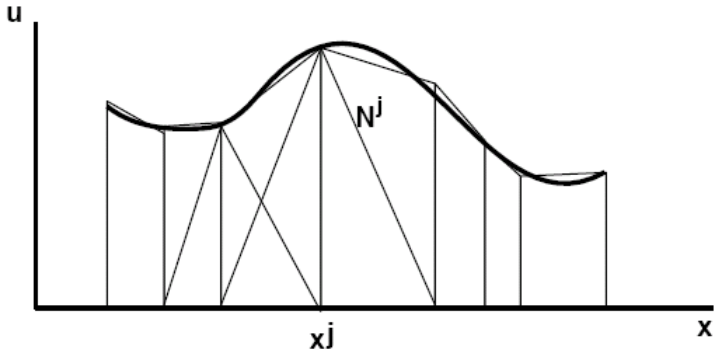
- Let u^h vary linearly within each element E
- Define a piecewise linear trial function that obeys:

$$N^j(x) = \begin{cases} 1 & \text{at node } j \\ 0 & \text{at all other nodes} \end{cases}$$

and N^j is only non-zero on the elements associated with node j

- Then, globally:

$$u(x) \approx u^h(x) = N^j(x)\hat{u}_j$$



Linear shape functions

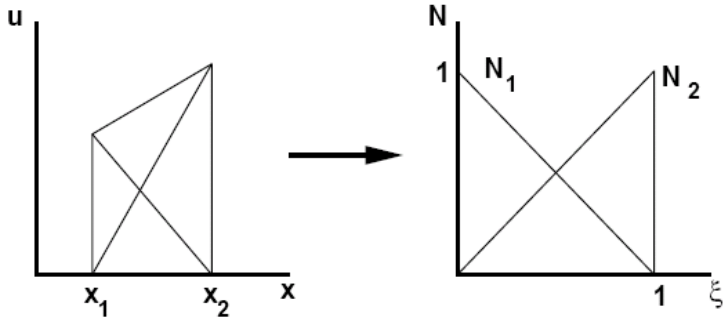
- Locally on element *E* with nodes 1 and 2:

$$u(x) \approx u^h(\xi) = N^1(\xi)\hat{u}_1 + N^2(\xi)\hat{u}_2$$

- Defining:

$$\xi = (x - x_1)/h_E$$

$$h_E = (x_2 - x_1)$$



we have:

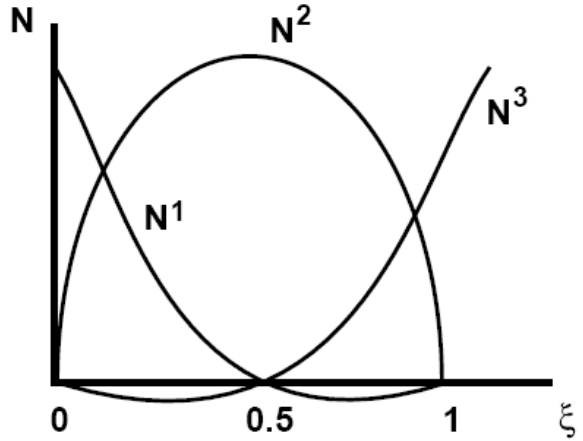
$$N_E^1(\xi) = (x_2 - x)/h_E = 1 - \xi$$

$$N_E^2(\xi) = (x - x_1)/h_E = \xi$$

$$\Rightarrow \quad u^h = \frac{(x_2 - x)}{h_E}\hat{u}_1 + \frac{(x - x_1)}{h_E}\hat{u}_2 = (1 - \xi)\hat{u}_1 + \xi\hat{u}_2$$

Quadratic trial functions

- Let u^h vary quadratically within each element E
- Place extra nodes at the middle of each element in addition to the nodes at its ends:



- The shape functions for an element E are:

$$N_E^1(\xi) = (1-\xi)(1-2\xi)$$

$$N_E^2(\xi) = 4\xi(1-\xi)$$

$$N_E^3(\xi) = -\xi(1-2\xi)$$

General properties of shape functions

- Interpolation property: $u^h = N^i(x)\hat{u}_i$

$$\Rightarrow \quad u^h(x_j) = N^i(x_j)\hat{u}_i = \hat{u}_j \Rightarrow N^i(x_j) = \delta^i_j$$

- Constant sum: must be able to represent a constant

$$u = 1 \Rightarrow u^h = 1 = N^i(x)\hat{u}_i$$

  using interpolation property:

$$\hat{u}_i = 1 \Rightarrow \sum_i N^i(x) = 1 \quad \forall x \in \Omega$$

- Conservation property: (derivate previous equation)

$$\sum_i N^i_{,k} = 0 \quad \forall x \in \Omega_E$$

Coordinate interpolation

- In general, we can select *u*=*x*, and apply the interpolating formulas using the element shape functions:

$$x = N^i(x)\, x_i$$

- This formula is valid in any number of dimensions and for any trial functions

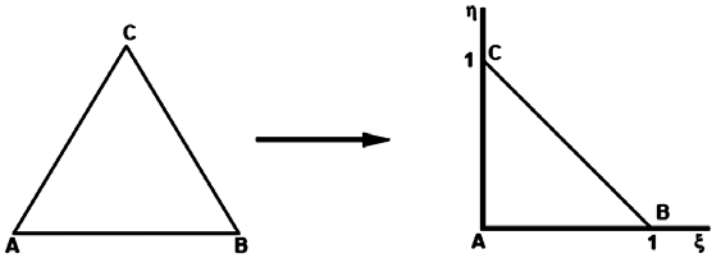
- This property is useful for deriving the shape functions of elements in 2D and 3D

Linear triangle: shape functions

- Shape functions:

$$\mathbf{x} = \mathbf{x}_A + (\mathbf{x}_B - \mathbf{x}_A)\xi + (\mathbf{x}_C - \mathbf{x}_A)\eta$$

$$\Rightarrow \quad \mathbf{x} = N^i \mathbf{x}_i = (1-\xi-\eta)\mathbf{x}_A + \xi \mathbf{x}_B + \eta \mathbf{x}_C$$

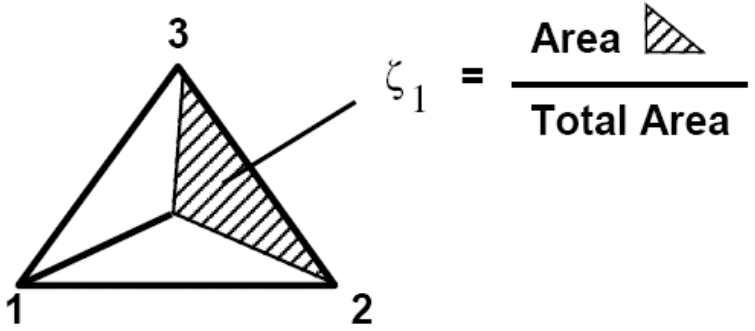


- Area coordinates:

$$N^1 = \zeta_1 = 1-\xi-\eta$$

$$N^2 = \zeta_2 = \xi$$

$$N^3 = \zeta_3 = \eta$$



Linear triangle: shape functions at a point

- The shape functions of a linear triangle can be found at the location of a point $\mathbf{x}_i$ as follows:

$$\mathbf{x}_i = (1-\xi-\eta)\,\mathbf{x}_A + \xi\,\mathbf{x}_B + \eta\,\mathbf{x}_C$$

$$\Rightarrow \quad \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} = \begin{bmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ z_a & z_b & z_c \end{bmatrix} \cdot \begin{pmatrix} \zeta \\ \xi \\ \eta \end{pmatrix}$$

$$\Rightarrow \quad \begin{pmatrix} \zeta \\ \xi \\ \eta \end{pmatrix} = \begin{bmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ z_a & z_b & z_c \end{bmatrix}^{-1} \cdot \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$$

Linear triangle: shape function derivatives

$$N^i_{,x} = N^i_{,\xi}\,\xi_{,x} + N^i_{,\eta}\,\eta_{,x}$$

$$\mathbf{J} = \begin{pmatrix} x_{,\xi} & x_{,\eta} \\ y_{,\xi} & y_{,\eta} \end{pmatrix} = \begin{pmatrix} x_{BA} & x_{CA} \\ y_{BA} & y_{CA} \end{pmatrix} \quad \Rightarrow \det(\mathbf{J}) = 2A_E = x_{BA}\,y_{CA} - x_{CA}\,y_{BA}$$

$$\mathbf{J}^{-1} = \begin{pmatrix} \xi_{,x} & \xi_{,y} \\ \eta_{,x} & \eta_{,y} \end{pmatrix} = \frac{1}{2A_E} \begin{pmatrix} y_{CA} & -x_{CA} \\ -y_{BA} & x_{BA} \end{pmatrix}$$

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \end{bmatrix}_{,x} = \frac{1}{2A_E} \begin{bmatrix} -y_{CA} + y_{BA} \\ y_{CA} \\ -y_{BA} \end{bmatrix}$$

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \end{bmatrix}_{,y} = \frac{1}{2A_E} \begin{bmatrix} x_{CA} - x_{BA} \\ -x_{CA} \\ x_{BA} \end{bmatrix}$$

Linear triangle: basic integrals

- Useful integrals:

$$\int_E N^i \, d\Omega = \frac{A_E}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\mathbf{M}_E = \int_E N^i N^j \, d\Omega = \frac{A_E}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

Quadratic triangles

- Place extra nodes at the mid-points of the triangle edges

- Shape functions:

$$N^1 = (1-\xi-\eta)(1-2\xi-2\eta)$$

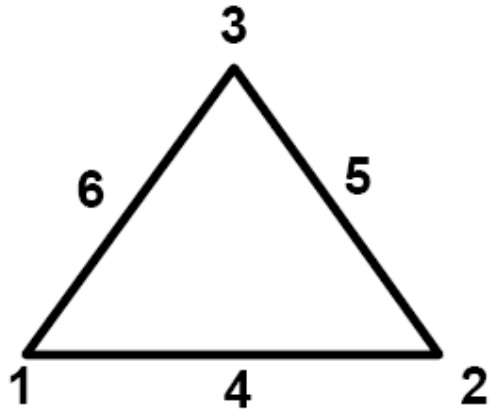
$$N^2 = \xi(2\xi-1)$$

$$N^3 = \eta(2\eta-1)$$

$$N^4 = 4\xi(1-\xi-\eta)$$

$$N^5 = 4\xi\eta$$

$$N^6 = 4\eta(1-\xi-\eta)$$



Bilinear quadrilateral elements

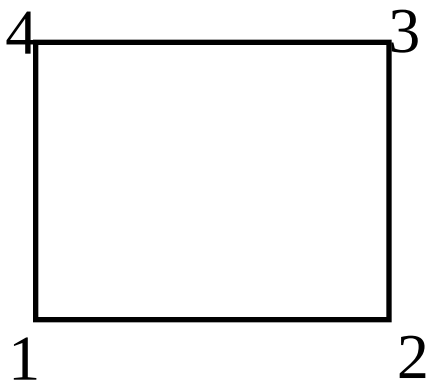
- Shape functions:

$$N^1 = (1-\xi)(1-\eta)$$

$$N^2 = \xi(1-\eta)$$

$$N^3 = \xi\eta$$

$$N^4 = (1-\xi)\eta$$



WRM with local functions

- The basic idea is to split the global integral as a sum of integrals on each element:

$$\int_{\Omega} \ldots d\Omega = \sum_{E} \int_{\Omega_E} \ldots d\Omega$$

- *Build* the integrals at the element level
- *Gather* info from global point-arrays to local element-arrays
- *Scatter-add* resulting integrals to global matrix/vector locations

$$K^{ij} u_j = \left[\sum_{E} K_E^{ij} \right] \left[u_j \right]_E = \sum_{E} r_E^i = r^i$$

- For these integrals we only need to know which nodes belong to each element (element connectivity data structure)

Finite element approximation of operators

- Function approximations:

  given *u,* approximate $| u - u^h | \rightarrow \min$

- Operator approximations:

  given *L(u)=0*, approximate $| L(u) - L(u^h) | \rightarrow \min$

  $\Rightarrow | L(u^h) | \rightarrow \min$

- Minimize error or residual:

$$\varepsilon_L^h = L(u^h) = L(N^i \hat{u}_i)$$

- WRM:

$$\int_\Omega W^i \varepsilon_L^h \, d\Omega = 0 \,, \quad i = 1, \ldots, m$$

Finite element methods

The choice of N^i and W^i defines the method:

- N^i polynomial, $W^i = \delta(x_i)$: FDM
- N^i polynomial, $W^i = 1$ *if* $x \in \Omega_E$ *, 0 otherwise*: FVM
- N^i polynomial, $W^i = N^i$: Galerkin FEM
- N^i polynomial, $W^i \neq N^i$: Petrov-Galerkin FEM
- N^i spectral (sin, cos), $W^i = \delta(x_i)$: spectral element methods

Example: gradient of a function

gradient of a scalar field : $\mathbf{v} = \nabla \phi$

weighted residual statement : $\int W \, \mathbf{v} \, d\Omega = \int W \, \nabla \phi \, d\Omega$

functional approximation : $\phi(x) = N^j(x) \phi_j \quad \mathbf{v}(x) = N^j(x) \, \mathbf{v}_j$

Galerkin method : $\int N^i \, N^j \mathbf{v}_j \, d\Omega = \int N^i \, \nabla N^j \, \phi_j \, d\Omega$

$\Rightarrow$ linear system : $M^{ij} \mathbf{v}_j = \mathbf{r}^i$

$M^{ij} = \int N^i \, N^j \, d\Omega$

$\mathbf{r}^i = \left(\int N^i \, d\Omega \right) \nabla N^j \, \phi_j$

Elementary matrices

$$\int_E N^i d\Omega = \frac{A_E}{3}\begin{bmatrix}1\\1\\1\end{bmatrix}$$

$$\mathbf{M}_c = \int_E N^i N^j d\Omega = \frac{A_E}{12}\begin{bmatrix}2 & 1 & 1\\1 & 2 & 1\\1 & 1 & 2\end{bmatrix}$$

$$\text{lumped mass matrix} : \mathbf{M}_c \rightarrow \mathbf{M}_l = \frac{A_E}{12}\begin{bmatrix}4 & 0 & 0\\0 & 4 & 0\\0 & 0 & 4\end{bmatrix}$$

Shape function gradient

$$\nabla N^i = \left(\frac{\partial N^i}{\partial x}, \frac{\partial N^i}{\partial y} \right) = \left(N^i{}_{,x}, N^i{}_{,y} \right)$$

$$2A_E = x_{BA} y_{CA} - x_{CA} y_{BA}$$

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \end{bmatrix}_{,x} = \frac{1}{2A_E} \begin{bmatrix} -y_{CA} + y_{BA} \\ y_{CA} \\ -y_{BA} \end{bmatrix}$$

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \end{bmatrix}_{,y} = \frac{1}{2A_E} \begin{bmatrix} x_{CA} - x_{BA} \\ -x_{CA} \\ x_{BA} \end{bmatrix}$$

Solving for the Consistent Mass Matrix

for the system : $\mathbf{M}_c \cdot \mathbf{u} = \mathbf{r}$

add and subtract lumped : $\left(\mathbf{M}_c - \mathbf{M}_l\right) \cdot \mathbf{u} + \mathbf{M}_l \cdot \mathbf{u} = \mathbf{r}$

move to the RHS : $\mathbf{M}_l \cdot \mathbf{u} = \mathbf{r} - \left(\mathbf{M}_c - \mathbf{M}_l\right) \cdot \mathbf{u}$

iterative procedure : $\mathbf{M}_l \cdot \mathbf{u}^{n+1} = \mathbf{r} - \left(\mathbf{M}_c - \mathbf{M}_l\right) \cdot \mathbf{u}^n$

or $\mathbf{M}_l \cdot \mathbf{u}^{n+1} = \mathbf{r} + \boldsymbol{\delta}$

convergence : $\mathbf{u}^{n+1} = \mathbf{u}^n \quad \Rightarrow \quad \mathbf{M}_c \cdot \mathbf{u}^{n+1} = \mathbf{r}$

Computing the RHS

RHS modification : $\boldsymbol{\delta} = \left(\mathbf{M}_c - \mathbf{M}_l\right) \cdot \mathbf{u}^n = \mathbf{A} \cdot \mathbf{u}^n$

index notation : $\delta^i = A^{ij}\, u_j^n$

assembling the matrix : $A^{ij} = \sum_k A_k^{ij} \quad \Rightarrow \quad \delta^i = \left(\sum_k A_k^{ij} \right) u_j^n$

element contributions to the RHS : $\delta^i = \left(\sum_k A_k^{ij}\, u_j^n \right) = \sum_k \delta_k^i$

Procedure

- Initialize lumped mass matrix and rhs vector
- For each element:
  - Gather element nodes
  - Gather node coordinates
  - Calc element jacobian (area)
  - Calc element shape function derivatives
  - Calc element contributions to rhs and lumped mass matrix
  - Add to global rhs and lumped mass matrix
- For each node:
  - Calc gradient field: inverse lumped mass matrix * rhs

Another example: curl of a vector field

curl of a vector field : $\mathbf{w} = \nabla \times \mathbf{v}$

weighted residual statement : $\int W\, \mathbf{w}\, d\Omega = \int W\, \nabla \times \mathbf{v}\, d\Omega$

functional approximation : $\mathbf{w}(x) = N^j(x)\, \mathbf{w}_j \quad \mathbf{v}(x) = N^j(x)\, \mathbf{v}_j$

Galerkin method : $\int N^i\, N^j \mathbf{w}_j\, d\Omega = \int N^i\, \nabla N^j\, \mathbf{v}_j\, d\Omega$

$\Rightarrow$ linear system : $M^{ij} \mathbf{w}_j = \mathbf{r}^i$

$M^{ij} = \int N^i\, N^j\, d\Omega$

$\mathbf{r}^i = \left(\int N^i\, d\Omega \right) \nabla N^j \times \mathbf{v}_j$

GRID INTERPOLATION

Interpolation on grids

Given:

- Field values at the nodes of a Cartesian grid
- Position in space

Find:

- Field value at the given position

Procedure:

- Find cell (element) containing given position
- Linearly interpolate field values from cell nodes

Interpolation on a Cartesian grid

grid bounding box : $[x_{\min}, x_{\max}, y_{\min}, y_{\max}]$

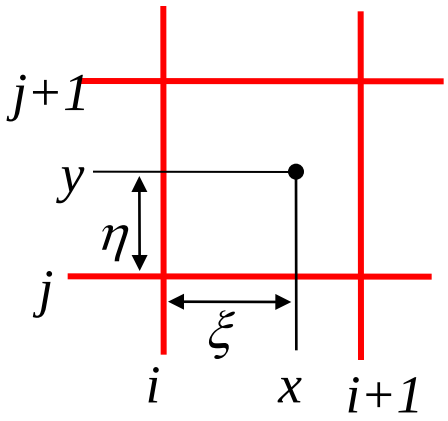
number of grid points in each direction : n_x, n_y

field values : $\phi_{ij} \quad i = 1 \ldots n_x \quad j = 1 \ldots n_y$

grid spacings : $\Delta x = \dfrac{x_{\max} - x_{\min}}{n_x}, \quad \Delta y = \dfrac{y_{\max} - y_{\min}}{n_y}$

cell that contains the given position *(x,y)*:

$$i = \left[\frac{x}{\Delta x} \right], \quad j = \left[\frac{y}{\Delta y} \right] \qquad [*] = \text{integer part}$$



local cell coordinates :

$$\xi = x - i * \Delta x, \quad \eta = y - j * \Delta y$$

bilinear field interpolation :

$$\phi(x, y) = (1-\xi)(1-\eta)\phi_{ij} + \xi(1-\eta)\phi_{i+1\,j} + (1-\xi)\eta\phi_{ij+1} + \xi\eta\phi_{i+1\,j+1}$$

Interpolation on unstructured grids

- “Host” element containing given position cannot be found directly as in Cartesian grids

- Need to search the mesh for the element containing the given position

Element shape functions at a point

coordinate interpolation : $\mathbf{x} = N^a \mathbf{x}_a + N^b \mathbf{x}_b + N^c \mathbf{x}_c$

constant sum property : $N^a = 1 - N^b - N^c$

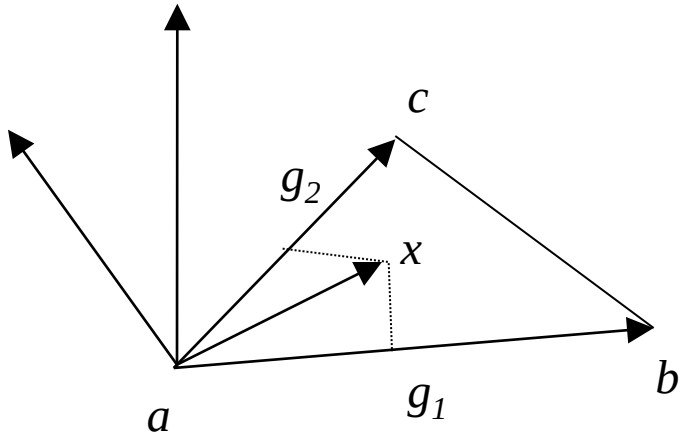
$\Rightarrow \quad \mathbf{x} = N^b (\mathbf{x}_b - \mathbf{x}_a) + N^c (\mathbf{x}_c - \mathbf{x}_a) + \mathbf{x}_a$

defining :

$\mathbf{g}_1 = \mathbf{x}_b - \mathbf{x}_a$

$\mathbf{g}_2 = \mathbf{x}_c - \mathbf{x}_a$

$\mathbf{x}_{pa} = \mathbf{x}_p - \mathbf{x}_a$



writing $\mathbf{x}$ in the $\mathbf{g}_1$ $\mathbf{g}_2$ reference frame :

$\mathbf{x} = (\mathbf{x}_{pa} \cdot \mathbf{g}_1)\mathbf{g}_1 + (\mathbf{x}_{pa} \cdot \mathbf{g}_2)\mathbf{g}_2 + \mathbf{x}_a$

equating :

$\Rightarrow \quad (\mathbf{x}_{pa} \cdot \mathbf{g}_1)\mathbf{g}_1 + (\mathbf{x}_{pa} \cdot \mathbf{g}_2)\mathbf{g}_2 = N^b \mathbf{x}_{ba} + N^c \mathbf{x}_{ca}$

Element shape functions at a point

we had :

$$(\mathbf{x}_{pa}\cdot\mathbf{g}_1)\mathbf{g}_1+(\mathbf{x}_{pa}\cdot\mathbf{g}_2)\mathbf{g}_2=N^b\mathbf{x}_{ba}+N^c\mathbf{x}_{ca}$$

writing $\mathbf{x}_{ba}$ and $\mathbf{x}_{ca}$ in the $\mathbf{g}_1\ \mathbf{g}_2$ system :

$$\mathbf{x}_{ba}=(\mathbf{x}_{ba}\cdot\mathbf{g}_1)\mathbf{g}_1+(\mathbf{x}_{ba}\cdot\mathbf{g}_2)\mathbf{g}_2$$

$$\mathbf{x}_{ca}=(\mathbf{x}_{ca}\cdot\mathbf{g}_1)\mathbf{g}_1+(\mathbf{x}_{ca}\cdot\mathbf{g}_2)\mathbf{g}_2$$

substituting :

$$(\mathbf{x}_{pa}\cdot\mathbf{g}_1)\mathbf{g}_1+(\mathbf{x}_{pa}\cdot\mathbf{g}_2)\mathbf{g}_2=N^b(\mathbf{x}_{ba}\cdot\mathbf{g}_1)\mathbf{g}_1+N^b(\mathbf{x}_{ba}\cdot\mathbf{g}_2)\mathbf{g}_2$$
$$+N^c(\mathbf{x}_{ca}\cdot\mathbf{g}_1)\mathbf{g}_1+N^c(\mathbf{x}_{ca}\cdot\mathbf{g}_2)\mathbf{g}_2$$

in matrix form :

$$\begin{pmatrix}(\mathbf{x}_{ba}\cdot\mathbf{g}_1) & (\mathbf{x}_{ca}\cdot\mathbf{g}_1)\\ (\mathbf{x}_{ba}\cdot\mathbf{g}_2) & (\mathbf{x}_{ca}\cdot\mathbf{g}_2)\end{pmatrix}\begin{pmatrix}N^b\\ N^c\end{pmatrix}=\begin{pmatrix}(\mathbf{x}_{pa}\cdot\mathbf{g}_1)\\ (\mathbf{x}_{pa}\cdot\mathbf{g}_2)\end{pmatrix}\quad\text{or}\quad\begin{pmatrix}g_{11} & g_{21}\\ g_{12} & g_{22}\end{pmatrix}\begin{pmatrix}N^b\\ N^c\end{pmatrix}=\begin{pmatrix}(\mathbf{x}_{pa}\cdot\mathbf{g}_1)\\ (\mathbf{x}_{pa}\cdot\mathbf{g}_2)\end{pmatrix}$$

Element shape functions at a point

system of equations:

$$\begin{pmatrix} g_{11} & g_{21} \\ g_{12} & g_{22} \end{pmatrix} \begin{pmatrix} N^b \\ N^c \end{pmatrix} = \begin{pmatrix} (\mathbf{x}_{pa} \cdot \mathbf{g}_1) \\ (\mathbf{x}_{pa} \cdot \mathbf{g}_2) \end{pmatrix}$$

where

$$g_{11} = g_{1x} * g_{1x} + g_{1y} * g_{1y}$$

$$g_{12} = g_{1x} * g_{2x} + g_{1y} * g_{2y}$$

$$g_{22} = g_{2x} * g_{2x} + g_{2y} * g_{2y}$$

solving the system:

$$\det = g_{11} * g_{22} - g_{12} * g_{12}$$

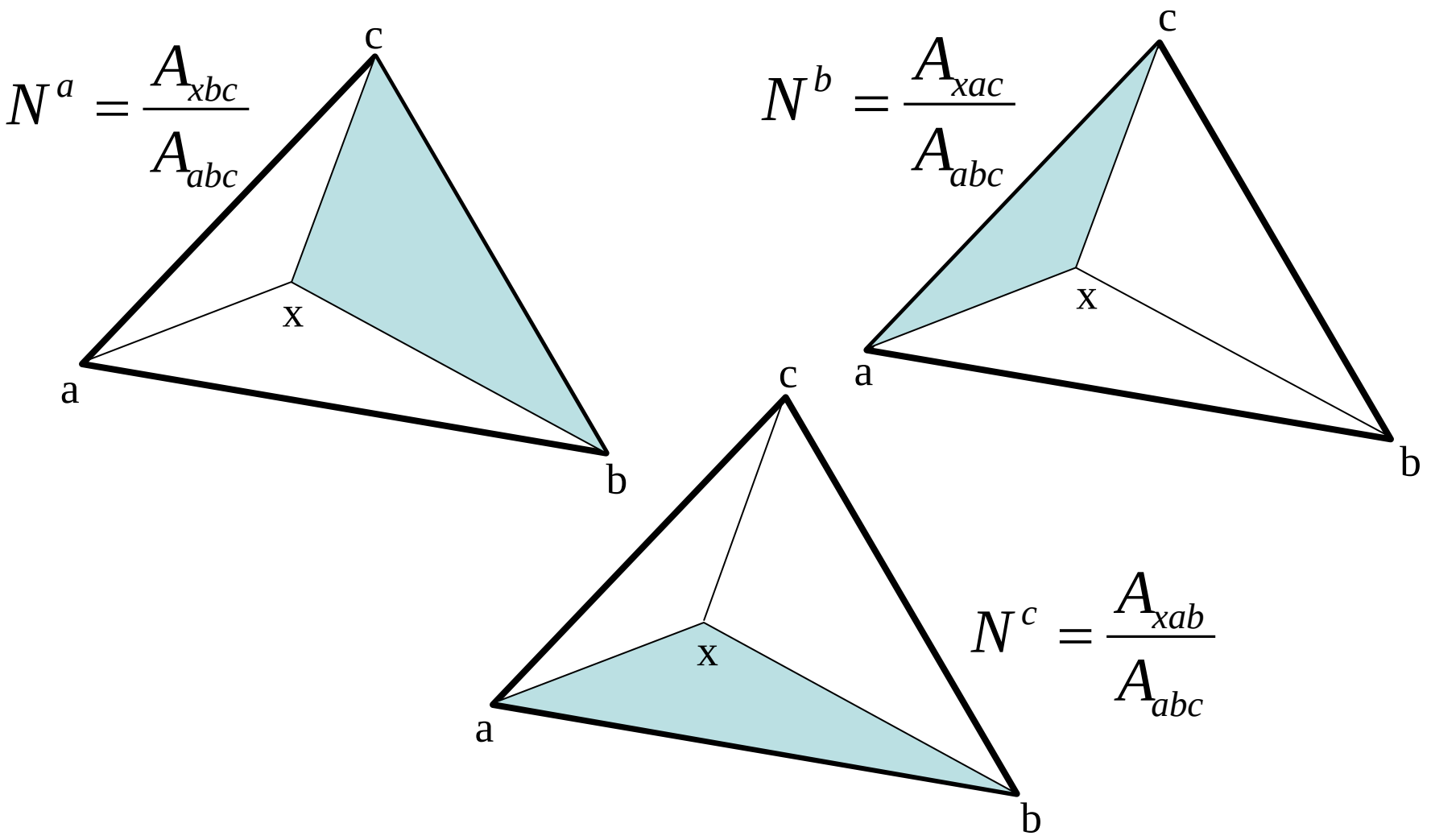
$$\eta = N^b = \left[g_{22} * (\mathbf{x}_{pa} \cdot \mathbf{g}_1) - g_{12} * (\mathbf{x}_{pa} \cdot \mathbf{g}_2) \right] / \det$$

$$\zeta = N^c = \left[g_{11} * (\mathbf{x}_{pa} \cdot \mathbf{g}_2) - g_{12} * (\mathbf{x}_{pa} \cdot \mathbf{g}_1) \right] / \det$$

$$\xi = N^a = 1 - \eta - \zeta$$

Element Shape Functions at a Point

- Linear triangle => area coordinates



Host element

conditions for inclusion in element :

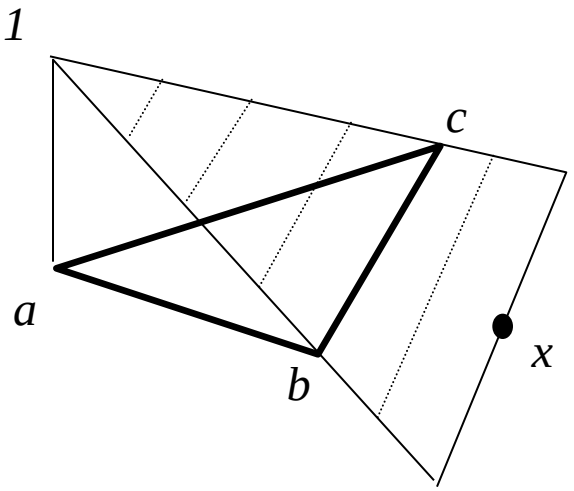
$$N^i, 1 - N^i \geq 0 \quad i = a, b, c$$

explicitly :

$$\xi > 0 \quad and \quad 1 - \xi > 0$$

$$\eta > 0 \quad and \quad 1 - \eta > 0$$

$$\zeta > 0 \quad and \quad 1 - \zeta > 0$$



Interpolating the field values

interpolation from element nodes

$$\phi(\mathbf{x}) = N^i(\mathbf{x})\,\phi_i$$

linear interpolation

$$\phi(\mathbf{x}) = \xi\,\phi_a + \eta\,\phi_b + \zeta\,\phi_c$$

Brute force interpolation

- Check whether point inside mesh bounding box
- Loop over all the elements
  - Evaluate element shape functions at given point
  - If point inside => found host
- End loop

- To interpolate between two grids the algorithm is $Np_{mesh1} \; x \; Np_{mesh2}$

Proximity in space

Problem: for a given set of points, consider:

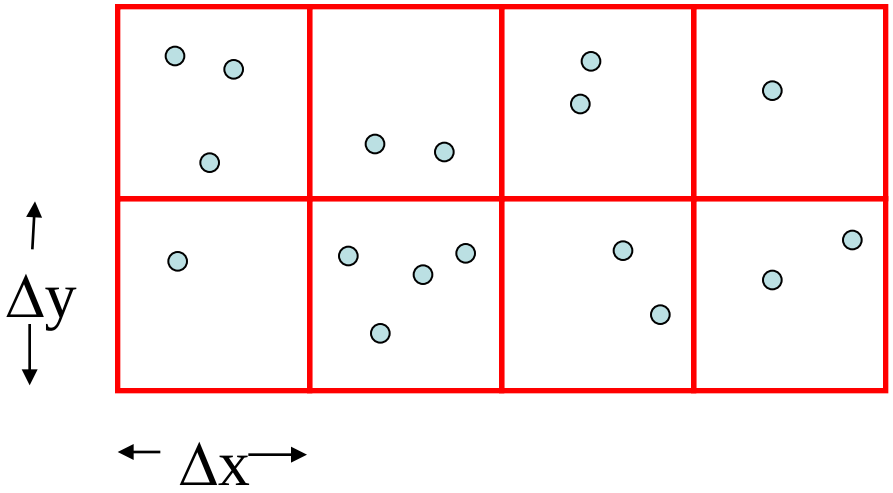
- Find the points closest to a given point
- Find all the points in a given region

Data structures:

- Bins
- Quad and Octrees

Bins Data Structure

- Subdivide space into boxes or bins
- Construct a list of points for each bin



- n_x=nr Bins in x
- $\Delta x=(x_{max}-x_{min})/n_x$
- $ix=(x_i-x_{min})/\Delta x$

n_y=nr bins in y

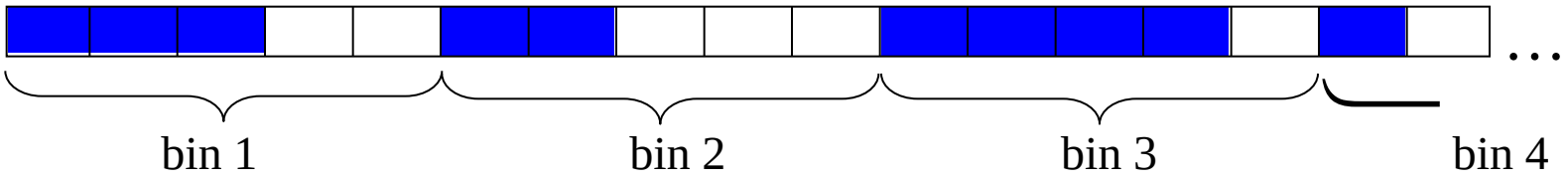
$\Delta y=(y_{max}-y_{min})/n_y$

$iy=(y_i-y_{min})/\Delta y$

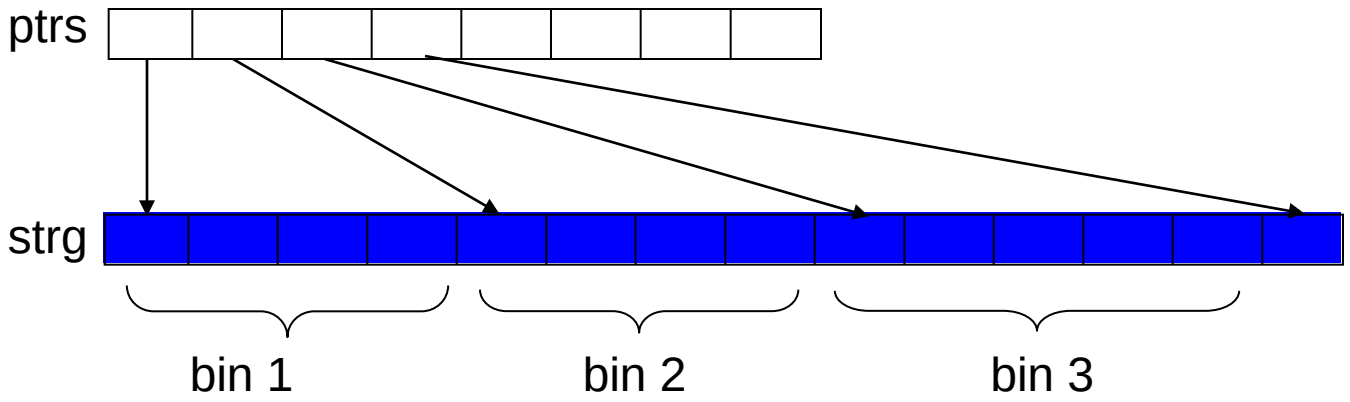
Add z for 3D

Bins: implementations

- Straight storage:



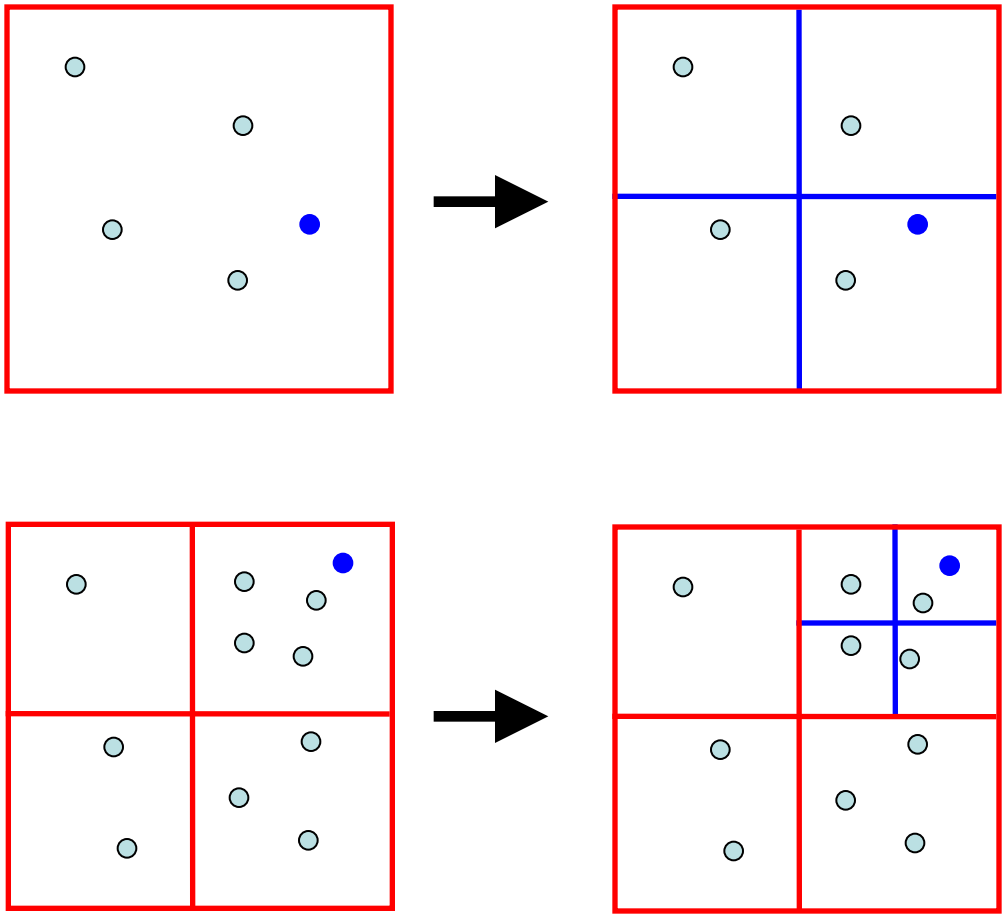
- Linked lists:



Good for data that is more or less evenly distributed in space

Quad-trees

- Recursively subdivide space into quads
- Store at most 4 points per quad



Quad-trees

For each quadrant need:

- Bounding box coordinates: x_{min}, x_{max}, y_{min}, y_{max}
- Storage space for 4 (n) points: x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4
- Pointers to the 4 children quads: c_1, c_2, c_3, c_4

Insertion / Searching

- When a quad is full and a new point inserted: create 4 child quads, store pointers & bounding boxes, and transfer points
- The quad-trees are traversed downwards
- For uniformly distributed data, it takes approximately $\log_4 N$ operations to locate all points in a search region

Octrees

- Extension of quad-trees to 3D
- Subdivide space into boxes or octants
- Store 8 (n) points per octant
- Store 8 children per octant
- Store bounding box for each octant: add z_{min}, z_{max}
- Searching: $\log_8 N$ operations for uniform distributions

Interpolation with Bins

- Load grid points into a bin data structure
- Find bin that contains the point
  - Similar to Cartesian grid interpolation
- Loop over the elements of this bin
  - Evaluate element shape functions at given point
  - If point inside => found host
- End loop

- Performance of grid to grid interpolation is $Np_1 \; x \; <Np_{1bin}>$

Octree search

- Load grid points into an octree/quadtree data structure
- Find point closest to given point
- Loop over the elements surrounding given point
  - Evaluate element shape functions at given point
  - If point inside => found host
- End loop

- Performance of grid to grid interpolation is $Np_1 \; x \; log(Np_2)*<Np_{2\_surr\_elem}>$

Neighbor to neighbor search

- Start at initial guess element
- While (not found)
  - Evaluate shape functions at given point
  - If inside => found
  - else jump to neighbor element opposite to node with largest negative shape function
- End while

- Performance depends on initial guess (how many elements away is the host element)

Layered approach

- Find initial guess from bin or octree data structures
- Perform neighbor to neighbor search
- If not found in a max number of steps, fall back to brute force algorithm

- Initialize guess for next point in grid from the current host (if grid points are close in space the host will be found in a few steps)

ELLIPTIC EQUATIONS

Finite element method

- The finite element method originated from the field of structural analysis.

- The concept of “elements” originated in stress calculations where a structure was subdivided into small substructures of various shapes and re-assembled after each element had been analyzed

- The mathematics of the finite element method has been quite extensively studied and is based on functional analysis and approximation theory

Laplace's equation

Laplace's equation : $\nabla^2 \phi = 0 \quad \text{in} \quad \Omega$

weighted residual statement : $\int W \nabla^2 \phi \, d\Omega = 0$

integration by parts : $\int W \nabla \phi \cdot n \, d\Gamma - \int \nabla W \cdot \nabla \phi \, d\Omega = 0$

natural boundary conditions : $\nabla \phi \cdot n = 0 \;\; \text{on} \;\; \Gamma$

$\Rightarrow$ weak form : $\int \nabla W \cdot \nabla \phi \, d\Omega = 0$

Finite element approximation

finite element approximation : $\phi(x) = N^j(x)\phi_j$

Galerkin method : $W = N^i \quad i = 1...n$

$$\int \nabla W \cdot \nabla \phi \, d\Omega = 0 \quad \Rightarrow \quad \int \nabla N^i \cdot \nabla N^j \phi_j \, d\Omega = 0$$

matrix form : $K^{ij} \, \phi_j = 0 \quad \leftarrow$ global system $n \times n$

Matrix assembling with linear triangles

$$K^{ij} = \int \nabla N^i \cdot \nabla N^j \, d\Omega = \left(\int d\Omega \right) \left(\nabla N^i \cdot \nabla N^j \right)$$

$$K^{aa} = A\left(\nabla N^a \cdot \nabla N^a \right) = A\left(N^a_{,x} \, N^a_{,x} + N^a_{,y} \, N^a_{,y} \right)$$

$$K^{bb} = \mathrm{A}\left(\nabla N^b \cdot \nabla N^b \right) = A\left(N^b_{,x} \, N^b_{,x} + N^b_{,y} \, N^b_{,y} \right)$$

$$K^{cc} = \mathrm{A}\left(\nabla N^c \cdot \nabla N^c \right) = A\left(N^c_{,x} \, N^c_{,x} + N^c_{,y} \, N^c_{,y} \right)$$

$$K^{ab} = \mathrm{A}\left(\nabla N^a \cdot \nabla N^b \right) = A\left(N^a_{,x} \, N^b_{,x} + N^a_{,y} \, N^b_{,y} \right)$$

$$K^{ac} = \mathrm{A}\left(\nabla N^a \cdot \nabla N^c \right) = A\left(N^a_{,x} \, N^c_{,x} + N^a_{,y} \, N^c_{,y} \right)$$

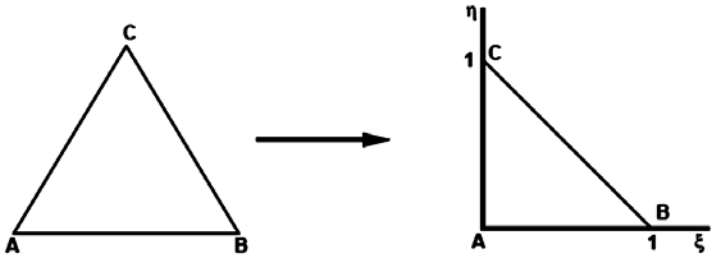
$$K^{bc} = \mathrm{A}\left(\nabla N^b \cdot \nabla N^c \right) = A\left(N^b_{,x} \, N^c_{,x} + N^b_{,y} \, N^c_{,y} \right)$$

Linear triangle: shape functions

- Shape functions:

$$\mathbf{x} = \mathbf{x}_A + (\mathbf{x}_B - \mathbf{x}_A)\xi + (\mathbf{x}_C - \mathbf{x}_A)\eta$$

$$\Rightarrow \quad \mathbf{x} = N^i \mathbf{x}_i = (1-\xi-\eta)\mathbf{x}_A + \xi \mathbf{x}_B + \eta \mathbf{x}_C$$

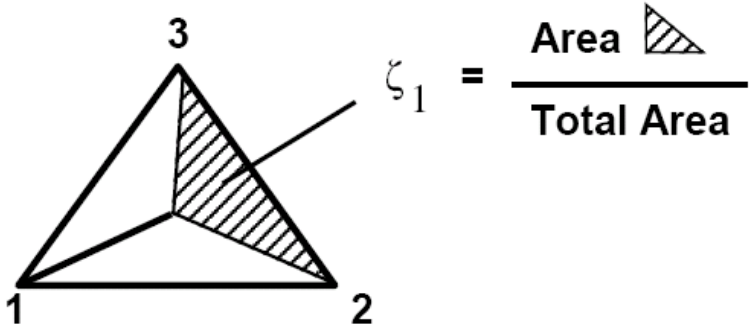


- Area coordinates:

$$N^1 = \zeta_1 = 1-\xi-\eta$$

$$N^2 = \zeta_2 = \xi$$

$$N^3 = \zeta_3 = \eta$$



Linear triangle: shape functions at a point

- The shape functions of a linear triangle can be found at the location of a point $\mathbf{x}_i$ as follows:

$$\mathbf{x}_i = (1-\xi-\eta)\,\mathbf{x}_A + \xi\,\mathbf{x}_B + \eta\,\mathbf{x}_C$$

$$\Rightarrow \quad \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} = \begin{bmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ z_a & z_b & z_c \end{bmatrix} \cdot \begin{pmatrix} \zeta \\ \xi \\ \eta \end{pmatrix}$$

$$\Rightarrow \quad \begin{pmatrix} \zeta \\ \xi \\ \eta \end{pmatrix} = \begin{bmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ z_a & z_b & z_c \end{bmatrix}^{-1} \cdot \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$$

Linear triangle: shape function derivatives

$$N^i_{,x} = N^i_{,\xi}\,\xi_{,x} + N^i_{,\eta}\,\eta_{,x}$$

$$\mathbf{J} = \begin{pmatrix} x_{,\xi} & x_{,\eta} \\ y_{,\xi} & y_{,\eta} \end{pmatrix} = \begin{pmatrix} x_{BA} & x_{CA} \\ y_{BA} & y_{CA} \end{pmatrix} \quad \Rightarrow \det(\mathbf{J}) = 2A_E = x_{BA}\,y_{CA} - x_{CA}\,y_{BA}$$

$$\mathbf{J}^{-1} = \begin{pmatrix} \xi_{,x} & \xi_{,y} \\ \eta_{,x} & \eta_{,y} \end{pmatrix} = \frac{1}{2A_E} \begin{pmatrix} y_{CA} & -x_{CA} \\ -y_{BA} & x_{BA} \end{pmatrix}$$

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \end{bmatrix}_{,x} = \frac{1}{2A_E} \begin{bmatrix} -y_{CA} + y_{BA} \\ y_{CA} \\ -y_{BA} \end{bmatrix}$$

$$\begin{bmatrix} N^1 \\ N^2 \\ N^3 \end{bmatrix}_{,y} = \frac{1}{2A_E} \begin{bmatrix} x_{CA} - x_{BA} \\ -x_{CA} \\ x_{BA} \end{bmatrix}$$

Boundary conditions

Natural boundary conditions : $\nabla \phi \cdot n = 0$

already accounted for in the formulation

Dirichlet boundary conditions : $\phi = \phi_0$

Option 1 :

modify system of equations : 1 in diagonal and ϕ_0 in the RHS

Option 2 :

delete equation from system and set $\phi = \phi_0$ for boundary points

Remark: Shape functions continuity

- Given: $\nabla^2 u = 0 \quad in\ \Omega\,, \quad u = 0 \quad on\ \Gamma(\Omega)$
- WRM: $\int_\Omega W^i \nabla^2 N^j \, d\Omega \hat{u}_j = 0$
  - N^j must have defined 2nd order derivatives
    $\Rightarrow$ must be at least C^1 continuous across elements
  - W^i can be the δ function
- Integrating by parts: $-\int_\Omega \nabla W^i \cdot \nabla N^j \, d\Omega \hat{u}_j = 0$
  - Order of max derivative reduced $\Rightarrow$ wider space of trial functions
  - N^j must have defined 1st order derivatives
    $\Rightarrow$ must be at least C^0 continuous across elements
  - W^i cannot be the δ function

Procedure

- Initialize global matrix
- Assemble global matrix:
  - loop over elements
    - Gather nodes & coordinates
    - Calc shape function derivatives
    - Calc elementary matrix
    - Scatter add to nodes
- Set boundary conditions:
  - Loop over boundary points
    - Set row to diagonal
    - Set rhs value to boundary value
- Solve linear system of equations

Example: Poisson's equation

Poisson's equation : $-\nabla^2 u = f$

WRM : $\int_\Omega \nabla W^i \cdot \nabla N^j \hat{u}_j \, d\Omega = \int W^i N^j f_j \, d\Omega$

Galerkin method : $\int_\Omega \nabla N^i \cdot \nabla N^j \, d\Omega \, \hat{u}_j = \int N^i N^j f_j \, d\Omega$

matrix system : $K^{ij} u_j = M^{ij} f_j$

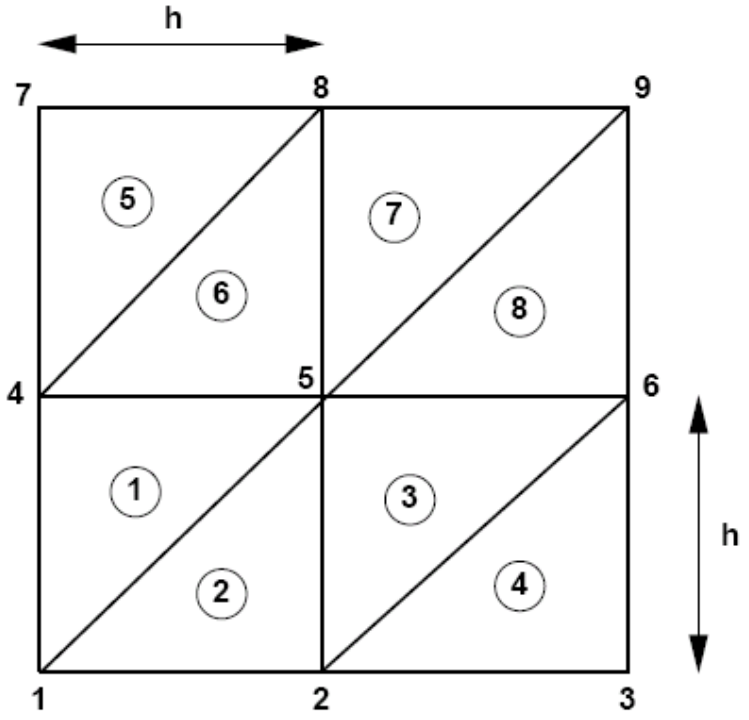
Mesh

- Regular triangular mesh – assemble element contributions to produce the equation for a typical interior point (point 5) for the Poisson operator:

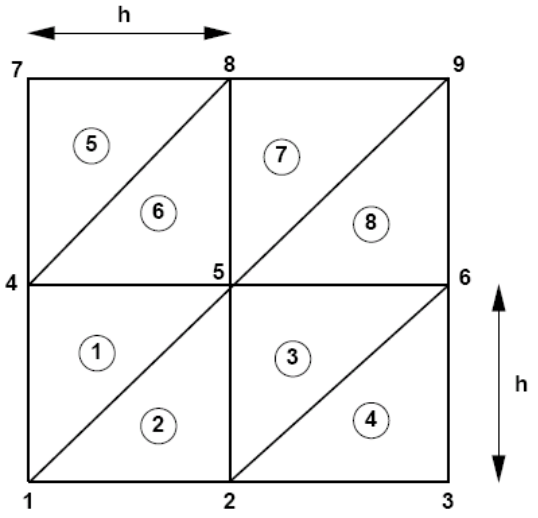
$$-\nabla^2 u = f$$

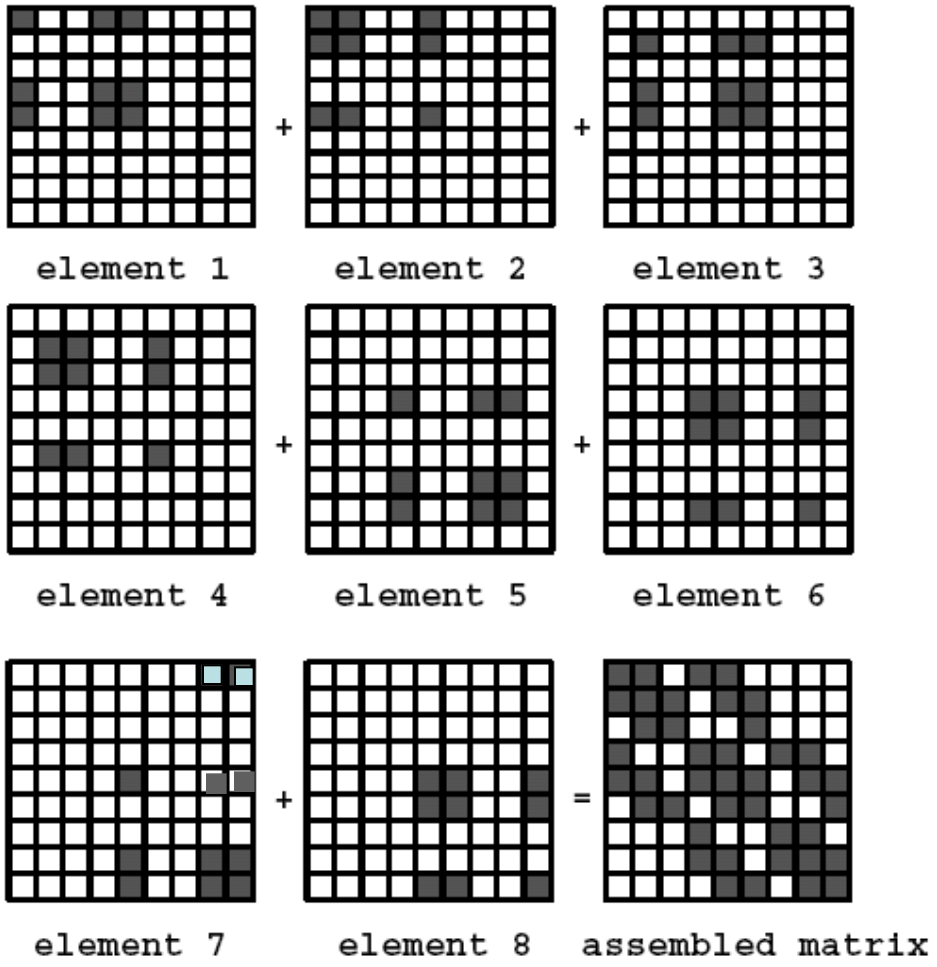
- Connectivity:

Element	Node1	Node 2	Node 3
1	1	5	4
2	2	5	1
3	2	6	5
4	2	3	6
5	4	8	7
6	4	5	8
7	5	9	8
8	5	6	9



Element contributions to global matrix





Contributions of element 1

- Shape function derivatives:

$$\begin{bmatrix} N^A \\ N^B \\ N^C \end{bmatrix}_{,x} = \frac{1}{h^2}\begin{bmatrix} 0 \\ h \\ -h \end{bmatrix}, \quad \begin{bmatrix} N^A \\ N^B \\ N^C \end{bmatrix}_{,y} = \frac{1}{h^2}\begin{bmatrix} -h \\ 0 \\ h \end{bmatrix}$$

- LHS contribution

$$\mathbf{K}_1 \cdot \mathbf{u}_1 = \frac{1}{2h^2}\begin{bmatrix} h^2 & 0 & -h^2 \\ 0 & h^2 & -h^2 \\ -h^2 & -h^2 & 2h^2 \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_1 \\ \hat{u}_5 \\ \hat{u}_4 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ -1 & -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_1 \\ \hat{u}_5 \\ \hat{u}_4 \end{bmatrix}$$

- RHS contribution

$$\mathbf{M}_1 \cdot \mathbf{f}_1 = \frac{h^2}{24}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} \hat{f}_1 \\ \hat{f}_5 \\ \hat{f}_4 \end{bmatrix} = \frac{h^2}{24}\begin{bmatrix} 2\hat{f}_1 + \hat{f}_5 + \hat{f}_4 \\ \hat{f}_1 + 2\hat{f}_5 + \hat{f}_4 \\ \hat{f}_1 + \hat{f}_5 + 2\hat{f}_4 \end{bmatrix}$$

Contributions of element 2

- Shape function derivatives:

$$\begin{bmatrix} N^A \\ N^B \\ N^C \end{bmatrix}_{,x} = \frac{1}{h^2}\begin{bmatrix} -h \\ h \\ 0 \end{bmatrix}, \quad \begin{bmatrix} N^A \\ N^B \\ N^C \end{bmatrix}_{,y} = \frac{1}{h^2}\begin{bmatrix} 0 \\ -h \\ h \end{bmatrix}$$

- LHS contribution

$$\mathbf{K}_2 \cdot \mathbf{u}_2 = \frac{1}{2h^2}\begin{bmatrix} h^2 & -h^2 & 0 \\ -h^2 & 2h^2 & -h^2 \\ 0 & -h^2 & h^2 \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_5 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_5 \end{bmatrix}$$

- RHS contribution

$$\mathbf{M}_1 \cdot \mathbf{f}_1 = \frac{h^2}{24}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} \hat{f}_1 \\ \hat{f}_2 \\ \hat{f}_5 \end{bmatrix} = \frac{h^2}{24}\begin{bmatrix} 2\hat{f}_1 + \hat{f}_2 + \hat{f}_5 \\ \hat{f}_1 + 2\hat{f}_2 + \hat{f}_5 \\ \hat{f}_1 + \hat{f}_2 + 2\hat{f}_5 \end{bmatrix}$$

Final equation

- Fully assembled equation for point 5:

$$4\hat{u}_5 - \hat{u}_2 - \hat{u}_4 - \hat{u}_6 - \hat{u}_8 = \frac{h^2}{12}\left(6\,\hat{f}_5 + \hat{f}_1 + \hat{f}_2 + \hat{f}_6 + \hat{f}_4 + \hat{f}_8 + \hat{f}_9\right)$$

- Compare this with the finite difference expansion for

$$\left[-\nabla^2 u - f\right]_{node5} = 0$$

$$4\hat{u}_5 - \hat{u}_2 - \hat{u}_4 - \hat{u}_6 - \hat{u}_8 = h^2\,\hat{f}_5$$

Elliptic PDE solution

- Discretize equations in space
  - Finite elements
  - Finite differences
- Assemble global matrix system
  - Full matrix
  - Compact storage schemes
- Solve linear system of equations
  - Direct solvers
  - Iterative solvers

SOLVING SYSTEMS OF ALGEBRAIC EQUATIONS

Solving linear systems

- Large systems of algebraic equations must be solved :

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

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

1. For obtaining the solution of elliptic PDE's (in particular, obtaining the solution of the field equations for particle systems using the PM method)
2. For advancing the solution of PDE's discretized using implicit schemes

- Typically these systems have to be solved more than once during a numerical simulation, therefore, efficient algorithms are needed

Classification of methods

Approach

- Direct solvers
- Iterative solvers

Algorithm

- Mesh relaxation methods
- Matrix methods
- Rapid elliptic solvers

Storage

- Matrix methods
- Matrix-free methods

Mesh relaxation methods

- Jacobi (J)
- Gauss-Seidel (GS)
- Successive Over Relaxation (SOR)
- Cyclic Chebyshev (CC)
- Block Iterative Methods
- Alternating-Direction Implicit (ADI)

Mesh relaxation methods

- Basic idea:

$$\text{System of equations:} \quad A \cdot u = r$$

$$\text{Split the matrix:} \quad A = B + R$$

$$B : \text{matrix that is easily invertible}$$

$$R = (A - B) : \text{remainder}$$

$$\Rightarrow \quad A \cdot u = (B + R) \cdot u = r$$

$$\Rightarrow \quad B \cdot u = -R \cdot u + r = -(A - B) \cdot u + r$$

Mesh relaxation iterations

Iterative procedure:

Start with an initial guess: $u^{(0)}$

Iterate: $B \cdot u^{(n+1)} = -(A - B) \cdot u^{(n)} + r \quad n = 1,2,...$

Since the matrix B was choosen to be easily invertible:

$$u^{(n+1)} = -B^{-1}(A - B) \cdot u^{(n)} + B^{-1} \cdot r$$

$$u^{(n+1)} = M \cdot u^{(n)} + B^{-1} \cdot r$$

M : iteration matrix

- Different iterative processes are defined by splitting the matrix *A* in different ways, i.e. by defining different forms for *B*

Relaxation factor

- Consider a general five point formula which represents a row of matrix *A* (for instance, from the discretization of the Poisson equation using second order finite central differences on a regular Cartesian grid):

$$a_{ij}\ \phi_{ij-1} + b_{ij}\ \phi_{ij+1} + c_{ij}\ \phi_{i-1j} + d_{ij}\ \phi_{i+1j} + e_{ij}\ \phi_{ij} = f_{ij}$$

- This formula can be converted into an explicit iterative scheme for point *ij* in terms of the surrounding points:

$$\phi_{ij} = \frac{1}{e_{ij}}\left[f_{ij} - a_{ij}\ \phi_{ij-1} - b_{ij}\ \phi_{ij+1} - c_{ij}\ \phi_{i-1j} - d_{ij}\ \phi_{i+1j}\right]$$

- Introducing a relaxation factor ω we have:

$$\phi_{ij}^{new} = \omega\phi_{ij}^{*} + (1-\omega)\phi_{ij}^{old}$$

Jacobi Method (J)

- In the Jacobi method the matrix *B* is the diagonal matrix formed from the diagonal elements of *A*
- The solution for the next iteration is obtained by solving each mesh point equation assuming that all the surrounding values are correct
- The new values do not replace the old values until all the new values have been calculated
- This implies $\omega=1$, and the iterative scheme is equivalent to averaging the values of the four adjacent nodes to find the next iteration of the Laplace's equation
- The method converges slowly and is not commonly used

Gauss-Seidel Method (GS)

- In this method, the matrix *B* is a lower triangular matrix formed from the lower triangular elements of *A*
- The inversion of *B* is obtained by forward substitution
- This method is computationally the same as the Jacobi method, except that the newly computed values for the next iterate replace old values from the last iteration as soon as they are calculated (again $\omega=1$)
- This method is somewhat faster than the Jacobi iteration
- However, it is really only suitable for scalar computation because one computation must be completed before another can be started

Successive Over-Relaxation (SOR)

- This method tries to improve the convergence rate of the Gauss-Seidel method by taking a linear combination of the old values and the values given by the GS iteration
- The relaxation factor lies in the range $1<\omega<2$ and corresponds to making an over-correction at the point in order to anticipate future corrections
- In some cases convergence can only be obtained with $\omega<1$, in which case the process is known as under-relaxation

SOR

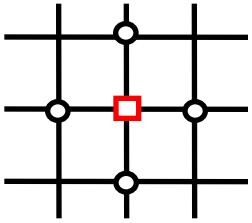
- This method converges faster than the other methods described so far
- However, convergence is still slow in the early stages of the iteration because the error can only propagate away from a cell at a rate of one cell per iteration
- Furthermore, the error during these stages can even grow from step to step

Block iterative methods

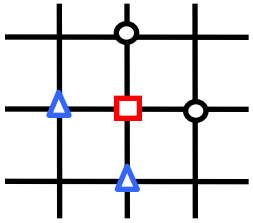
- So far we have considered methods involving pointwise adjustment of the approximate solution
- Block methods calculate entire groups of unknowns at the same time, using an implicit or semi-implicit method
- The common feature of these methods is that the convergence rate improves as larger and larger blocks of the problem are solved implicitly
- Block methods are less frequently used than point SOR principally because they are more complicated to program

Successive line overrelaxation (SLOR)

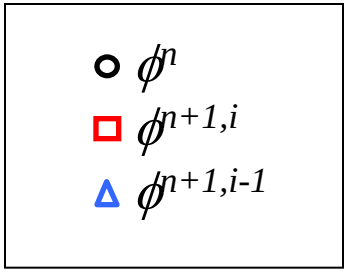
- Point iterative methods update unknowns at a single point at a time, the order of processing depends on the method used:



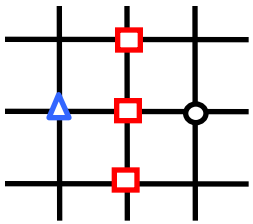
Jacobi method

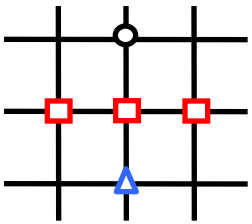


Gauss-Seidel method



- In block iterative methods a block of unknowns are updated simultaneously using an implicit method
- The SLOR for the Laplace operator updates three unknowns by solving a tridiagonal system for each row or column





Matrix methods

- Gaussian Elimination
- Thomas Tridiagonal Algorithm
- Sparse Matrix (SM)
- Conjugate Gradient Algorithm (CGA)
- Incomplete Cholesky – Conjugate Gradient (ICCG)
- Generalized Minimal Residuals (GMRES)

Upper triangular matrices

- For an upper triangular matrix

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- We can solve the system using *back-substitution:*

$$x_n = b_n / a_{nn}$$

$$x_i = \left(b_i - \sum_{j=i+1}^{n} a_{ij}\ x_j \right) / a_{ii}$$

where *i=n-1, …, 1*

Lower triangular matrices

- Similarly, for a lower triangular matrix

$$\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- we can solve the system using *forward-substitution:*

$$x_1 = b_1 / a_{11}$$

$$x_i = \left(b_i - \sum_{j=1}^{i-1} a_{ij}\ x_j \right) / a_{ii}$$

where *i=2, ... , n*

Moving to a triangular form

- Since we can always solve upper and lower triangular matrices, we want to find a way to move all matrices into these forms

- If we can write $\mathbf{A} \cdot \mathbf{x} = \mathbf{LU} \cdot \mathbf{x} = \mathbf{b}$

- We then solve the simple problem $\mathbf{L} \cdot \mathbf{y} = \mathbf{b}$ by forward-substitution

- Then we solve the problem $\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$ by backward substitution

Gaussian elimination or LU decomposition

- Doing this is conceptually fairly easy – we need to find a series of matrices M_i each of which adds a zero into the correct location.

- This is equivalent to subtracting multiples of the rows

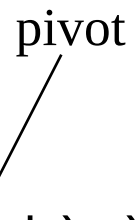
- The system we end up with is:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Gaussian elimination

The algorithm is:

```
for
   for
      for
         a(i,j)=a(i,j)-( (a(i,k)/a(k,k) ) * a(k,j)
      end
   end
end
```



- L and U are stored in the same matrix space
- This algorithm has three possible loop orders ~$O(N^3)$.
- The order of loop execution may have a large effect on execution time.
- Accessing non-contiguous memory is slower than addressing contiguous memory

Gaussian elimination

- In some cases, we may be multiplying a row by a large number
- Examine $\begin{bmatrix} 1\times10^{-6} & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 12 \\ 23 \end{bmatrix}$ $M = \begin{bmatrix} 1 & 0 \\ -1\times10^{-6} & 1 \end{bmatrix}$
- So $L = \begin{bmatrix} 1 & 0 \\ 1\times10^{-6} & 1 \end{bmatrix}$ $U = \begin{bmatrix} 1\times10^{-6} & 1 \\ 0 & 1-1\times10^{-6} \end{bmatrix}$
- The multiplier needed to zero element 21 will cause a severe numerical problem with element 22

Gaussian elimination

- We use *pivoting* to change the order of the elimination to prevent these numerical errors from happening
- Specifically, we wish to require all multipliers to have a size less than 1
- In other words, we pick the minimum for every column to use as the row multiplier to use for each stage of the elimination

Tridiagonal matrices

- For tridiagonal matrices

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{pmatrix}$$

  both forward and back substitutions reduce to a single loop:

```
for i=1,N-1
   a(i+1,N+1)-= a(i,N+1)*a(i+1,i)/a(i,i)
   a(i+1,i+1)-= a(i,i+1)*a(i+1,i)/a(i,i)
for j=N,1,-1
   x(j)=(a(j,N+1)-a(j,j+1)*x(j+1))/a(j,j)
```

- A tridiagonal system can be solved in linear time

Direct Solvers

- The work required by a *direct* solver is ~O($N_{eq}\ N_{ba}^2$)
- The storage requirement is ~O($N_{eq}\ N_{ba}$)
- For N_{eq} fixed, try to reduce the bandwidth N_{ba}
- This is achieved by *renumbering*:
  - Cuthill-McKee
  - Wavefront
  - Nested dissection
- Iterative solvers usually have less work and storage requirements and must be used for non-linear problems

Conjugate Gradient Algorithm (CGA)

- The conjugate gradient algorithm is a general method for finding the minimum of a function
- It may be used to solve a set of difference equation

$$A\,\phi = q$$

by defining the quadratic form

$$V(\phi) = \tfrac{1}{2}\,\phi^T A\,\phi - q^T\,\phi$$

where ϕ is the vector of unknowns, q is the vector of RHS values and A is an $N \times N$ matrix

- The CGA applies if the matrix A is symmetric and positive definite
- This is the case if A is the finite difference representation of $-\nabla^2$

CGA

- Differencing the quadratic equation $V(\phi) = \frac{1}{2}\,\phi^T A\,\phi - q^T\,\phi$ we obtain the gradient vector r:

$$\frac{\partial V}{\partial \phi} = A\phi - q = r$$

- And differencing again shows that *A* is the matrix of second derivatives

$$A_{ij} = \frac{\partial^2 V}{\partial \phi_i \partial \phi_j}$$

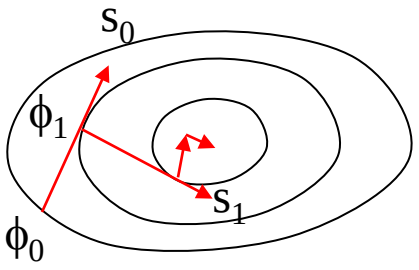
- These equations show that all first derivatives are zero at a solution of the original equations, and if *A* is positive definite, this extremum is a minimum

CGA

- The CGA finds the minimum of a quadratic function of *N* variables in a maximum of *N* iterations
- The CGA successively finds the minima in subspaces of the *N* variables, starting with a one dimensional subspace
- Having found the minimum in one subspace, it expands the subspace by one dimension and locates the new minimum
- When the dimension of the subspace reaches *N*, the number of original equations, the solution is found

Steepest Descent Method

- The gradient r_0 is evaluated at the initial guess ϕ_0
- The initial subspace is defined by the search direction $s_0=-r_0$ which is in the "steepest downhill" direction at ϕ_0
- The minimum of $V(\phi)$ is then computed along the direction s_0
- If w is the distance along s_0, $V(\phi_0+w\,s_0)$ is a scalar quadratic function of w and the minimum at $w=w_0$ can be directly evaluated
- This position $\phi_1= \phi_0+w\,s_0$ defines the next value in the iteration towards the solution



SDM

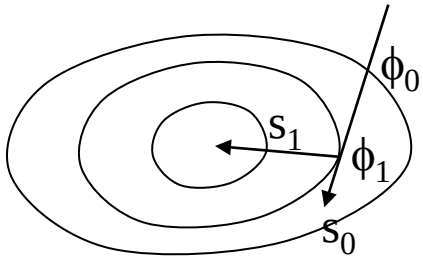
- The steepest descent method can be extremely slowly convergent if the ratio of the maximum to minimum eigenvalues of A (called the condition number) is large
- This happens when the valley around the minimum of V is very long and narrow
- In this case successive iterates tend to bounce across the sides of the valley and only progress very slowly towards the minimum

CGA

- The CGA overcomes this problem by choosing the next search direction s_1 as a linear combination of s_0 and the new steepest-descent direction $-r_1$ at ϕ_1:

$$s_1 = -r_1 + \beta s_0$$

$$\beta = \frac{r_1 \cdot r_1}{r_0 \cdot r_0}$$



- It can be shown that s_1 is the direction of the minimum in the 2D subspace containing r_1 and s_0
- In the 2D example the minimum along s_1 is the solution of the problem, it has been reached in two steps

CGA

- The 2D case can be regarded as a slice through a 3D problem, and the method extends naturally to higher dimensions
- The successive residual vectors have the important property of being orthogonal to each other
- Since there can be only *N* different orthogonal vectors in an *N* dimensional space, the *N* plus first residual vector must be the null vector and the solution has been found

Generalized Minimal Residuals (GMRES)

- Conjugate gradient algorithms fail for non-symmetric matrices
- The problem is that for complex eigenvalues, two search directions are not enough
- Thus, the idea is to take more search directions v^k, $k=1,\ldots, m$
- Defining the residual as:
- Starting vector:

$$r^n = A\phi - \rho$$

$$v^1 = \frac{r^n}{|r^n|}$$

- For $j=1,2,\ldots, m-1$ we have

$$w^{j+1} = A \cdot v^j - \sum_{i=1}^{j} h^{ij} v^i \qquad h^{ij} = v^i \cdot A \cdot v^j$$

$$v^{j+1} = \frac{w^{j+1}}{|w^{j+1}|}$$

GMRES

- It can be shown that the vectors v^k are orthogonal:

$$v^k \cdot v^{j+1} = \frac{1}{|w^{j+1}|}\left[v^k \cdot A \cdot v^j - \sum_{i=1}^{j} (v^i \cdot A \cdot v^j) v^k \cdot v^i \right]$$

$$\text{assuming that first j vectors are orthogonal}$$

$$\Rightarrow \text{only inner product left is k = i :}$$

$$v^k \cdot v^{j+1} = \frac{1}{|w^{j+1}|}\left[v^k \cdot A \cdot v^j - v^k \cdot A \cdot v^i \right] = 0$$

$$\Rightarrow \text{orthogonal}$$

- The space defined by the vectors v^k is called the *Krylov space*

GMRES

- The basic scheme updates the solution as a linear combination of the search vectors v^k:

$$\Delta\phi = v^k \alpha_k$$

- Then the residual is minimized, in least squares sense

$$| A \cdot (\phi + \Delta\phi) - \rho |^2 \rightarrow \min$$

$$\Rightarrow \quad | A \cdot (v^k \alpha_k) - r^n |^2 \rightarrow \min$$

- The solution can then be written as

$$(A \cdot v^k) \cdot (A \cdot v^i) \alpha_i - (A \cdot v^k) \cdot r^n = 0$$

$$\Rightarrow \quad B^{ki} \alpha_i = b^k \quad \text{or} \quad B\alpha = b$$

- The solution of this system is obtained with a direct solver since it is small: N_k x N_k with N_k: Krylov space dimension ~10

GMRES: Properties

- Easy to code
- Easy to maintain
- Easy to vectorize
- Easy to parallelize
- Slow convergence if *A* is badly conditioned
- => need a good preconditioner *P*

Non-Linear Problems

- In a non-linear problem we can define a Newton linearization in order to enter an iterative approximation sequence
- If the non-linear system is of the form

$$A(\phi) = q$$

- The Newton iteration is

$$A(\phi^{n+1}) = A(\phi^n + \Delta\phi) = A(\phi^n) + \left(\frac{\partial A}{\partial \phi}\right) \cdot \Delta\phi = -q$$

- The Jacobian of *A* defines the iterative scheme

$$K \cdot \Delta\phi^n = -R^n \qquad K = \left(\frac{\partial A}{\partial \phi}\right)$$

Non-Linear Problems

- If the linear equation $K\,\Delta\phi^n = -R^n$ is solved by a direct method we would obtain the exact solution $\phi' = \phi^n - K^{-1} R^n$
- The error with respect to the exact solution satisfies

$$K\,e^n = K\,(\phi^n - \phi') = R^n$$

- If the system is solved by an iterative method represented by the preconditioning operator P/τ

$$P/\tau\,\Delta\phi = -R^n$$

- The error will be amplified by an operator G, such that

$$e^{n+1} = \phi^{n+1} - \phi' = e^n + \Delta\phi$$

$$= e^n - \tau P^{-1} R^n$$

$$= (1 - \tau P^{-1} K) e^n = G e^n$$

Non-Linear Problems

- For a non-linear system of equations we can consider that the conditioning operator should be an approximation to the Jacobian matrix of the system (*K*)

- The iterative scheme will converge if the spectral radius (value of the maximum eigenvalue) of *G* is lower or equal to one

Constant and secant stiffness methods

- These names come from finite element applications where the following preconditioning matrices are used

Constant stiffness method: $P = A(\phi^0)$

- The convergence matrix P is taken as the operator A at a previous iteration and kept fixed

Secant stiffness method: $P = A(\phi^{n-1})$

- The Jacobian is approximated by the previous value of the matrix $A(\phi^{n-1})$

- The systems obtained can be solved by any of the methods described, direct or iterative
- If a direct solution is used, we have a *semi-direct* method in the sense that the iterations treat only the non-linearity

HYPERBOLIC EQUATIONS

Hyperbolic PDE solvers

In general the solution of PDE's proceeds as follows:

1. Construct a spatial grid to represent continuous functions
2. Build approximations to the spatial derivatives on this grid (spatial discretization)
3. Obtain a system of ODE's
4. Construct a temporal FD scheme to solve the ODE system
5. Advance the solution in time

Example: transient heat equation

Given the 1D heat equation: $u_{,t} = u_{,xx}$

1. Construct spatial grid:
   - Uniform 1D grid: element size $h = x_{i+1} - x_i = const.$
2. Build approximations to the derivatives on the grid:
   - FD approximation: $u_{,xx} = \frac{1}{h^2}(u_{i+1} - 2u_i + u_{i-1})$
3. Obtain coupled system of ODE's:
   - For each point *i*: $u_{i,t} = \frac{1}{h^2}(u_{i+1} - 2u_i + u_{i-1})$
   - Matrix form: $\mathbf{u}_{,t} = \mathbf{K} \cdot \mathbf{u}$

Example: transient heat equation

4. Construct the FD temporal scheme to solve the ODE system:

$$u_i^{n+1} = u_i^n + \frac{\Delta t}{h^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

5. Code it and solve it

Remark 1: you will need to estimate the allowable time-step for your solution scheme

Remark 2: you will also need to build approximations of the boundary conditions and take them into account in the solution algorithm

Explicit and implicit schemes

- Typically time-dependent PDE’s are first discretized in space using any of the methods described before
- This results in a system of coupled ODE’s that is then discretized usually using finite differences
- Numerical schemes are then classified as explicit or implicit depending on how the time derivatives are discretized

Explicit and implicit schemes: example

$$\text{1D unsteady heat equation: } u_{,t} = k\,u_{,xx}$$

$$\text{explicit scheme: } u_i^{n+1} - u_i^n = \frac{k\,\Delta t}{\Delta x^2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)$$

$$\text{implicit scheme: } u_i^{n+1} - u_i^n = \frac{k\,\Delta t}{\Delta x^2}\left(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}\right)$$

Mixed schemes

- Hybrid methods are obtained by linear combination of explicit and implicit schemes:

$$u_{,t} = \theta u_{explicit} + (1-\theta) u_{implicit}$$

$$u_i^{n+1} - u_i^n = (1-\theta)\frac{k\Delta t}{\Delta x^2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \theta\frac{k\Delta t}{\Delta x^2}\left(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}\right)$$

- θ=0 : Forward Euler (explicit)
- θ=1/2: Crank-Nicholson (hybrid)
- θ=1 : Backward Euler (fully implicit)

Advancing explicit systems

$$\text{explicit scheme:} \quad u_i^{n+1} - u_i^n = \frac{k\,\Delta t}{\Delta x^2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)$$

$$\text{updating scheme:} \; u_i^{n+1} = \frac{k\,\Delta t}{\Delta x^2}u_{i+1}^n - (1 - 2\frac{k\,\Delta t}{\Delta x^2})u_i^n + \frac{k\,\Delta t}{\Delta x^2}u_{i-1}^n$$

$$\text{matrix form:} \quad \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_m^{n+1} \end{bmatrix} = \begin{bmatrix} a & b & 0 & 0 & 0 & \cdots & 0 \\ a & b & a & 0 & 0 & \cdots & 0 \\ 0 & a & b & a & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & a & b \end{bmatrix} \cdot \begin{bmatrix} u_1^n \\ u_2^n \\ u_3^n \\ \vdots \\ u_m^n \end{bmatrix}$$

$$\text{with} \quad a \equiv \frac{k\,\Delta t}{\Delta x^2} \quad b \equiv 1 - 2\frac{k\,\Delta t}{\Delta x^2}$$

Advancing implicit systems

$$\text{implicit scheme:} \quad u_i^{n+1} - u_i^n = \frac{k\,\Delta t}{\Delta x^2}\left(u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}\right)$$

$$\text{updating scheme:} \quad -\frac{k\,\Delta t}{\Delta x^2}u_{i+1}^{n+1} + (1 + 2\frac{k\,\Delta t}{\Delta x^2})u_i^{n+1} - \frac{k\,\Delta t}{\Delta x^2}u_{i-1}^{n+1} = u_i^n$$

$$\text{matrix form:} \quad \begin{bmatrix} a & b & 0 & 0 & 0 & \cdots & 0 \\ a & b & a & 0 & 0 & \cdots & 0 \\ 0 & a & b & a & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & a & b \end{bmatrix} \cdot \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ u_3^{n+1} \\ \vdots \\ u_m^{n+1} \end{bmatrix} = \begin{bmatrix} u_1^n \\ u_2^n \\ u_3^n \\ \vdots \\ u_m^n \end{bmatrix}$$

$$\text{with} \quad a \equiv -\frac{k\,\Delta t}{\Delta x^2} \quad b \equiv 1 + 2\frac{k\,\Delta t}{\Delta x^2}$$

Explicit vs Implicit

$$\text{explicit scheme: } u^{n+1} = A \cdot u^{n}$$

$$\text{implicit scheme: } A \cdot u^{n+1} = u^{n}$$

- Explicit schemes require matrix multiplication
- Implicit schemes require matrix invertion (solution of algebraic system at each time step)

- So, why use implicit schemes ? …

Analysis of numerical schemes

Consistency

- Approximation $\rightarrow$ PDE for $\Delta t, \Delta x \rightarrow 0$

Stability

- Long term effects of local and round-off errors

Convergence

- Approximation $\rightarrow$ exact solution for $\Delta t, \Delta x \rightarrow 0$

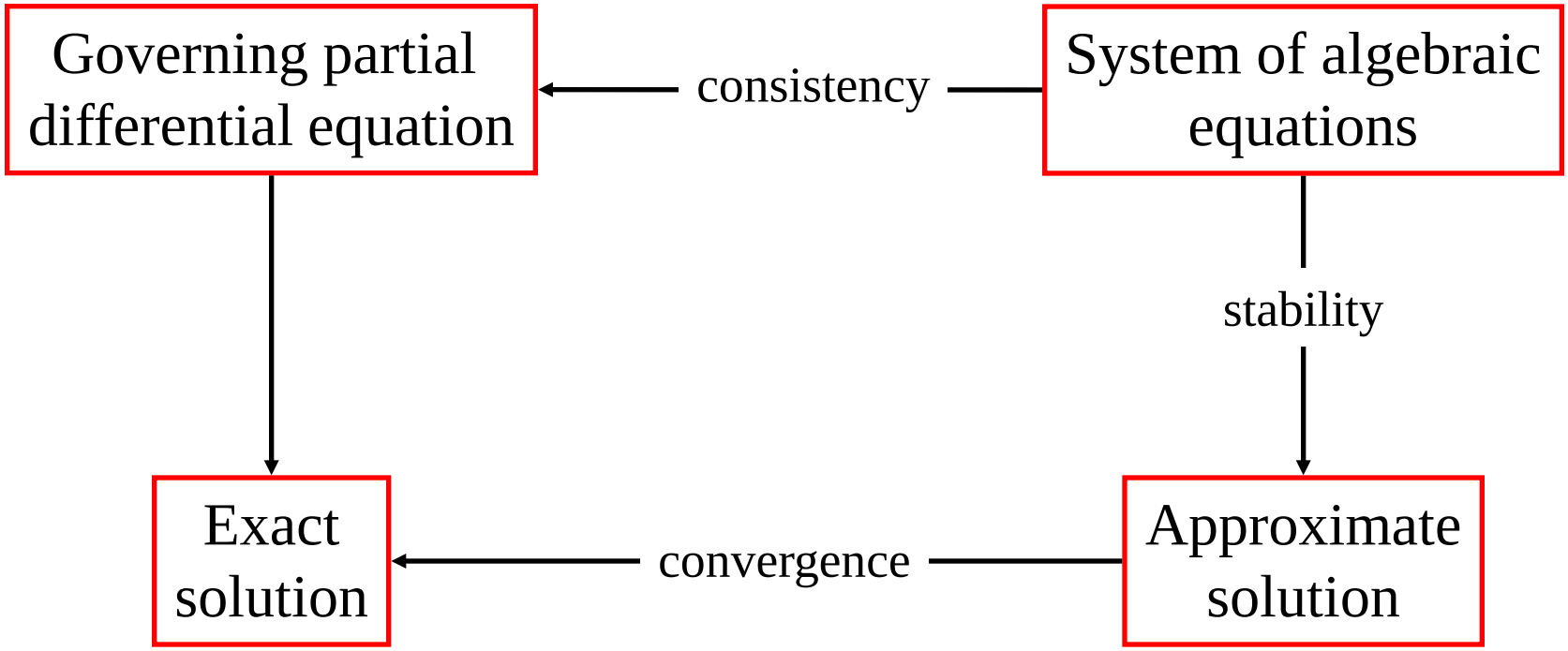
Accuracy

- Magnitude of local errors

Efficiency

- CPU and storage vs accuracy

Consistency, Stability & Convergence



Consistency

- Consistency expresses that the discretized equations should tend to the differential equations when Δt and Δx tend to 0
- Condition on structure of numerical formulation
- Discretized equation $\Leftrightarrow$ differential equation

Consistency: example

$$\text{Laplacian in 1D}: \; u_{,xx} = 0$$

$$\text{Approximation}: \; \frac{1}{\Delta x^2}\left(u_{i-1} - 2u_i + u_{i+1}\right) = 0$$

$$u_{i+1} = u(x+\Delta x) \approx u(x) + \Delta x\, u_{,x} + \frac{\Delta x^2}{2} u_{,xx} + \frac{\Delta x^3}{6} u_{,xxx} + \frac{\Delta x^4}{24} u_{,xxxx} + \cdots$$

$$u_{i-1} = u(x-\Delta x) \approx u(x) - \Delta x\, u_{,x} + \frac{\Delta x^2}{2} u_{,xx} - \frac{\Delta x^3}{6} u_{,xxx} + \frac{\Delta x^4}{24} u_{,xxxx} - \cdots$$

$$\Rightarrow \quad \lim_{\Delta x \to 0}\left[u_{,xx} - \frac{1}{\Delta x^2}\left(u_{i-1} - 2u_i + u_{i+1}\right)\right] = -\frac{\Delta x^2}{12} u_{,xxxx} + \cdots = 0$$

In general, not difficult to prove

Accuracy

- The order of accuracy of a method is defined by expressing the measure of error in powers of the grid size *h*

$$E \approx h^{p}$$

Accuracy: example

$$\text{Laplacian in 1D}: \; u_{,xx} = 0$$

$$\text{Approximation}: \; \frac{1}{\Delta x^2}\left(u_{i-1} - 2u_i + u_{i+1}\right) = 0$$

$$\text{Taylor series}: \quad u_{,xx} = \frac{1}{\Delta x^2}\left(u_{i-1} - 2u_i + u_{i+1}\right) - \frac{\Delta x^2}{12} u_{,xxxx} + \cdots$$

$$\Rightarrow \quad \text{Leading term of error}: \frac{\Delta x^2}{12} u_{,xxxx}$$

$$\Rightarrow \quad \text{Second order accurate}$$

Stability

- The numerical scheme should not allow errors to grow unbounded, i.e. amplified without bound between two steps
- Condition on solution of numerical scheme
- Numerical solution $\Leftrightarrow$ exact solution of discretized equation

Stability: example

1D heat equation : $u_{,t} = u_{,xx}$

numerical scheme: $u_i^{n+1} - u_i^n = \frac{\Delta t}{\Delta x^2}\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right)$

Fourier series: $u = \sum_{j=1}^{\infty} g_j^p \exp\left(\frac{i\pi x}{j\Delta x}\right)$

Fourier mode: $u_m = g_j^p \exp\left(\frac{i\pi x}{j\Delta x}\right)$

stability : $g_j < 1$

most constraining mode: $\pi/2 \Rightarrow j = 1 \quad \Rightarrow \quad \Delta t < \Delta x^2/2$

May be very constraining for small Δx

Determining stability is more difficult but essential

Convergence

- The numerical solution should approach the exact solution of the differential equation as Δt and Δx tend to zero
- Condition on solution of numerical scheme
- Numerical solution $\Leftrightarrow$ exact solution of differential equation

Lax equivalence theorem

- *For a well-posed initial value problem and a consistent discretization scheme, stability is the necessary and sufficient condition for convergence:*

consistency & stability $\Rightarrow$ convergence

Analysis of numerical schemes:

1. Analyze the consistency condition: this leads to the determination of the order of accuracy of the scheme and its truncation error
2. Analyze the stability properties: this leads to detailed on the frequency distribution of the error

➢ From these two steps convergence can be established

Schemes for hyperbolic equations

- Hyperbolic PDE's are in general remarkably more difficult to solve than elliptic or parabolic equations

- Many interesting problems are described by hyperbolic differential equations:
  - Gas dynamics
  - Dispersion transport
  - Climate modeling
  - Acoustics
  - Electromagnetics
  - …

Model equations

- We will illustrate the schemes for hyperbolic equations on the 1D conservation law, which can be written in conservation form as:

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = q$$

- Or in quasi-linear form as:

$$\frac{\partial u}{\partial t} + a(u)\frac{\partial u}{\partial x} = q \quad \text{with} \quad a(u) = \frac{\partial f}{\partial u}$$

- Particular (non-trivial) cases:

$$\text{Burgers equation}: \quad \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0$$

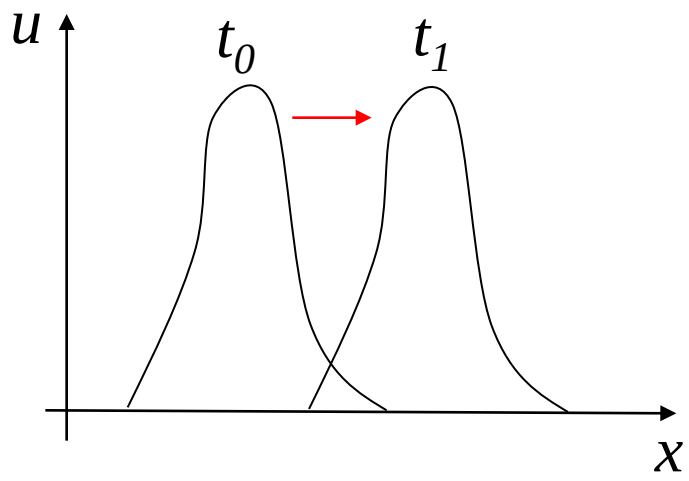
$$\text{First order wave equation}: \frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0 \quad a = const$$

1D linear advection equation

- When the speed *a* is constant (first order wave equation) this equation becomes linear and has analytical solution:

$$u = u_0(x - at)$$

- This solution represents a pure translation with speed *a* with no deformation or dispersion of the initial “disturbance” (no new maxima or minima, i.e. *monotonicity* is preserved)



- This turns out to be a very difficult problem !

Explicit first order space-centered scheme

- The simplest scheme that can be written for the 1D advection equation is forward Euler in time and central differences in space:

$$u_i^{n+1} - u_i^n = -\frac{\sigma}{2}\left(u_{i+1}^n - u_{i-1}^n\right)$$

$$= -\frac{\sigma}{2}\left[a\left(u_{i+1}^n + u_i^n\right) - a\left(u_i^n + u_{i-1}^n\right)\right]$$

$$= -\frac{\sigma}{2}\left(f_{i+1/2}^n - f_{i-1/2}^n\right)$$

$$\text{Flux}: \quad f = a\,u$$

$$\text{Courant or CFL number}: \sigma = \frac{a\,\Delta t}{\Delta x}$$

- 1st order in time, 2nd order in space
- Unconditionally unstable !

Lax-Friedrichs scheme

- The stabilizing procedure consists of the following replacement:

$$u_i^n \rightarrow \left(u_{i+1}^n + u_{i-1}^n\right)/2$$

$$\Rightarrow \quad u_i^{n+1} = \frac{1}{2}\left(u_{i+1}^n + u_{i-1}^n\right) - \frac{\sigma}{2}\left(u_{i+1}^n - u_{i-1}^n\right)$$

- In conservation form:

$$u_i^{n+1} = \frac{1}{2}\left(u_{i+1}^n + u_{i-1}^n\right) - \frac{\Delta t}{2\Delta x}\left(f_{i+1}^n - f_{i-1}^n\right)$$

$$= u_i^n - \frac{\Delta t}{2\Delta x}\left(f_{i+1}^n - f_{i-1}^n\right) + \frac{1}{2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)$$

Numerical dissipation

- The last term of the RHS can be thought as the 2nd order central difference discretization of $u_{,xx}$

- Therefore, the Lax-Friedrichs scheme can be viewed as being obtained from an explicit forward Euler time integration of an equation of the form:

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = \alpha \frac{\partial^2 u}{\partial x^2}$$

$$\alpha = \frac{\Delta x^2}{2\Delta t}$$

- This is a convection-diffusion equation with *numerical viscosity or dissipation* α

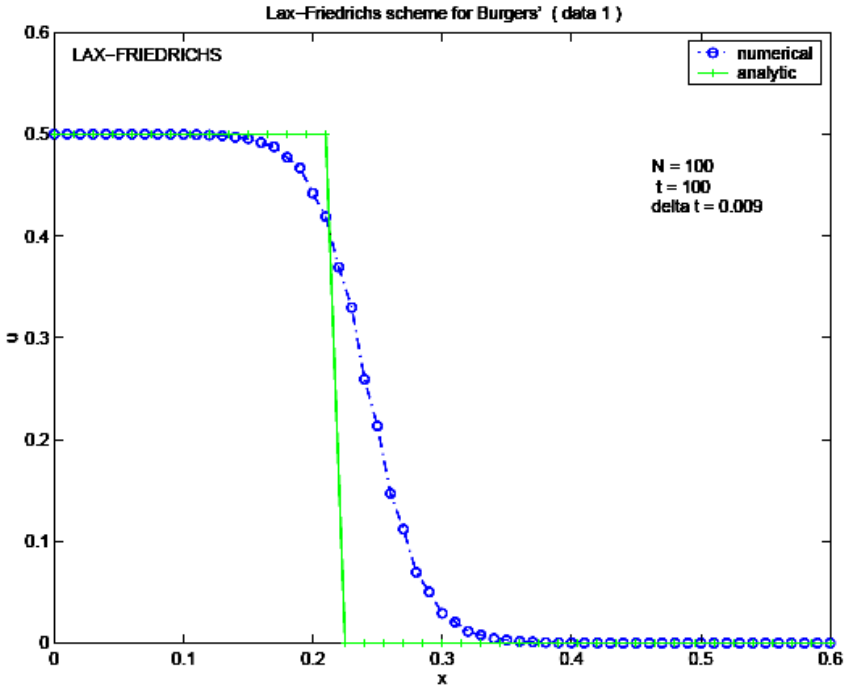
Properties

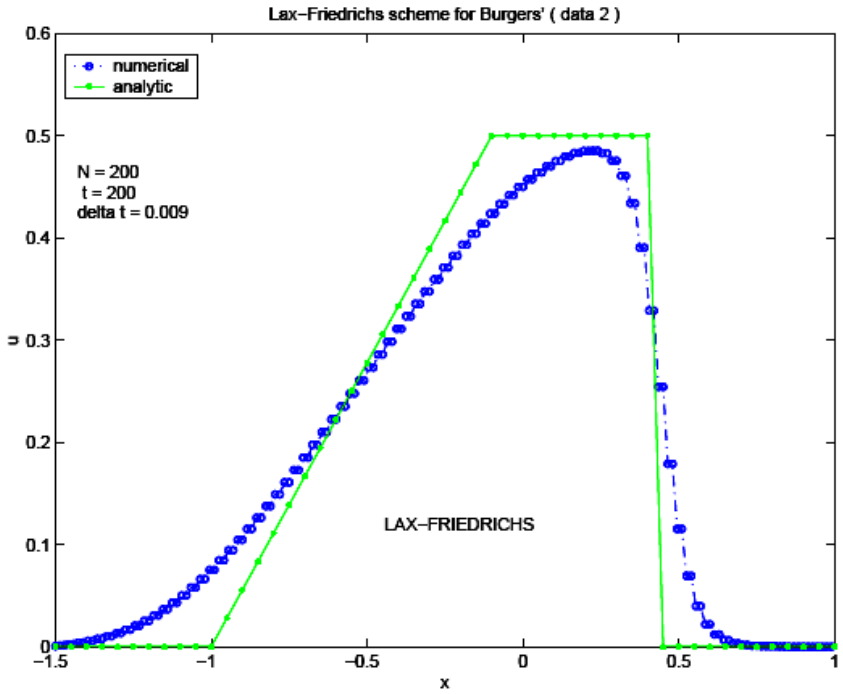
- This scheme is first order in time and space
- This scheme is monotonicity preserving

- The Lax-Friedrichs scheme is *conditionally stable*
- The stability or CFL condition is:

$$\sigma = \frac{a\,\Delta t}{\Delta x} \leq 1$$

Solutions





- Numerical dissipation => results too “smeared”

Upwind schemes

- Another alternative for stabilizing our original scheme is to replace the central difference formula for the spatial derivative by a one-sided formula in the direction of propagation (i.e. upwind formula):

$$u_i^{n+1} = u_i^n - \sigma\left(u_i^n - u_{i-1}^n\right)$$

$$\Rightarrow \quad f_{i+1/2}^n = a\left(u_{i+1}^n + u_i^n\right) - |a|\left(u_{i+1}^n - u_i^n\right)$$

Properties

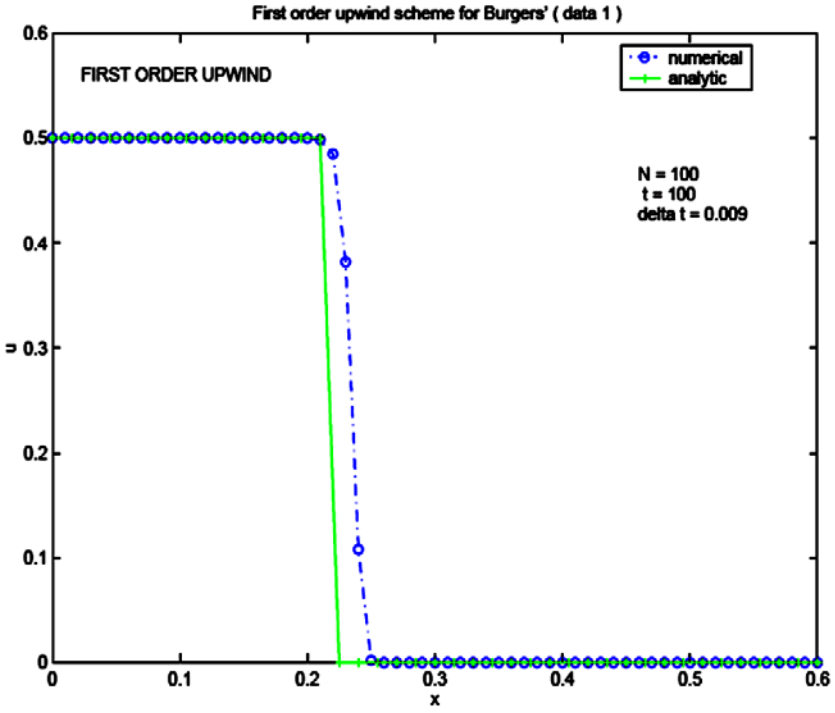
- First order accurate in space and time
- Monotonicity preserving scheme

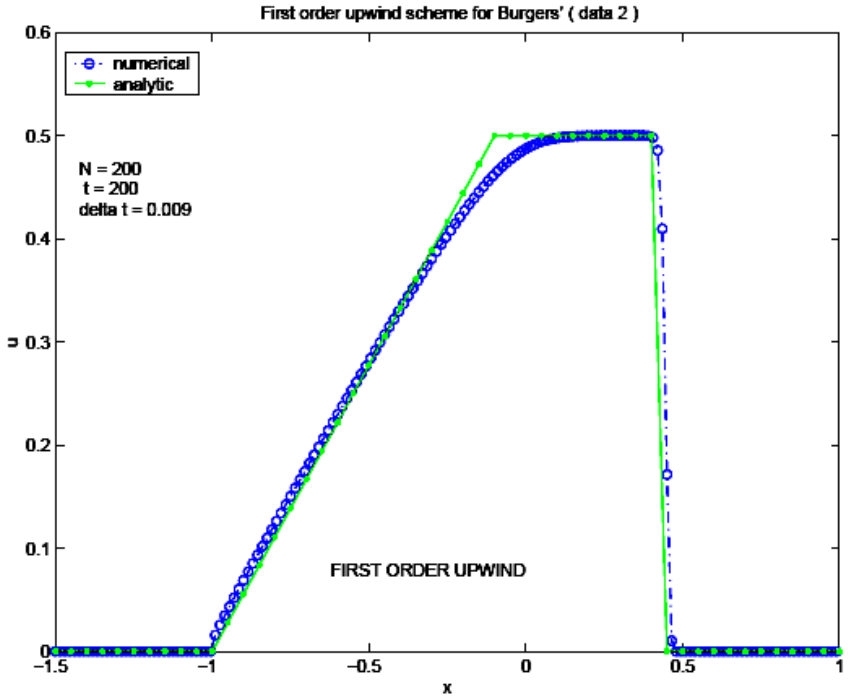
- This scheme is also conditionally stable for

$$\sigma = \frac{a\,\Delta t}{\Delta x} \leq 1$$

- The numerical viscosity term vanishes for *a=0* (stagnation regions) and comparing to the Lax-Friedrichs scheme it is smaller by a factor of $(\sigma+1)/\sigma$

Solutions





- Less numerical dissipation => less “smeared” solutions

Lax-Wendroff scheme

- Basic idea: combine space and time discretization to achieve second order accuracy

- The Taylor series expansion in time yields:

$$u(t+\Delta t)-u(t)=\Delta t\,u_{,t}+\frac{\Delta t^2}{2}u_{,tt}+\cdots$$

- Inserting the advection equation repeatedly:

$$u(t+\Delta t)-u(t)=-\Delta t\,au_{,x}+\frac{a^2\Delta t^2}{2}u_{,xx}+\cdots$$

$$\Rightarrow\quad u_i^{n+1}-u_i^n=-\frac{a\Delta t}{2\Delta x}\left(u_{i+1}^n-u_{i-1}^n\right)+\frac{a^2\Delta t^2}{\Delta x^2}\left(u_{i+1}^n-2u_i^n+u_{i-1}^n\right)$$

Properties

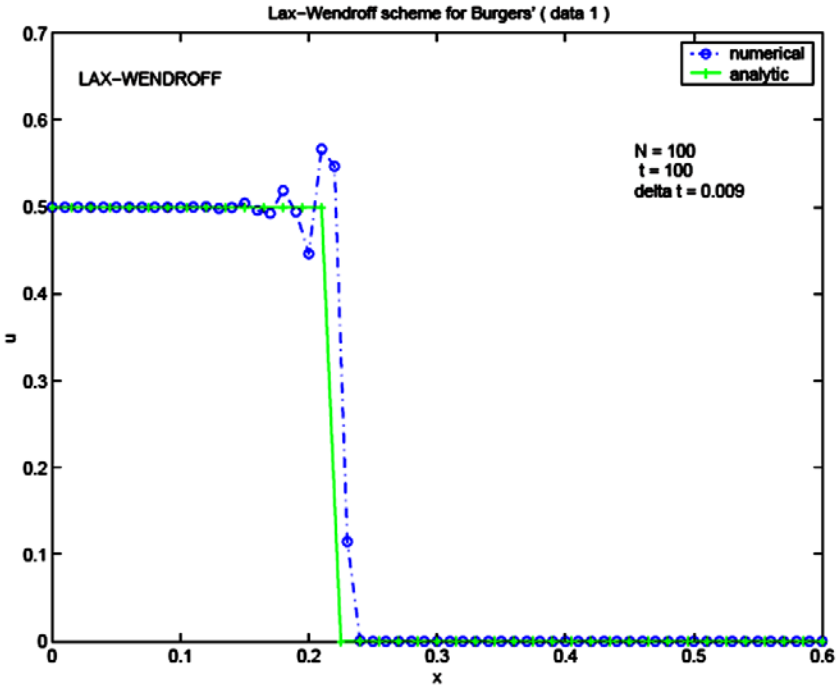
- Second order in space and time
- Conditionally stable for

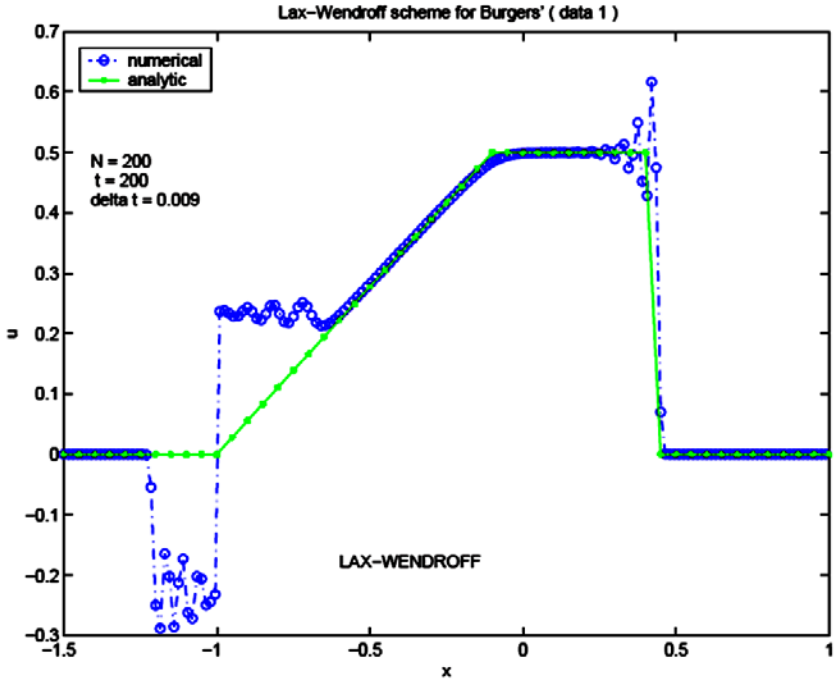
$$\sigma = \frac{a\,\Delta t}{\Delta x} \leq 1$$

- Not monotonicity preserving !

- The numerical dissipation term is fourth order

Solutions





- Solutions contain non-physical oscillations (too little numerical dissipation where it is needed)

Artificial viscosity

- Second order methods such as Lax-Wendroff generate oscillations around discontinuities
- First-order schemes have truncation errors proportional to a second derivative that acts as an added numerical viscosity.
- Therefore, these schemes will damp the high-frequency components and smooth out gradients

Artificial viscosity

- In order to remove the high-frequency oscillations around discontinuities Von Neumann & Richtmyer introduced the concept of artificial viscosity

- These additional terms should simulate the effects of physical dissipation, *on the scale of the mesh*, around discontinuities and be negligible (of an order equal or larger than the truncation error) in smooth regions

- Many artificial viscosity terms have been devised for different hyperbolic equations

Godunov's theorem

- No linear scheme of order higher than one is monotonicity preserving

- First order is not accurate enough for practical applications !

➢ Need non-linear schemes

Flux Corrected Transport schemes

Basic idea:

- We desire a high order scheme
- But a low order scheme is safe
- Then write a scheme of the form:

$$u^{n+1} = u^n + \Delta u^h = u^n + \Delta u^l + (\Delta u^h - \Delta u^l)$$

- Introduce a “limitor” function to limit the “anti-diffusive” fluxes:

$$u^{n+1} = u^n + \Delta u^l + c \cdot (\Delta u^h - \Delta u^l)$$

- The limitor is designed such that:
  - *c=0* => low order scheme
  - *c=1* => high order scheme

“Limitor” function

- The aim in the design of the limitor function is to create no new minima or maxima by adding the “anti-diffusive” term

$$(\Delta u^h - \Delta u^l)$$

- First the allowed values for u^{n+1} are obtained
- Then the worst case scenario is evaluated:
  - All positive fluxes => possible new maxima
  - All negative fluxes => possible new minima
- Then limit the anti-diffusive terms so that the new values are within the acceptable range of values (given by the low order scheme)
- These schemes yield remarkably good results for conservation laws or systems of conservation laws

Multidimensional Upwind Scheme

Advection equation : $\frac{\partial u}{\partial t} + \mathbf{v} \cdot \nabla u = 0$

if $v_x > 0$ *and* $v_y > 0$

$$u_{ij}^{n+1} = u_{ij}^{n} - \sigma_x \left(u_{ij}^{n} - u_{i-1\,j}^{n} \right) - \sigma_y \left(u_{ij}^{n} - u_{ij-1}^{n} \right)$$

if $v_x < 0$ *and* $v_y > 0$

$$u_{ij}^{n+1} = u_{ij}^{n} - \sigma_x \left(u_{i+1\,j}^{n} - u_{ij}^{n} \right) - \sigma_y \left(u_{ij}^{n} - u_{ij-1}^{n} \right)$$

...

$\sigma_x = v_x \, \Delta t / \Delta x \qquad \sigma_y = v_y \, \Delta t / \Delta x$

Multidimensional Lax-Wendroff Scheme

$$\text{Taylor expansion}: \Delta u = u(t+\Delta t) - u(t) = \Delta t\, u_{,t} + \frac{\Delta t^2}{2} u_{,tt} + \cdots$$

$$\text{2D advection equation}: u_{,t} = -v_x\, u_{,x} - v_y\, u_{,y}$$

$$\text{derivatives}: u_{,tx} = -v_x\, u_{,xx} - v_y\, u_{,yx} \qquad u_{,ty} = -v_x\, u_{,xy} - v_y\, u_{,yy}$$

$$u_{,tt} = -v_x\, u_{,tx} - v_y\, u_{,ty}$$

$$\text{replacing}: \Delta u = \Delta t\,(-v_x u_{,x} - v_y u_{,y}) + \frac{\Delta t^2}{2}(v_x{}^2 u_{,xx} + v_y{}^2 u_{,yy} + 2 v_x v_y u_{,xy})$$

$\Rightarrow$ use central differences

FINITE VOLUME METHOD

Finite volume method

- A *conservation law* can be written in the general integral form

$$\frac{\partial}{\partial t}\int_V u\,dV + \oiint_{S(V)} \mathbf{F}\cdot\mathbf{dS} = \int_V q\,dV$$

- If the flux function is continuous, we can apply Gauss' theorem to get the equivalent partial differential equation

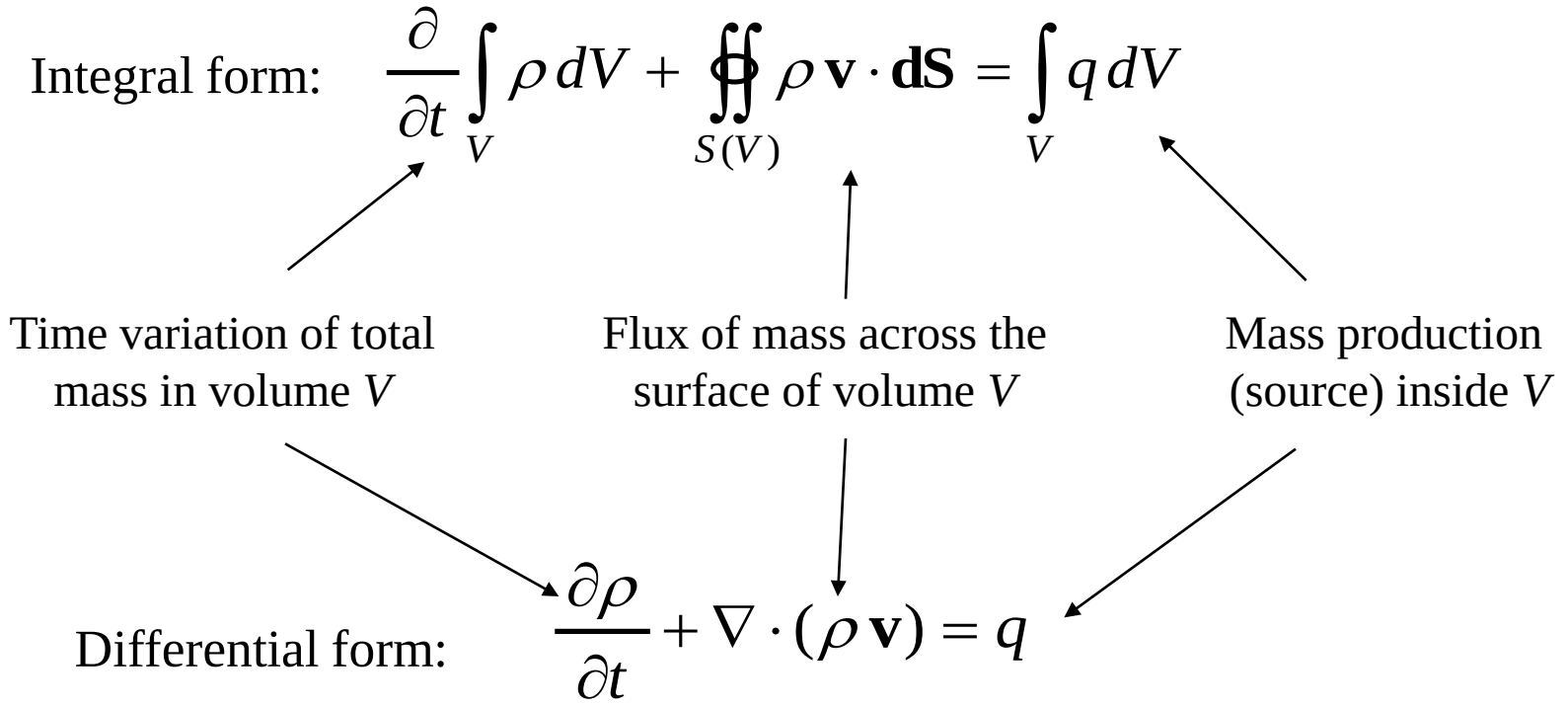
$$\int_V \frac{\partial u}{\partial t}dV + \int_V \nabla\cdot\mathbf{F}\,dV - \int_V q\,dV = \int_V \left(\frac{\partial u}{\partial t} + \nabla\cdot\mathbf{F} - q\right)dV = 0$$

$$\Rightarrow \quad \frac{\partial u}{\partial t} + \nabla\cdot\mathbf{F} = q$$

- The finite volume method is based on the integral form of conservation laws

Example: mass conservation

- Consider the time variation of the mass of fluid of density ρ flowing through a volume V with a velocity $\mathbf{v}$:



Basis of the finite volume method

- Suppose the computational domain is divided into non-overlapping sub-volumes, then the integral form of the conservation law can be written for each of these sub-volumes or control cells:

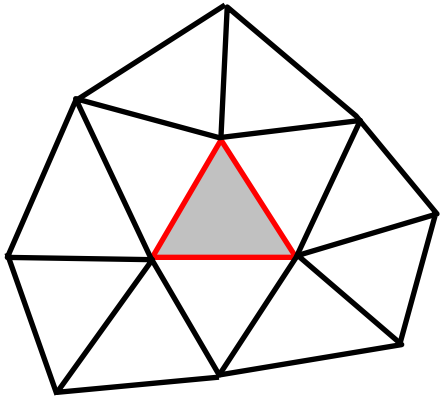
$$\frac{\partial}{\partial t}(u_i \, v_i) + \sum_{faces} \mathbf{F} \cdot \mathbf{s} = q_i \, v_i$$

  where u_i is the average of u over the sub-volume v_i , and the sum of the flux terms refers to all external sides or faces of the control cell i

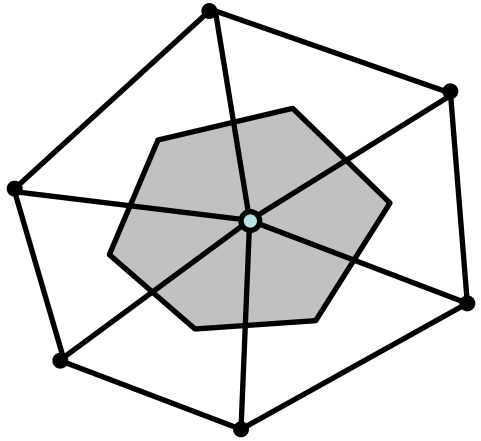
- By summing over all control volumes the original integral conservation law is satisfied

Control volumes

- This is the general formulation of the finite volume method
- In addition to the control volumes, it is necessary to select the way to compute the cell volumes and face areas, and how to approximate the fluxes at the faces
- The FV method is quite general and can handle any kind of mesh (restricted to elements with rectilinear sides)



Cell-centered scheme



Node-centered scheme

Finite volumes in 3D

- The finite volume method is applied in a straightforward manner in three dimensions
- Typically, the space is subdivided into hexahedral or tetrahedral elements
- Expressions for evaluating the volume of these elements and the areas of their faces (quads or triangles) are available

Evaluation of fluxes at cell faces

- The evaluation of fluxes at cell faces depends on the selected scheme and on the location of the unknown variables
- *Central scheme*, Option 1:

$$f_{AB} = f\left(\frac{u_A + u_B}{2}\right)$$

- *Central scheme*, Option 2:

$$f_{AB} = \frac{1}{2}(f_A + f_B) \qquad f_A = f(u_A), \quad f_B = f(u_B)$$

- The second choice corresponds to the trapezoidal formula for the flux integral

Central and upwind schemes

- The *central schemes* take the average of the fluxes on both sides of the cell face
- The *upwind schemes* take the flux from the cell upstream of the cell face. The propagation direction of the associated convection speed is determined by the flux Jacobian

$$\mathbf{A}(u) = \frac{\partial \mathbf{F}}{\partial u}$$

- Example: mass conservation law

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0$$

$$\mathbf{F} = \rho \mathbf{v} \quad \Rightarrow \quad \frac{\partial \mathbf{F}}{\partial \rho} = \mathbf{v}$$

Conservative discretizations

- Consider the 1D conservation law

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = q$$

- Discretize this equation with a central difference scheme

$$\text{point } i: \quad \frac{\partial u_i}{\partial t} + \frac{f_{i+1/2} - f_{i-1/2}}{\Delta x} = q_i$$

$$\text{point } i+1: \quad \frac{\partial u_{i+1}}{\partial t} + \frac{f_{i+3/2} - f_{i+1/2}}{\Delta x} = q_{i+1}$$

$$\text{point } i-1: \quad \frac{\partial u_{i-1}}{\partial t} + \frac{f_{i-1/2} - f_{i-3/2}}{\Delta x} = q_{i-1}$$

- The sum of these equations yields a consistent discretization for the conservation law in the interval *(i-3/2, i+3/2)*:

$$\frac{\partial}{\partial t}\frac{(u_{i+1} + u_i + u_{i-1})}{3} + \frac{f_{i+3/2} - f_{i-3/2}}{3\Delta x} = \frac{1}{3}(q_{i+1} + q_i + q_{i-1})$$

Conservative schemes

- This happens because the internal fluxes cancel out, therefore the discretization is said to be conservative (no numerical volumetric sources)
- The conservation property is satisfied if the numerical scheme can be written as

$$\frac{\partial u_i}{\partial t} + \frac{f^*_{i+1/2} - f^*_{i-1/2}}{\Delta x} = q_i$$

  where *f** is called the *numerical flux*
- For instance, flux $f_{i+1/2}$ is added to cell *i* and subtracted from cell *i+1* , thus the global conservation of the quantity *u* is guaranteed
- This is exactly what we get with a finite volume method

ORDINARY DIFFERENTIAL EQUATIONS

Reducing the order of ODE's

- If you have an ODE of second order or higher, you can always write it as a set of coupled, first order equations
- Given $$\frac{d^2y}{dx^2} + p(x)\frac{dy}{dx} + q(x)y = g(x)$$
- We can define *v(x)* so the set

$$\frac{dv}{dx} + p(x)v + q(x)y = g(x)$$

$$v(x) = \frac{dy}{dx}$$

  is equivalent to the original 2nd order equation

- This is an essential trick for numerical solution of ODE's

Example: harmonic oscillator

$$\text{second order equation}: \quad \frac{d^2 y}{dt^2} = -ky$$

$$\text{first order system}: \quad \frac{dv}{dt} = -ky$$

$$v(t) = \frac{dy}{dt}$$

$$\text{initial contitions}: \quad y(t=0) = y_0, \quad v(t=0) = v_0$$

Numerical solution of ODE's

- A first order non-linear ODE can be written in the form

$$\frac{dy}{dt} = f(t, y) \qquad y(t_0) = y_0$$

where the independent variable *t* increases from an initial value t_0. The analogous set of coupled ODE's can be written in vector form:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}) \qquad \mathbf{y}(t_0) = \mathbf{y}_0$$

- These are called *initial value problems*. Given the initial conditions $\mathbf{y}_0$ we want to integrate this system forward in time to determine the value of $\mathbf{y}(t)$ at later times.

Finite Differences for ODE's

- We will concentrate on *finite difference methods*
- These methods divide the interval of time t over which we wish to integrate the equations into a discrete set of values $\{t_n\}$, where $n=0,1,\ldots,N$
- The intervals or timesteps between the various values t_n are denoted as $\{h_n\}$, where $h_n=t_n - t_{n-1}$

Numerical schemes for ODE's

- A scalar ODE algorithm can be written as

$$y_{n+1} = y_n + h_n F\left(h_n, \{y_i\}, \left\{ \left. \frac{dy}{dt} \right|_i \right\}, \ldots \right)$$

  where *i=0,1,…,n+1* indicates the various discrete steps at which the solution has been calculated. The numerical schemes that advances y_n is embodied in the function *F*

- The most useful numerical schemes depend on at most a few previous values, i.e. *i=n, n-1, n-2, …*

Explicit and implicit methods

- Explicit methods: the value of y_{n+1} depends only on values at previous steps (n, $n-1$, …)

- Implicit schemes: the value of y_{n+1} depends also on values at step $n+1$

Explicit vs implicit schemes

Explicit schemes

- Easy to code
- Easy to apply boundary conditions
- Easy to vectorize / parallelize
- Easy to maintain / upgrade
- No limitation on the order of accuracy
- Step size limited by stability constraint

Implicit schemes

- Can use arbitrary step size
- Order of accuracy for stable schemes limited to 2
- Large overhead in solving large systems of equations

Situations where implicit schemes pay off

- Stiff system of equations / physical stiffness:

  relevant or physical step size >> step size required for stability

Caution

- If step size too large and non-linear system:

$\Rightarrow$ Chaotic solutions possible

$\Rightarrow$ Need to conduct convergence study

- Possible solution: step size adaptivity / sensing in time

Classical methods of solution

- One-step methods
- Linear multi-step methods
- Extrapolation methods
- Leap-frog integration methods
- Implicit methods for stiff ODE’s

One-step methods

- Any one-step method can be written as

$$y_{n+1} = y_n + h\,F(t_n, y_n, h)$$

- The derivative approximation *F* can be evaluated either from a Taylor series expansion about t_n

$$y_{n+1} = y_n + h\frac{dy_n}{dt} + \frac{h^2}{2!}\frac{d^2 y_n}{dt^2} + \cdots$$

or from approximations of the function *F* in the integral form

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} F\big(t, y(t)\big)dt$$

- In the latter case, approximations to *F* typically take the form of polynomials or exponentials

Euler method

- The simplest one-step method is the explicit, first order Taylor algorithm or Euler method:

$$y_{n+1} = y_n + h\, f(t_n, y_n)$$

- In this method, we truncate the Taylor series after the first term ~$O(h)$

- Taylor series algorithms for higher orders can be written analogously from the Taylor series expansion. Typically the higher derivatives are found from analytical differentiation of the original ODE's and substituting into the Taylor series

Runge-Kutta methods

- RK methods are efficient, easily programmed, one-step algorithms that generally give higher order accuracy than Taylor series methods

- Their gains come from evaluating the function *f(t,y)* at more than one point in the neighborhood of (t_n, y_n) instead of evaluating higher derivatives

- The function *F* is expressed as a weighted average of first derivatives obtained numerically at points in the region [t_n , t_{n+1}]

Runge-Kutta schemes

- The generic form for an N-stage Runge-Kutta scheme is

$$k_i = f\left(t_n + h\,a_i , y_n + h\sum_{j=1}^{N} b_{ij}\, k_j \right)$$

where *i=1,2,…, N* labels the stage, and

$$y_{n+1} = y_n + h\sum_{i=1}^{N} c_i\, k_i$$

- Explicit schemes are obtained when all *k* values used are calculated at an earlier step, i.e. when $b_{ij}=0$ for all $j \geq i$

Modified Euler method

- Explicit
- 2nd order accurate
- Two-stage method with $a_2=1/2$, $b_{21}=1/2$, $c_2=1$ and all other constants zero:

$$k_1 = f(t_n, y_n)$$
$$k_2 = f(t_n + h/2, y_n + h k_1/2)$$
$$y_{n+1} = y_n + h k_2$$

- The Euler method is used to estimate the value of y at the half step $[t_n, t_{n+1}]$. Then the average estimate of the derivative is a step-centered quantity. No extra storage is required.

Improved Euler method

- Explicit
- 2nd order accurate
- Two-stage method with $a_2=1$, $b_{21}=1$, $c_2=1/2$ and all other constants zero:

$$k_1 = f(t_n, y_n)$$
$$k_2 = f(t_n + h, y_n + h k_1)$$
$$y_{n+1} = y_n + \frac{1}{2} h (k_1 + k_2)$$

- This method takes the average of the old derivative and the first-order estimate of the new derivative at the end of the step
- This method requires extra storage for k_2

Classical Runge-Kutta scheme

- Explicit
- 4^{th} order accurate
- Four-stage method:

$$k_1 = f(t_n, y_n)$$
$$k_2 = f(t_n + h/2, y_n + hk_1/2)$$
$$k_3 = f(t_n + h/2, y_n + hk_2/2)$$
$$k_4 = f(t_n + h, y_n + hk_3)$$
$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

- This is the most commonly used one-step method
- Requires extra storage for k_i

Low storage Runge-Kutta scheme

- Explicit
- N-stage scheme:

$$y_{n+i} = y_n + \alpha_i \, h \, f(y_{n+i-1})$$

$$\alpha_i = \frac{1}{N+1-i}$$

where *i=1, 2, …, N*

- N^{th} order accurate for linear ODE's
- Low storage (only need to store previous stage)

Linear multi-step methods

- A N-step multi-step method can be written in general form:

$$y_{n+i} = h\beta_i \; f_{n+i} + \sum_{j=0}^{i-1} \left(h\beta_j - \alpha_j \; y_{n+j} \right)$$

for *i=1, …, N*

- We typically need to know the set of past quantities

$$\left(t_n, y_n\right), \left(t_{n-1}, y_{n-1}\right), \ldots$$

at equally spaced intervals in *h* to compute y_{n+1}. Thus, results must be stored for several steps back.

Adams-Bashford method

- 4th order accurate
- Explicit scheme:

$$y_{n+1} = y_n + h\, f_n$$
$$y_{n+2} = y_{n+1} + h/2\,(3 f_{n+1} - f_n)$$
$$y_{n+3} = y_{n+2} + h/12\,(23 f_{n+2} - 16 f_{n+1} + 5 f_n)$$
$$y_{n+4} = y_{n+3} + h/24\,(55 f_{n+3} - 59 f_{n+2} + 37 f_{n+1} - 9 f_n)$$

- The N=1 method is the same as the first-order Euler method

Adams-Moulton method

- 4th order accurate
- Implicit scheme:

$$y_{n+1} = y_n + h/2(f_{n+1} + f_n)$$
$$y_{n+2} = y_{n+1} + h/12(f_{n+2} + 8f_{n+1} - f_n)$$
$$y_{n+3} = y_{n+2} + h/24(9f_{n+3} + 19f_{n+2} - 5f_{n+1} + f_n)$$
$$y_{n+4} = y_{n+3} + h/720(251f_{n+4} + 646f_{n+3} - 264f_{n+2} - 106f_{n+1} - 19f_n)$$

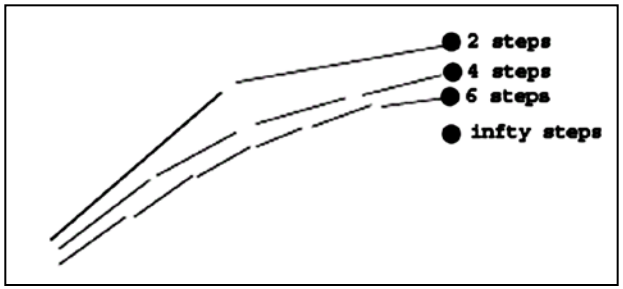
- The N=1 method is called the *trapezoidal method* because it is a trapezoidal quadrature formula

Predictor-corrector methods

- The Adams-Bashford and Adams-Moulton methods are *predictor-corrector* methods because they use a lower order method to predict the answers until enough timesteps have accumulated to carry out the full timestep procedure
- In practice, predictor-corrector methods compare favorable to Runge-Kutta methods, they can be made more accurate with equal computational effort
- Runge-Kutta methods are *self-starting* because they only require data at one time level to begin the integration. Predictor-corrector methods often use a one-step method to accumulate enough values from previous times to proceed
- In Runge-Kutta methods it is easier to vary the timestep (adaptive steps)

Extrapolation methods

- The idea of the Euler-Romberg method is to integrate the equation over the interval *h* many times using Euler's method with *h, h/2, h/4*, …



- Then an extrapolation is made to *h=0* with the increasingly accurate integrations with smaller and smaller *h* values

- This algorithm is selft-starting an rivals the best predictor-corrector schemes for efficiency and accuracy

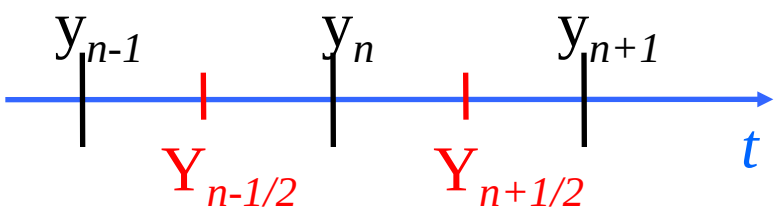
- The choice of step size is fairly arbitrary because the method successively halves the step size until the required accuracy is achieved at each step

Leap-frog integration methods

- In this method, two sets of staggered variables are used:

$$\mathbf{y}_n \quad \text{at} \quad t_{n-1}, t_n, t_{n+1}, \ldots$$

$$\mathbf{Y}_n \quad \text{at} \quad t_{n-1/2}, t_{n+1/2}, \ldots$$



- The system of equations is:

$$\frac{d\mathbf{y}}{dt} = f(\mathbf{Y}, t)$$

$$\frac{d\mathbf{Y}}{dt} = f(\mathbf{y}, t)$$

- Because the derivative of $\mathbf{y}_n$ depends on the dependent variables $\mathbf{Y}_{n+1/2}$ and vice versa, the centered derivative for each of the variables can always be evaluated explicitly using the most recently updated values for the other set of variables

Leap-frog scheme

- The second order explicit leap-frog algorithm is:

$$\mathbf{Y}_{n+1/2} = \mathbf{Y}_{n-1/2} + h\,F(\mathbf{y}_n, t_n)$$
$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\,f(\mathbf{Y}_{n+1/2}, t_{n+1/2})$$
$$t_{n+1/2} = t_n + h/2$$

- This algorithm is reversible, thus long integrations can be traced back to their initial conditions. This property is particularly useful for systems of ODE's that exhibit time reversibility
- One of the major uses of this algorithm is in particle dynamics where the positions $\mathbf{x} \equiv \mathbf{y}$ depend on the velocities $\mathbf{v} \equiv \mathbf{Y}$ and the velocities depend on a force that is only a function of the instantaneous positions

Schemes for stiff ODE's

- Stiffness is related to the presence of a wide range of time scales affecting the system dynamics
- Mathematically, a system is stiff when its Jacobian matrix has eigenvalues whose magnitudes differ by a large ratio
- This means that at least two independent homogeneous solutions vary with time at greatly disparate rates
- Stiff problems make us seek methods that do not restrict the step size for stability reasons, and which can treat widely disparate scales in some reasonable manner

Stiff ODE's

- When there are two or more very different "time-scales" in the problem:
  - You can't integrate with the big time steps, because you will miss the rapid changes
  - You can't integrate with the small time steps, because you may miss the slow changes

- Example: $y' = -1000y + 99e^{-x}$

  solution: $y = e^{-x} - e^{-1000x}$

  the second term dominates very rapidly when $x<0$
  However, for $x>0$, the first term dominates

Backward differentiation formulas

- These are linear multi-step methods:

$$y_{n+i} = h\,\beta_i\; f_{n+i} - \sum_{j=0}^{N-1} \alpha_j\; y_{n+j}$$

with $\alpha_0 \neq 0$ and $\beta_i \neq 0$

- Coefficients for backward differentiation methods:

i	β_i	α_0	α_1	α_2	α_3
1	1	-1	1		
2	2/3	1/3	-4/3	1	
3	6/11	-2/11	9/11	-18/11	1

Backward differentiation methods

- These are the most common methods for solving stiff ODE’s
- The order of accuracy of these methods is equal to the number of steps *N*
- The first order method is the backward Euler method

Exponential methods

- Implicit methods for stiff ODE's are derived by *curve-fitting*: an interpolating function is adopted with free parameters determined by requiring the interpolant to satisfy certain conditions on the approximate solution and its derivatives
- Example: choose the two parameter polynomial function

$$I(t) = A + Bt$$

as the interpolant and require that *I(t)* satisfy

$$I(0) = y_n\,, \quad I'(0) = f_n\,, \quad I(h) = y_{n+1}$$

on the interval [*0,h*]=[t_n , t_{n+1}]. This results in Euler's method:

$$y_{n+1} = y_n + h\, f_n$$

- Other constraints and interpolants result in other previously described explicit and implicit schemes

Exponential methods

- The basis of this approach is that exact solutions of stiff ODE's behave like decaying exponential functions
- Exponentials are poorly described by polynomials when the step size is larger than the characteristic decay rate
- Thus, using exponential approximations should allow considerably longer timesteps

Exponential methods

- Consider the three-parameter exponential interpolant:

$$I(t) = A + Be^{Zt}$$

  for which *A, B* and *Z* must be determined.

- The constraints (same as previous example) determine *A* and *B* in terms of Z:

$$A = y_n - \frac{f_n}{Z}, \qquad B = \frac{f_n}{Z}$$

- Thus, the scheme is:

$$y_{n+1} = y_n + h\, f_n \left[\frac{e^{Zh} - 1}{Zh} \right]$$

Exponential methods

- There are a number of ways to choose the parameter Z:

$$\text{explicit}: \quad Z = f_n' / f_n$$

$$\text{explicit}: \quad Z = \frac{1}{h_{n-1}} \ln\left(\frac{f_n}{f_n - 1} \right)$$

$$\text{implicit}: \quad Z = \frac{1}{h} \ln\left(\frac{f_{n+1}}{f_n} \right)$$

$$\text{implicit}: \quad Z = \frac{1}{2}\left[\frac{f_n'}{f_n} + \frac{f_{n+1}'}{f_{n+1}} \right]$$

$$f' = \frac{d^2 y}{dt^2}$$

- It has been shown that exponential methods can be at least comparable in speed and accuracy to the backward differentiation methods for stiff ODE's

Variable step sizes

- One of the “big ideas” in numerical ODE solvers (and in computational sciences as a whole) is variable step sizes

- Not ALL steps need to be the same size. The step size taken should depend on the local behavior of the function

- Taking a previous example: $y = e^{-x} - e^{-1000x}$

  At $x>>1$ the function changes very slowly $\Rightarrow$ large steps
  For $0<x<1$, it changes very rapidly $\Rightarrow$ smaller steps

- The trick is understanding when to change the step size

Adaptive step sizes

- The fundamental ingredient for an adaptive step size method is a way to estimate the error of the numerical approximation
- There are two basic ways of checking the local error:
  - Use two methods with different orders over the same interval
  - Use two step sizes over the same interval
- The first method allows you to use two integrators with different order on the same step, and you estimate the error by

$$E_{local} \approx \left| y_n(high) - y_n(low) \right|$$

- Usually, the low order method will have less accuracy than the high order method

High / low order step control

- The basic idea is to solve the same problem using a high order and a low order method to estimate the error
- The most common example combines a 4th and a 5th order Runge-Kutta routine (known as Runge-Kutta Fehlberg):
  - Make a trial calculation using the 4th order method
  - Make a trial calculation using the 5th order method
  - If the difference is small, take a step using the 5th order trial result
  - Based on the error between the two methods and a prescribed tolerance, set the step-size for the next step:

$$h_{new} = h_{old} \left| \frac{\delta_{target}}{\delta_{measured}} \right|^{1/5}$$

  - If the difference between the two methods is large, repeat the steps with a smaller step-size

Error estimation

- In some cases, you can't use a high order method for a step
- But you can always take two sets of steps over the same interval. The first step can use a step-size *h*, and the second a step-size *h/2*

- Again, we form the approximate error by considering the difference of the two methods

$$E_{local} \approx \left| y_n(h) - y_n(h/2) \right|$$

- The assumption we are making is that the small step size is a better estimate than the big step size. If the method converges, this is very likely

Using error estimates

- After you have calculated the local error, you then use it to make decisions about the local step. You need to set an error tolerance for the integrators

- The algorithm works something like this:
  - Calculate the local error
  - If the local error is larger than the error tolerance, decrease the step size and repeat the step
  - If the local error is MUCH smaller than the error tolerance, accept the step, but increase the step size for the next iteration
  - If the local error is acceptable, then accept the step and continue

Using error estimates

- You always use the highest order method for the actual step since it is more accurate than the lower order method
- Variable step sizes will not work well with all methods
- In predictor-corrector methods, for example, it is difficult to change the step size since they rely on the information from the previous steps

Global conservation measures

- The idea is to find global quantities to monitor during the integration which should be conserved. In some simulations, total energy, total angular momentum, total mass, etc. can be tracked and checked for conservation. If these quantities vary much from their initial values, something went wrong

- In reality, global conservation measures really aren’t the best way to control the error in ODE integration. However, they provide an independent check of the accuracy of the results

- We could look at (say) the energy change at each time-step and see if it goes beyond a preset limit, and adjust the step size accordingly.

Local step size criteria

- One of the most common ways to set the step size is to use local step size criteria. Basically, we find some local stability criteria in the equations which should not be violated, and then set the step-size to be based on this value

- The best examples of this are in fluid mechanics, where the step size is driven by the Courant-Frederichs-Levy (CFL) condition. The assumption is that information should propagate between cells at a rate which does not exceed the time the fastest fluid element can cross the smallest grid cell

- Similar criteria are used in other fields

PARTICLE METHODS

Particle systems

- Particle simulations are common in many fields of computational sciences
- Many continuous problems can be re-cast as particle systems
- Many problems can be thought of as particle systems (e.g. visualization / computer graphics – smoke, fire, …)
- Pros: particles are easier to handle than meshes
- Cons: usually need many particles, boundaries are difficult

Governing equations

- System of coupled ODE's given by Newton's second Law:

$$m_i \frac{d\mathbf{v}_i}{dt} = \mathbf{f}_i$$

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i$$

$$\mathbf{x}_i : \text{position vector}$$

$$\mathbf{v}_i : \text{velocity vector}$$

$$\mathbf{f}_i : \text{force vector}$$

$$m_i : \text{mass}$$

- The force vector is the sum of the forces exerted by all other particles and external forces

Particle forces

Different types of forces can be applied to the particles:

- Forces from an external field
  - Particles traveling through an electro-magnetic field (Lorentz forces)
  - Particles traveling through a gravitational field
  - Particles moving with a given velocity field (streamlines)
- Forces from other particles
  - Charged particles
  - Gravitating particles
  - Collisions
- Forces from the domain boundaries
  - Contact forces

Example force fields

$$\text{Gravitational field}: \mathbf{F} = m\,\mathbf{g}$$

$$\text{Lorentz force}: \mathbf{F} = q\left[\mathbf{E} + \mathbf{v} \times \mathbf{B}\right]$$

Example particle-particle forces

- Classical (Newton's) gravitation:

$$\mathbf{f}_{ij} = G\, m_i\, m_j \frac{(\mathbf{x}_i - \mathbf{x}_j)}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3}$$

- Electrostatic (Coulomb) forces:

$$\mathbf{f}_{ij} = \frac{q_i\, q_j}{4\pi\varepsilon_0} \frac{(\mathbf{x}_i - \mathbf{x}_j)}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3}$$

- Molecular dynamics (Lennard-Jones potential):

$$V(r_{ij}) = \varepsilon\left[\left(\frac{r_0}{r_{ij}}\right)^{12} - 2\left(\frac{r_0}{r_{ij}}\right)^{6}\right] \quad \Rightarrow \quad \mathbf{f}_{ij} = -\nabla V(r_{ij})$$

Basic methods

- Particle-Particle Method (PP)
- Particle-Mesh Method (PM)
- Particle-Particle Particle-Mesh Method (P^3M)

Particle-Particle Method

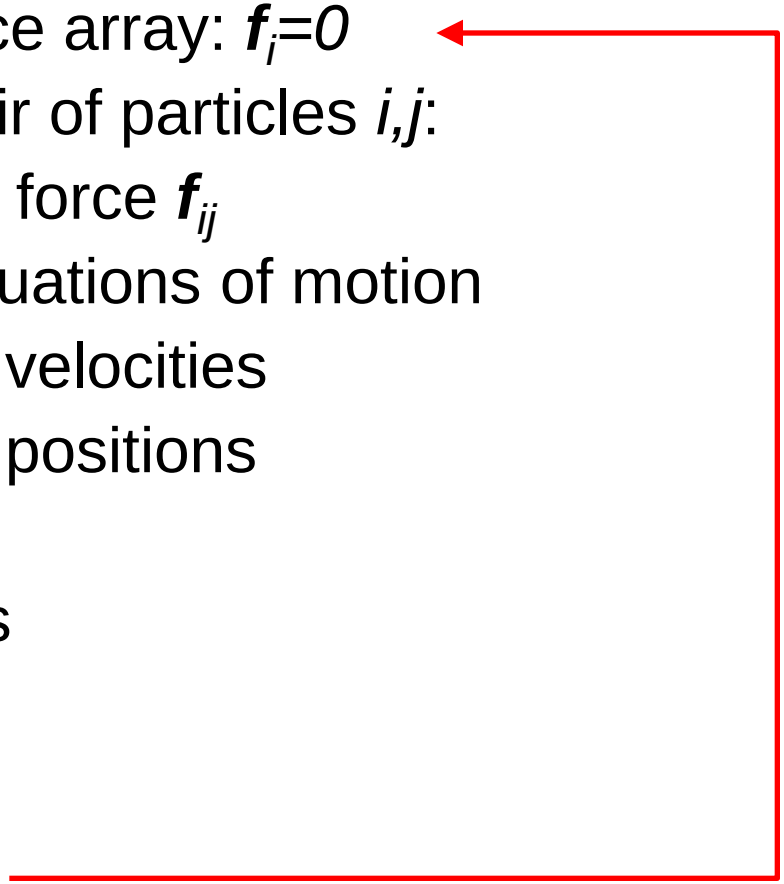
- Simplest method to advance a particle system

Basic idea:

- Compute total force on each particle as sum of forces exerted by all other particles
- Advance particle velocities using Newton’s second law
- Advance particle positions from current velocities

PP basic loop

- Initialize force array: $\boldsymbol{f}_i = 0$
- For each pair of particles i,j:
  - Compute force $\boldsymbol{f}_{ij}$
- Integrate equations of motion
  - Advance velocities
  - Advance positions
- Update
  - Velocities
  - Positions
  - Time
- Loop back



Code structure

```
t=0.0; init(x,v);                                          //  initialization
for(k=0; k<Ntime; k++) {                                   // time loop
   for(i=0;i<3*Npart;i++) {xi[i]=x[i]; vi[i]=v[i];}        // init stage temp arrays
   for(i=0; i<Nstage; i++) {                               // loop over RK stages
      alpha=dt/(Nstage-1+i);                               // RK factor
      calcForces(x,f);                                     // calc RHS (forces)
      for(j=0; j<Npart; j++) {                             // loop over particles
         xi[3*j]=x[3*j]+alpha*v[j*3];  …                   // advance positions
         vi[3*j]=v[3*j]+alpha*f[j*3];   …                  // advance velocities
      }
   }
   for(i=0; i<3*Npart; i++) {x[i]=xi[i]; v[i]=vi[i];}      // update veloc & pos
   t+=dt;                                                  // update time
}
output(x,v,t);
```

Force calculation

```
calcForces(x,f) {
   for(i=0; i<3*Npart; i++) f[i]=0.0;              // init force array =0
     for(i=0; i<Npart; i++) {                      // loop over particles i
        for(j=0; j<Npart; j++) {                   // loop over particles j
           if( j==i ) continue;                    // skip j=i
           dx=x[i*3   ]-x[j*3    ];               // distance in x
           dy=x[i*3+1]-x[j*3+1];                  // distance in y
           dz=x[i*3+2]-x[j*3+2];                  // distance in z
           r2=dx*dx+dy*dy+dz*dz; r=sqrt(r2);  // distance^2 and distance
           f[i*3]+=k/r2 * dx/r;  …                 // forces in x y and z
        }
     }
  }
}
```

=> Loop: Npart * Npart

Operation count

- For one time step and N_p particles:
  - Force calculation: $O(N_p^2)$
  - Update: $O(N_p)$
- => FLOPS $\approx 10\ N_p^2$

(FLOPS: floating point operations per second)

N_p	FLOPS/timestep
10^2	10^5
10^3	10^7
10^4	10^9
10^5	10^{11}
10^6	10^{13}
10^7	10^{15}

➢ To do one timestep per second with 10^6 particles, we would need a 10 Tflops machine

PP Improvements

- Symmetric force calculations
- Short range interactions
- Long range interactions

Symmetric force calculations

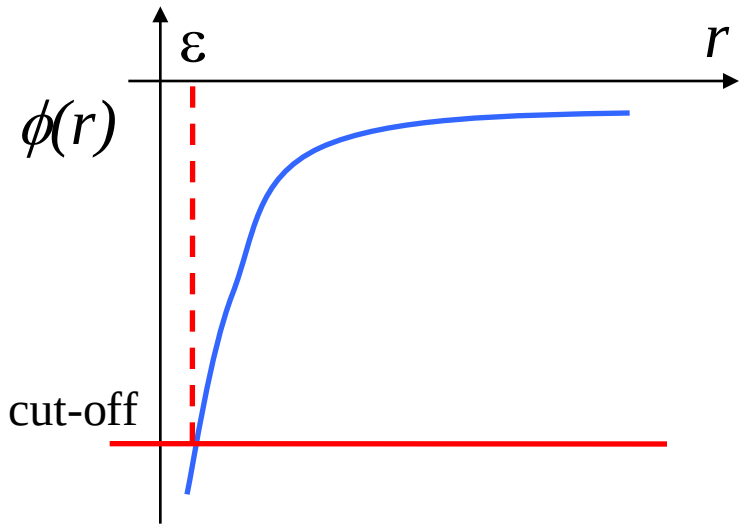
- Newton's 3rd law: $f_{ij} = -f_{ji}$
- Can cut the inner loop in the force calculation by 2

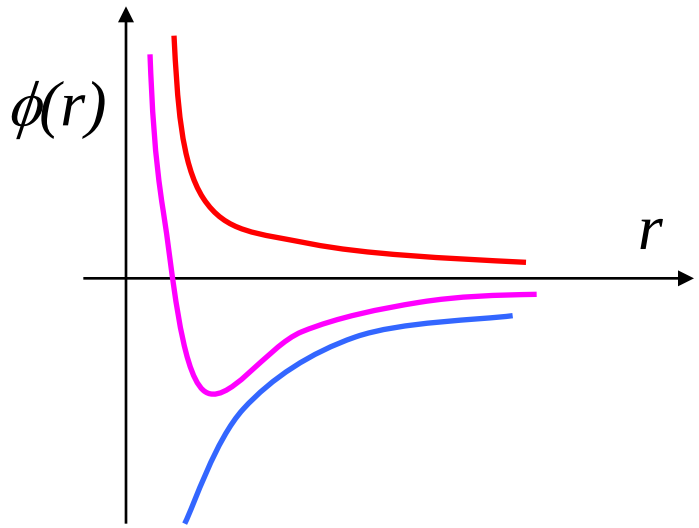
```
for(i=0; i<Npart-1; i++) {
   for(j=i+1; j<Npart; j++) {
      compute f_ij
      add to i and subtract from j
   }
}
```

=> Loop: Npart * (Npart -1) / 2

Avoiding force divergences

- Several force potentials diverge at *r=0* (when two particles become too close) causing numerical instabilities
- Option 1: use adaptive time-steps
- Option 2: add a force cut-off for $r<\varepsilon$
  (neglect very short range force effects)
- Option 3: add repulsive term to model particle collisions





Adaptive time-steps

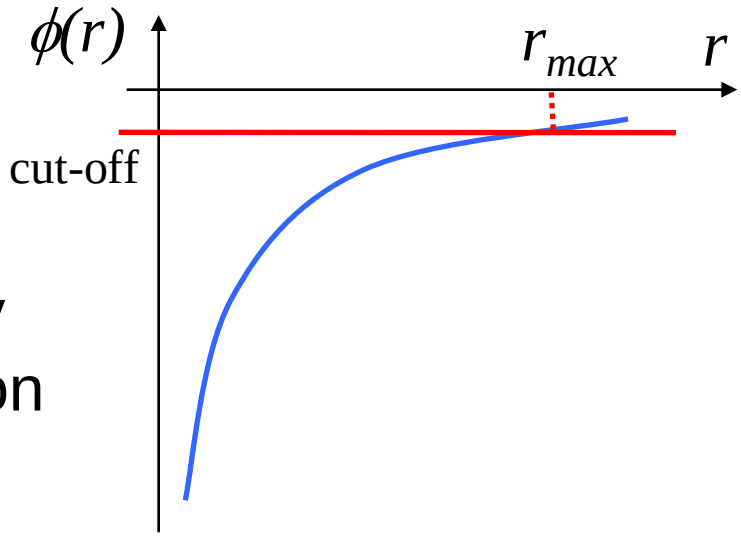
- Use adaptive time-steps to increase temporal resolution when particle accelerations are large

- High-low order step control
  - Use a high and a low order method to advance each step
  - Estimate local error & adapt time step

- Step doubling
  - Take each step twice
  - Estimate local error & adapt time step

Force cut-offs and fast searching

- Neglect very long range force effects: add a force cut-off by making force=0 for $r>r_{max}$
- Then consider only neighboring particles with $r>r_{max}$
- Use appropriate data-structures for fast searching of the closest particles to a given particle:
  - Octrees / Quadtrees
  - Bins



- Clustering of particles is still a problem since there may be many particles inside the force interaction range

Multipole expansions

- Consider a localized distribution of charge that is described by the charge density $\rho(x)$ which is non-vanishing only inside a sphere of radius R
- The potential outside the sphere can be written as an expansion in spherical harmonics:

$$\phi(x) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \frac{4\pi}{2l+1} q_{lm} \frac{Y_{lm}(\theta, \vartheta)}{r^{l+1}}$$

- The coefficients can be found from the Poisson integral (Green's theorem for the Poisson integral):

$$\phi(x) = \int \frac{\rho(x')}{|x - x'|} d^3x' \Rightarrow q_{lm} = \int Y_{lm}^{*}(\theta', \vartheta') r'^{l} \rho(x') d^3x'$$

Multipole expansion

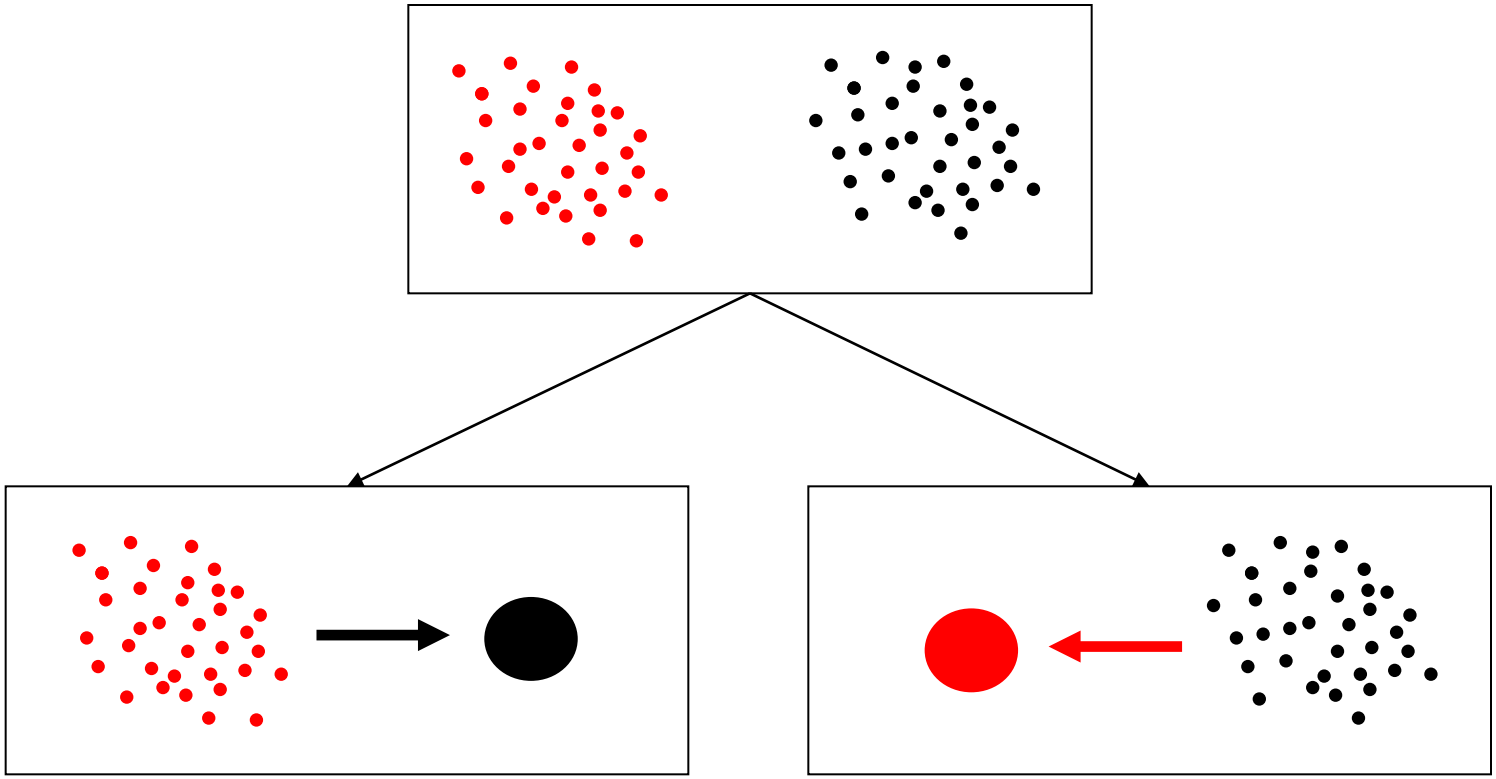
- The q_{lm} coefficients are called the *multipole moments*
- q_{00} is the *monopole* moment
- q_{11} and q_{10} are the *dipole* moments
- q_{22}, q_{21} and q_{20} are the *quadrupole* moments
- …

- There are formulas for computing the monopole, dipole and quadrupole terms from the given density distribution

- Multipole expansions are useful for approximating the potential of a localized charge distribution (by truncating the expansion for example at the quadrupole terms)

Multipole expansion

- Situations where a multipole expansion may pay off include interacting clusters of particles
- Examples: interacting galaxies / stellar clusters



Particle-Mesh Method

- The basic idea is to compute the forces on particles from an energy potential function evaluated on a grid
- The basis of this approach is that there are fast solvers for obtaining the solution of the governing equation for the potential (i.e. faster than N^2)
- This method is faster but the force evaluation is less accurate

Force calculations from potentials

$$\text{Force:}\quad \mathbf{f}_{ij} = \alpha\, p_i\, p_j \frac{(\mathbf{x}_i - \mathbf{x}_j)}{\left|\mathbf{x}_i - \mathbf{x}_j\right|^3}$$

$$\text{Can be computed as:}\quad \mathbf{f}_{ij} = -p_i \nabla \phi_{ij}$$

$$\text{Potential:}\quad \phi_{ij} = \alpha\, p_j \frac{1}{\left|\mathbf{x}_i - \mathbf{x}_j\right|}$$

Potential fields

Forces and potentials are fields:

force on particle i: $\mathbf{f}_i = \mathbf{f}(\mathbf{x}_i)$

potential of particle i: $\phi_i = \phi(\mathbf{x}_i)$

Potential field equation:

$$\nabla^2 \phi = -\beta\rho$$

Examples potentials

Gravitation:

$$p_i = m_i, \quad \alpha = G, \quad \beta = 4\pi G$$

Electrostatics:

$$p_i = q_i, \quad \alpha = \frac{1}{4\pi\varepsilon_0}, \quad \beta = \frac{1}{\varepsilon_0}$$

Fast Field Solvers Using Fourier Transforms

- The Fourier Transform of a function *h(t)* is defined as

$$H(\omega) = \int_{-\infty}^{\infty} h(t) \exp(i\omega t)\, dt$$

- And the inverse Fourier Transform is

$$h(t) = \int_{-\infty}^{\infty} H(\omega) \exp(-i\omega t)\, d\omega$$

where $i = \sqrt{-1}$ $\exp(i\omega t) = \cos(\omega t) + i \sin(\omega t)$

and ω is the angular frequency

- Fourier Transforms move the function from the time domain to the frequency domain

FFT's

- There are numerous algorithms available to compute the Fourier Transforms in *N log N* time
- These are called Fast Fourier Transforms or FFT
- See for example Numerical Recipes

Fourier Convolution Theorem

- Given two functions *f(t)* and *g(t-τ)*, their convolution is defined as:

$$f \otimes g = \int_{-\infty}^{\infty} f(t)\, g(t-\tau)\, dt$$

- For a continuous Fourier Transform, it can be shown that

$$FT(f \otimes g) = FT(f)\, FT(g)$$

and for a discretely binned set of functions

or

$$(f \otimes g)_i = \sum_{j=1}^{n} f(i)\, g(j-i)$$

$$FFT(f \otimes g) = FFT(f)\, FFT(g)$$

since the FFT's each take $O(N \log_2 N)$ calculations, the convolution using an FFT wing BIG over brute force convolution

Field Solver

$$\text{Poisson equation}: \ \nabla^2 \phi = \rho$$

$$\text{Poisson integral}: \ \phi(x) = \int G(x - x')\,\rho(x')\,d^3x'$$

$$\text{Green's function}: \ G(x - x') = \frac{1}{\left| x - x' \right|}$$

$$\Rightarrow \quad \phi(x) = G(x) \otimes \rho(x)$$

$$G(x): \text{potential of a unit charge at the origin}$$

Field Solution Using FFT's

Potential : $\phi(x) = G(x) \otimes \rho(x)$

Fourier transform : $F[\phi(x)] = F[G(x) \otimes \rho(x)]$

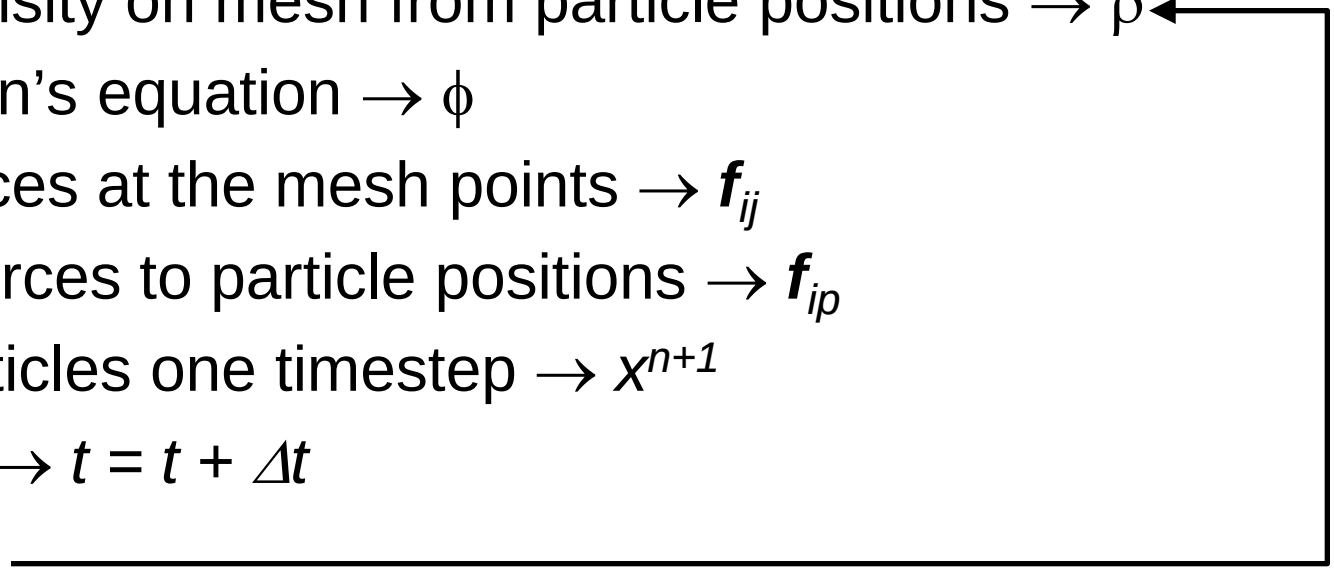
Convolution theorem : $F[\phi(x)] = F[G(x)] \cdot F[\rho(x)]$

Solution : $\phi(x) = F^{-1}\{F[G(x)] \cdot F[\rho(x)]\}$

This solution procedure is ~ *O(N log N)*

PM basic loop

- Compute density on mesh from particle positions $\rightarrow \rho$
- Solve Poisson's equation $\rightarrow \phi$
- Compute forces at the mesh points $\rightarrow \boldsymbol{f}_{ij}$
- Interpolate forces to particle positions $\rightarrow \boldsymbol{f}_{ip}$
- Advance particles one timestep $\rightarrow x^{n+1}$
- Update time $\rightarrow t = t + \Delta t$
- Loop back



Operation count

- For one time step with N_p particles and N_m mesh points:
  - Density calculation ~ $O(N_p)$
  - Potential calculation ~ $O(N_m \log N_m)$
  - Force calculation ~ $O(N_p)$
  - Update ~ $O(N_p)$
- => FLOPS $\approx 20\, N_p + 5\, N_m \log_2 N_m$

N_p	N_m	FLOPS / Δt
10^2	8^3	2.5 x 10^4
10^3	8^3	4.3 x 10^4
10^4	16^3	4.4 x 10^5
10^5	16^3	2.2 x 10^6
10^6	64^3	4.4 x 10^7
10^7	128^3	4.2 x 10^8

- To do one timestep per second with 10^6 particles, we would need a 40 Mflops machine (workstation)

Properties

- Gain in speed at the cost of resolution
- Potential and force of a single particle poorly represented for distance < mesh spacing *h*
- Resolution depends not only on the grid size but also on the choice of interpolating functions

Charge and force evaluations

Forces are computed by:

1. Accumulating densities (particles $\rightarrow$ mesh)
2. Solving a Poisson equation
3. Interpolating forces to particles (mesh $\rightarrow$ particles)

Accurate forces => make transfers as accurate as possible

Notation

h : cell / element size of mesh [length]

x_p : location of particle p

x_i : location of grid point i

$x = x_p - x_i$

$\xi = x / h$

Nearest Grid Point (NGP) Method

- Affects only closest mesh points

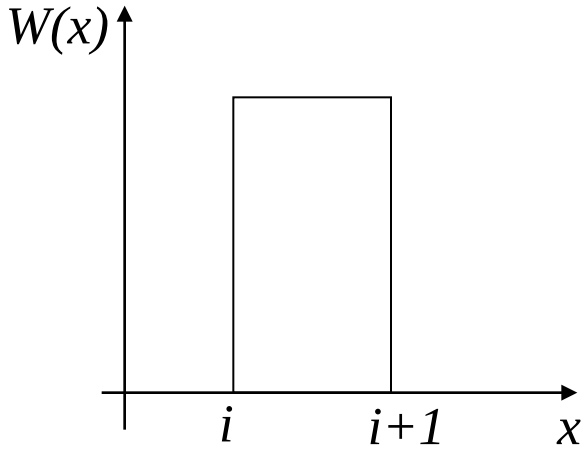
  $\Rightarrow 1^{d}$ points

- Zeroth order

- Interpolating function in 1D :

$$W(x) = \begin{cases} 1 & if \quad |\xi| < 1 \\ 0 & if \quad |\xi| \geq 1 \end{cases}$$

- Main drawback: "jumpy"



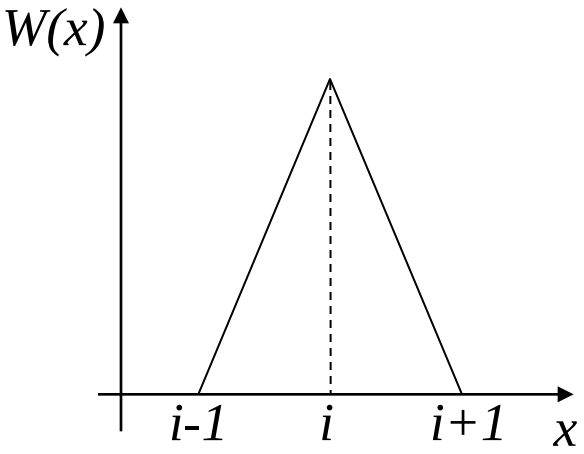
Cloud in Cell (CIC) Method

- Affects only 'host' cell or element

$$\Rightarrow 2^{d} \text{ points}$$

- First order
- Interpolating function in 1D :

$$W(x) = \begin{cases} 1 - |\xi| & if \quad |\xi| < 1 \\ 0 & if \quad |\xi| \geq 1 \end{cases}$$



Triangular Shaped Cloud (TSC) Method

- Affects 'host' cell and first surrounding layer

  $\Rightarrow 3^d$ points

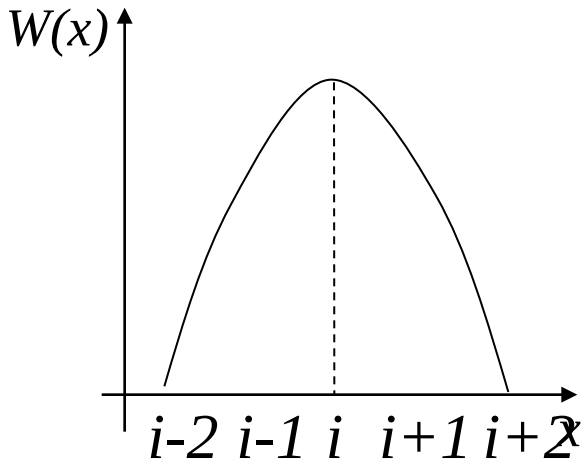
- Second order

- Interpolating function in 1D :

$$W(x) = \begin{cases} 3/4 - \xi^2 & if \quad |\xi| < 1/2 \\ 1/2(3/2 - |\xi|)^2 & if \quad 1/2 \le |\xi| \le 3/2 \\ 0 & if \quad |\xi| > 3/2 \end{cases}$$

- Main drawback: computationally expensive

  (27 mesh points per particle in 3D)



Particle-Particle Particle-Mesh Method

Basic idea:

- Split forces into short and long range contributions
- Compute short range forces from close particles (PP)
- Compute long range forces from potential (PM)

Advantages:

- Avoids the global N_p^2 complexity
- Retains local accuracy

Difficulty:

- Need to properly define the cut-off distance

Long range force (PM)

- The total “mesh” force on a particle is computed as in the PM method:

1. Assign charges to the potential mesh
2. Solve for the potential
3. Difference potentials to find mesh fields
4. Interpolate to find forces on particles

Short range force (PP)

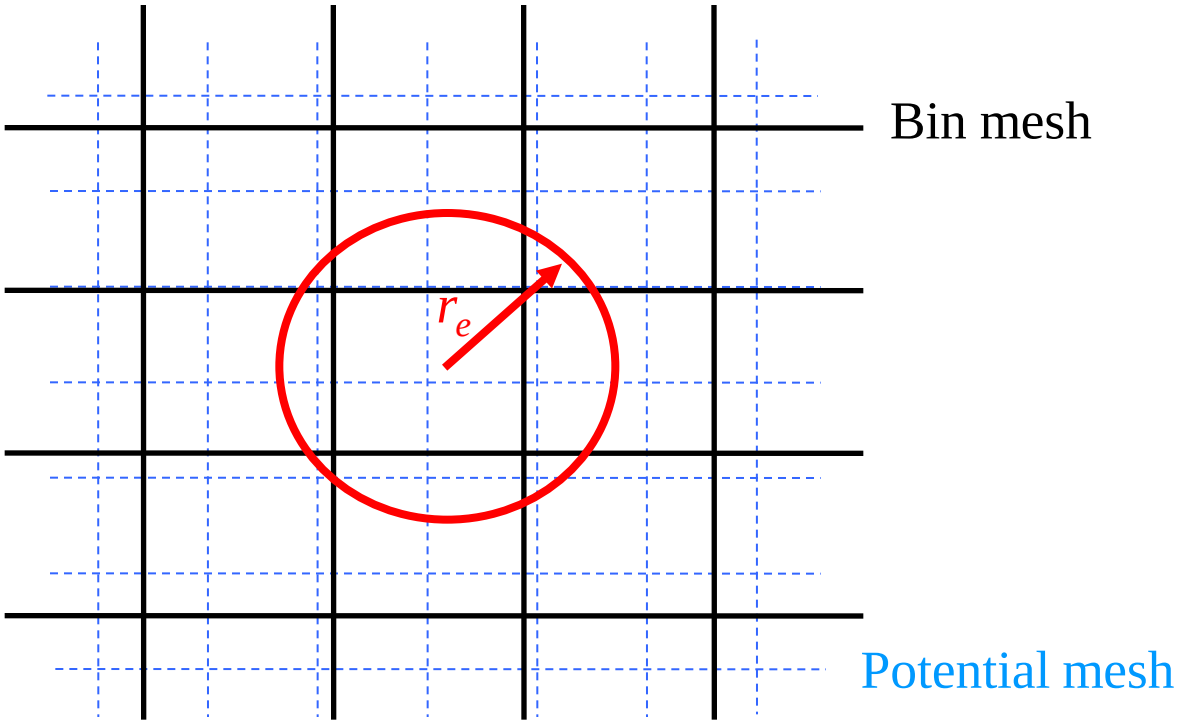
- The short range part of the force on a particle is computed as the sum of the inter-particle force with a cut-off:

$$f_i = \sum_{j=1}^{Np} f_{ij}^{sr}$$

$$f_{ij}^{sr} = \begin{cases} f_{ij} & r_{ij} < r_e \\ 0 & r_{ij} \geq r_2 \end{cases}$$

Short range force calculation

- To avoid looping over *Np* particles a bin data structure is used



- Use linked list to store particles in each bin
- For each particle, search for particles inside r_e by looking at the bins that contains the particle and its neighbors

Optimizations

- To avoid duplicate calculations of force pairs, loop over the bins and compute the force between particle pairs in this bin and with particles in neighboring bins, then add the contributions to both particles

- Considerable savings are made in the force calculation by tabulating the values of the force at uniform intervals of r^2

- Then for each force pair calculation the force r^2 is computed and the value of the force is interpolated from the table instead of computing the square root

Operation count

- For one time step with N_p particles and N_m mesh points:
  - PM force calculation ~ O($N_p + N_m \log_2 N_m$)
  - PP force calculation ~ O($(r_e/h)^3 * N_p^2 / N_m$)
- => FLOPS $\approx \alpha N_p + \beta N_m \log_2 N_m + \gamma (r_e/h)^3 * N_p^2 / N_m$

- Increased cost from PM method but still avoids the N^2 problem

➢ To do one timestep per second with 10^6 particles, we would need a supercomputer

Smoothed Particle Hydrodynamics

- SPH was invented to deal with problems in astrophysics involving fluid masses moving arbitrarily in three dimensions, in the absence of boundaries

- The fundamental idea of SPH is to use an interpolation method which allows any function to be expressed in terms of its values at a set of disordered points (particles)

- So, SPH is a particle method to approximate the solution of the equations of motion of continuous fluids

Interpolation

- The integral interpolant of any function *A(r)* is defined as:

$$A(r) = \int A(r')W(r - r', h)\, d^3 r$$

where the integral is over the entire space and *W* is the interpolating kernel, which has the following properties:

$$\int W(r - r')\, d^3 r = 1$$

$$\lim_{h \to 0} W(r - r', h) = \delta(r - r')$$

h defines the smoothing length of the kernel *W*

Numerical Interpolation

- For numerical work the integral interpolant is approximated by a summation interpolant:

$$A(r) = \sum_i m_i \frac{A_i}{\rho_i} W(r - r_i, h)$$

  where the particle *i* has mas m_i, position r_i, density ρ_i and velocity v_i. The value of any quantity *A* at the position of particle *i* is A_i

- The essential point is that function derivatives can be obtained from the kernel derivatives

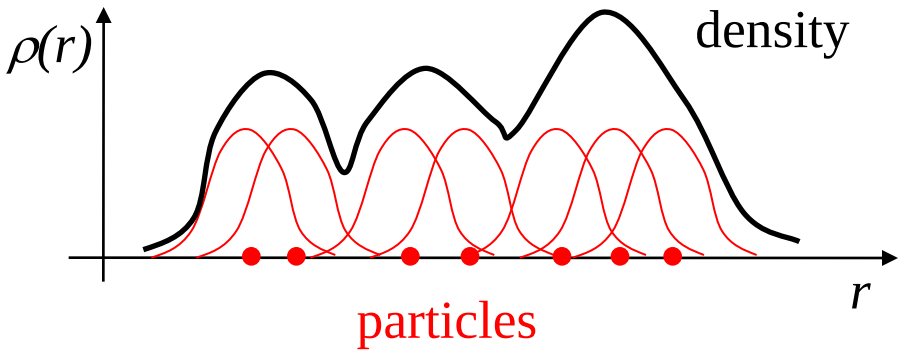
$$\nabla A(r) = \sum_i m_i \frac{A_i}{\rho_i} \nabla W(r - r_i, h)$$

Density

- For example, the density can be estimated everywhere as

$$\rho(r) = \sum_i m_i W(r - r_i, h)$$

- This can be interpreted as the smoothing of the particle's point mass by the kernel so as to obtain a continuous density field from a set of particles



Mass conservation equation

$$\text{Continuity equation}: \frac{d\rho}{dt} + \rho \nabla \cdot v = 0$$

$$\text{Vector identity}: \quad \nabla \cdot (\rho v) = \rho \nabla \cdot v + v \cdot \nabla \rho$$

$$\text{Interpolant}: \quad \nabla \cdot v = \sum_i m_i \, v_i \cdot \nabla W(r - r_i, h)$$

$$\Rightarrow \quad \rho_i (\nabla \cdot v)_i = \sum_j m_j \, (v_j - v_i) \cdot \nabla_i W(r_i - r_j, h)$$

$$\Rightarrow \quad \frac{d\rho_i}{dt} = \sum_j m_j \, v_{ij} \cdot \nabla_i W_{ij}$$

Momentum conservation equation

$$\text{Momentum equation}: \ \rho \frac{dv}{dt} = -\nabla P + F$$

$$\text{Interpolant}: \qquad \rho_i \nabla P_i = \sum_j m_j (P_j - P_i) \nabla_i W_{ij}$$

$$\text{Vector identity}: \qquad \frac{\nabla P}{\rho} = \nabla \left(\frac{P}{\rho} \right) + \left(\frac{P}{\rho^2} \right) \nabla \rho$$

$$\Rightarrow \quad \frac{dv_i}{dt} = -\sum_j m_j \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2} \right) \nabla_i W_{ij} + F_i$$

$$\text{Equation of motion}: \ \frac{dr_i}{dt} = v_i$$

Thermal energy equation

Energy equation (1st law of thermodynamics): $\frac{dE}{dt} = -\left(\frac{P}{\rho}\right)\nabla \cdot v$

Similar derivations: $\frac{dE_i}{dt} = \frac{1}{2}\sum_j m_j \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2}\right) v_{ij} \cdot \nabla_i W_{ij}$

Equation of state: $P = f(\rho, E)$

Polytropic gas: $P = \rho^{\gamma}$

Ideal gas: $P = \frac{\rho KT}{\mu}$ $E = \frac{3}{2} NKT$

More physics

Extra terms have been devised to include different effects into the previous equations:

- External pressures
- Viscosity
- Thermal conduction
- Gravitational forces
- Magnetic fields (MHD)
- Special relativity

Kernels

- Several kernels have been devised:
- Gaussian kernels
- Splines

- Design criteria:
- Smoothness
- Compact support

Applications

- SPH has been successfully applied to a variety of problems:
- Shocks and shock tubes
- Blast and wave phenomena
- Interaction / collision of starts
- Meteor / comet impacts
- High speed impacts of metals
- Binary star formation
- Structure formation in the universe
- Supernova explosions
- Nuclear collisions (special relativity)
- Neutron stars
- Plasma physics

Initialization

- Initialization of particle simulations must be done carefully
- The best way to initialize particle codes is from given density and velocity distributions
- Density distributions are usually given in the form of a probability function of space, such as a uniform distribution, a Gaussian distribution, etc
- Velocity distributions can be given in the form of a velocity probability distribution, such as the Maxwell-Boltzmann distribution (Gaussian velocity distribution); or from an initial velocity field, such as a differential rotation, etc.
- Random numbers with a non-uniform deviate are used to generate the initial conditions from given probability distributions

HIGH PERFORMANCE COMPUTING

Why Supercomputers ?

We want to:

- Run many problems in a timely manner (e.g. optimization)
- Run more complex problems (e.g. multi-scale models)
- Run larger problems (e.g. more resolution)
- Run for longer times (e.g. atmospheric dispersion models)

Major issues:

- Speed
- Memory
- Storage

Computer Hardware

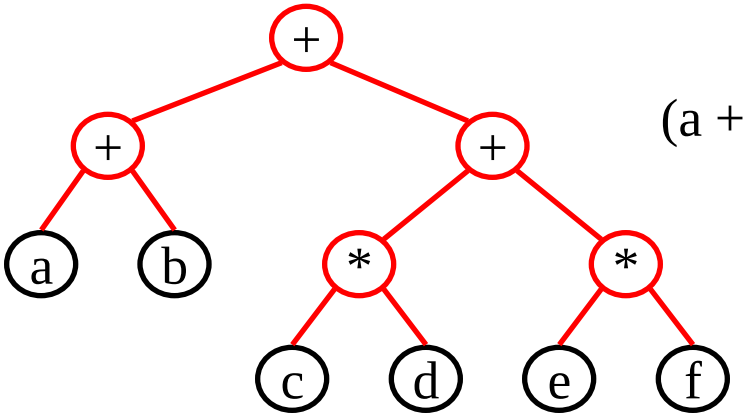
- Workstations → PC's , SGI workstations
  - Cache
  - Small RAM
  - Graphics
  - Single or dual processors
- Vector Machines → Cray , NEC, …
  - Degradation for scalar operations
  - Need appropriate coding
  - No graphics
- Parallel Machines → SGI Altix/Origin , Clusters, …
  - Cache
  - Degradation for scalar operations
  - Message passing programming
  - Data distribution

Forms of Parallelism

- Multiple functional units
- Pipelining
- Vector processors
- Multiprocessor systems
- Distributed systems

Multiple Functional Units

- This is one of the earliest forms of parallelism
- It consists of multiplying the number of functional units such as adders and multipliers
- The detection of parallelism is done at compile time with a dependence analysis tree

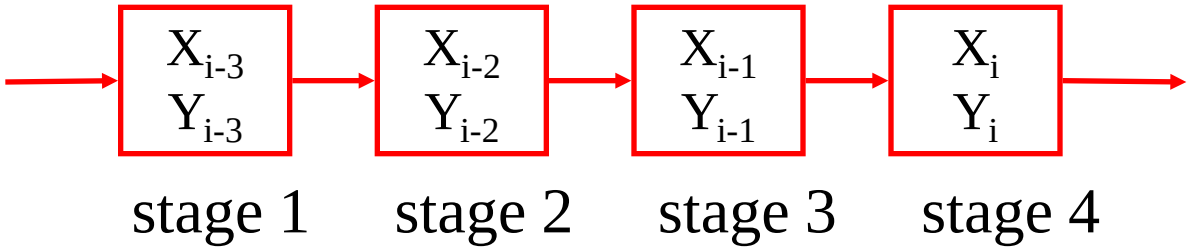


(a + b) + (c * d + d * e)

- Example of dependence analysis for arithmetic expression and parallel processing of operations

Pipelining

- The concept of pipeline is that of an assembly line
- Assume that an operation takes *s* stages to complete
- Then the operands can be passed through the *s* stages instead of waiting for all stages to be completed for the first two operands



- If each stage takes a time *t* to complete, then an operation with *n* numbers will take the time $s*t + (n-1)*t=(n+s-1)*t$
- The *speedup* would be the ratio of the time to complete the *s* stages in a nonpipelined unit: $S=ns/(n+s-1)$ $(S \approx s \text{ for } n>>1)$
- Examples: Cray

Vector Processors

- Vector computers are equipped with pipelined functional units such as pipelined floating point adders and multipliers
- In addition they incorporate vector instructions explicitly as part of their instructions sets. For example:
  - vload: load a vector from memory to a vector register
  - vadd: add the content of two vector registers
  - vmul: multiply the content of two vector registers
- Similarly to multiple functional units for scalar machines, vector pipelines can be duplicated into *multiple vector pipelines*
- Examples: NEC, Fujitsu

Multiprocessor Systems

- A *multiprocessor system* is a computer, or a set of computers, consisting of several processing elements, each consisting of a CPU, a memory and an I/O system
- The processing elements are interconnected with a bus or network
- Examples: Dual Processor PC’s, IBM SP3

Distributed Systems

- *Distributed computing* is a more general form of multiprocessing in which the processors are actually computers linked by a local area network
- Distributed systems can be *homogeneous* (same computer in each node) or *heterogeneous* (linking different types of computers)
- *Computer clusters* are usually homogeneous systems with a fast network connection between them
- *Computer farms* are usually heterogeneous systems connected with a slower network

Parallel Processing Models

- Vector pipeline model
- Shared memory model
- Single instruction multiple data model or data parallel models
- Distributed memory message passing model

Vector pipeline model

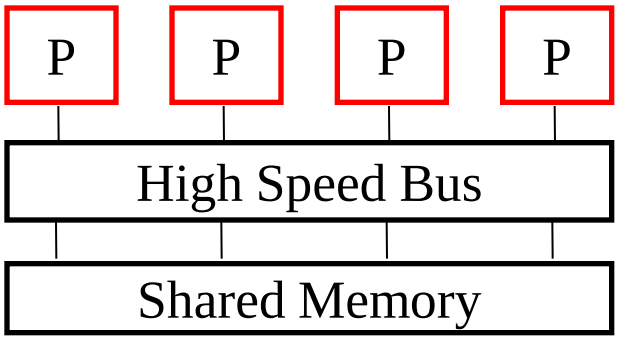
- Vector computers have vector pipeline processors connected to a large global memory
- The parallelization is done at the level of the arithmetic operations
- Useful when we need to do many times the same operation on an array of data (vector operation)
- Model: subdivide the arithmetic operations into different stages and perform every stage on a different entry

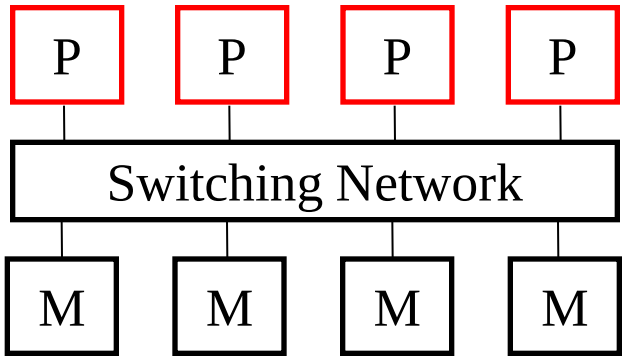
Shared memory model

- Shared memory computers have the processors connected to a large global memory with the same global view
- The parallelization is done at the level of the loops
- The advantage of this model is that the transparent data access greatly simplifies the programming

- There are two types of shared memory machines:

1. Bus-based architectures
2. Switch-based architectures





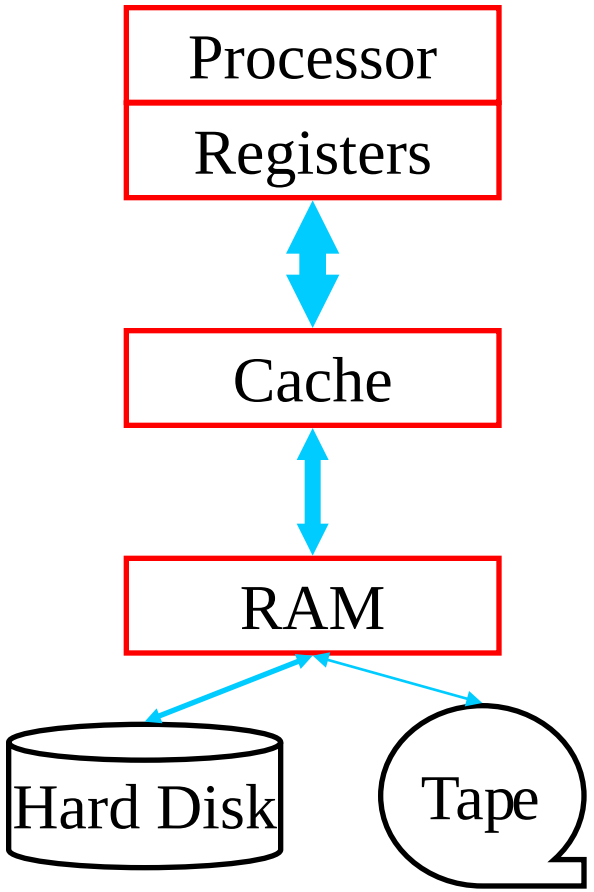
High Performance Programming

- Coding for scalar computers
- Coding for vector computers
- Coding for shared memory computers
- Coding for distributed memory computers

Memory architecture

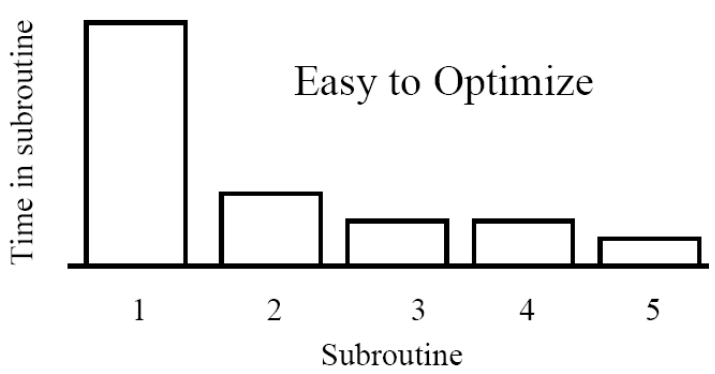
Modern computers have a hierarchical memory system:

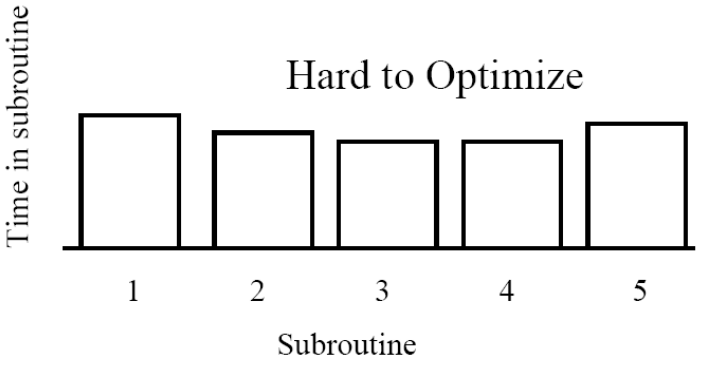
- Backup system: tapes / DVD / CD-ROM / floppy
- Hard disk / virtual or swap memory
- RAM
- Cache
- Registers on the processor



Timing and profiling

- Code optimization goes hand in hand with code profiling
- You should always optimize the portions of your code that take the longest time to execute (bottlenecks)





- You can use the Unix “time” command to get the total execution time of your code
- You can access the system time in your code to time individual routines
- Or you can use profiling programs to time your code modules

Simple Code Optimization

- Reducing operation counts
- Reducing indirect addressing
- Reducing cache misses

Example: PP code

```
for i=1,n
   for j=1,n
      if( i==j ) continue
      if( sqrt((x(i,1)-x(j,1))**2+(x(i,2)-x(j,2))**2)<1.0e-1 ) {
         f(i,1)=f(i,1)+1/4/3.14/e0*q(i)*q(j)*(x(j,1)-x(i,1))/
                        (sqrt(((x(i,1)-x(j,1))**2+(x(i,2)-x(j,2))**2))**3)
         f(i,2)=f(i,2)+1/4/3.14/e0*q(i)*q(j)*(x(j,2)-x(i,2))/
                        (sqrt(((x(i,1)-x(j,1))**2+(x(i,2)-x(j,2))**2))**3)
       }
    }
}
```

$$f_{ij} = \begin{cases} \dfrac{q_i\, q_j}{4\pi\varepsilon_0} \dfrac{(x_j - x_i)}{\left|x_j - x_i\right|^3} & \left|x_j - x_i\right| < r_{\max} \\ 0 & otherwise \end{cases}$$

Optimization #1

Don’t use Matlab !

- Using the Matlab profiler we find that one force calculation for 200 particles takes about 55.56 seconds
- Assuming the code scales as N^2, it will take $(10^6/200)^2$ *55.56 seconds to do a single calculation for a million particles
- This means we need about 44 years per timestep, or about 44,000 years for a 1000 timestep run
- Using F90 on the same machine (SGI O2) one time step with 200 particles took 0.026 seconds (2000 faster than Matlab)
- With this improvement we move down from 44 years per timestep to only 7.5 days per timestep ! And the entire run would take about 2 years

Reducing Operation Counts

- The simplest code optimization is to try to reduce the number of operations performed to a minimum, taking into account that some operations are more expensive than others
- Some operations from more to less expensive:
  - function calls
  - memory allocation (malloc)
  - if statements
  - trigonometric functions (sin, cos, exp)
  - sqrt, ** or ^ or pow
  - division (/)
  - multiplication (*)
  - addition (+, -)
  - bit operations (&, <, >, ~, ^)

Integer operations are faster than floating point operations

Example: PP code

```
rmax=1.0e-1;  rmax2=rmax*rmax
const=1.0/(4.0*e0)
for i=1,n-1
   for j=i+1,n
      dx=x(j,1)-x(i,1);   dy=x(j,2)-x(i,2);
      r2=dx*dx+dy*dy
      if( r2<rmax2 ) {
         tmp=const*q(i)*q(j)/(r2*sqrt(r))
         f(i,1)=f(i,1)+tmp*dx;  f(j,1)=f(j,1)-tmp*dx;
         f(i,2)=f(i,2)+tmp*dy;  f(j,2)=f(j,2)-tmp*dy;
       }
   }
}
```

Reducing Indirect Addressing

- Fetching variable values from memory by following a pointer or array index takes longer than accessing the value stored in a variable directly (indirect addressing)
- => the objective is to minimize the memory access through indirect addressing

Example: PP code

```
rmax=1.0e-1;  rmax2=rmax*rmax
const=1.0/(4.0*e0)
for i=1,n-1
   xi=x(i,1); yi=x(i,2); qi=q(i);
   for j=i+1,n
      dx=x(j,1)-xi;   dy=x(j,2)-yi;
      r2=dx*dx+dy*dy
      if( r2<rmax2 ) {
         tmp=const*qi*q(j)/(r2*sqrt(r))
         tmpx=tmp*dx;  tmpy=tmp*dy
         f(i,1)=f(i,1)+tmpx;  f(j,1)=f(j,1)-tmpx
         f(i,2)=f(i,2)+tmpy;  f(j,2)=f(j,2)-tmpy
      }
   }
}
```

Reducing Page and Cache Misses

- Whenever you request a value that is not in the RAM, you get a *page fault* (the value is fetched from swap memory – i.e. the hard drive). The access time is then 100 times slower
- Page faults can only be avoided by allocating less memory and using only the memory that is actually needed

- Whenever you request a value that is not in the cache, the processor loads it from RAM and you get a *cache miss*. The access time is 10 or more times slower
- When an array element is loaded from RAM to the cache, the processor actually loads a chunk of the array. Therefore, performance can be improved by processing that are close in memory in order to avoid cache misses

Reducing Cache Misses

The main strategies for reducing cache misses include:

- Proper memory allocation (by chunks)
- Proper array access in loops
- Renumbering of array elements (sort the array elements in the processing order)

Example: PP code

```
rmax=1.0e-1;  rmax2=rmax*rmax
const=1.0/(4.0*e0)
for i=1,n-1
   xi=x(1,i); yi=x(2,i); qi=q(i);
   for j=i+1,n
      dx=x(1,j)-xi;   dy=x(2,j)-yi;
      r2=dx*dx+dy*dy
      if( r2<rmax2 ) {
          tmp=const*qi*q(j)/(r2*sqrt(r))
          tmp=const*qi*q(j)/(r2*sqrt(r))
          tmpx=tmp*dx;  tmpy=tmp*dy
          f(1,i)=f(1,i)+tmpx;  f(1,j)=f(1,j)-tmpx
          f(2,i)=f(2,i)+tmpy;  f(2,j)=f(2,j)-tmpy
      }
   }
}
```

Other examples

Recursive function calls:

```
double fact(double a) {
  return (a==1)?1:fact(a-1);
}

double fact(double a)
{
   double r=1;
   for(i=2; i<=a; i++) r=r*i;
   return r;
}
```

Memory allocation:

```
while( !EOF ) {read(x,y,z);
     listadd(x,y,z);}

listadd(x,y,z) {
   newpt[num++]=malloc(3);
   …
}

listadd(x,y,z) {
   if(num+1>=nalloc)
     pts=realloc(3*chunk)
   …
}
```

Compiler optimizations

Some of these optimizations are already performed by compilers (when invoked with the –O –O2 or –O3 flags), but usually what a compiler can do are simple optimizations:

- Removal of inaccessible code
- Removal of code that produces unused results
- Simplification of constants
- Constant folding (un-redefined variables)
- Common sub-expression elimination
- Mathematical simplifications
- Removal of loop invariant code
- Simplification of inductive code

Vectorization

- The idea of pipelining is to divide the operations into stages and perform each stage on different entries simultaneously

- Example: adding two arrays a(i)=b(i)+c(i) i=1,n
- Stages:
  - Fetch b(i)
  - Fetch c(i)
  - Compare exponents
  - Align mantissas
  - Add
  - Normalize
  - Store a(i)
- In reality there are more stages (~16), therefore gains of approximately 1:16 can be obtained

Vectorization

Vectorization pays off if:

- Vectors are sufficiently long
  (at least n>16, but the longer the better)
- Loops can be vectorized:
  - No recurrence
  - No complex if statements or code branching
  - Orderly memory access

- Your program must be carefully coded in order for the compilers to properly vectorize operations

Example

Original code:

```
do i=1,1000
   do j=1,3
      a(i,j)=(a(i,j)+b(i,j))*a(i,j)+b(i,j)
   enddo
enddo
```

Improved code (longer inner loop => larger vector processing):

```
do j=1,3
   do i=1,1000
      a(i,j)=(a(i,j)+b(i,j))*a(i,j)+b(i,j)
   enddo
enddo
```

Another example

- Force calculation for a spring system

```
do iedge=1,nedge
  a =edpo(1,iedge)
  b =edpo(2,iedge)
  fx=k*( x(1,j)-x(1,j) )
  fy=k*( x(2,j)-x(2,j) )
  f(1,i)=f(1,i)+fx
  f(2,i)=f(2,i)+fy
  f(1,j)=f(1,j)-fx
  f(2,j)=f(2,j)-fy
enddo
```

$$\mathbf{f}_{ij} = k * (\mathbf{x}_j - \mathbf{x}_i)$$

$$\mathbf{f}_{ji} = -\mathbf{f}_{ij}$$

Problem: possible access of same point by different edges
=> *memory contention*

Avoiding memory contentions

- Memory contentions can be usually avoided by processing groups of disconnected elements (in this case edges)
- Idea: in each group no point is accessed more than once
- *Coloring algorithms* are use to create these groups
- The code in our example would then look like this:

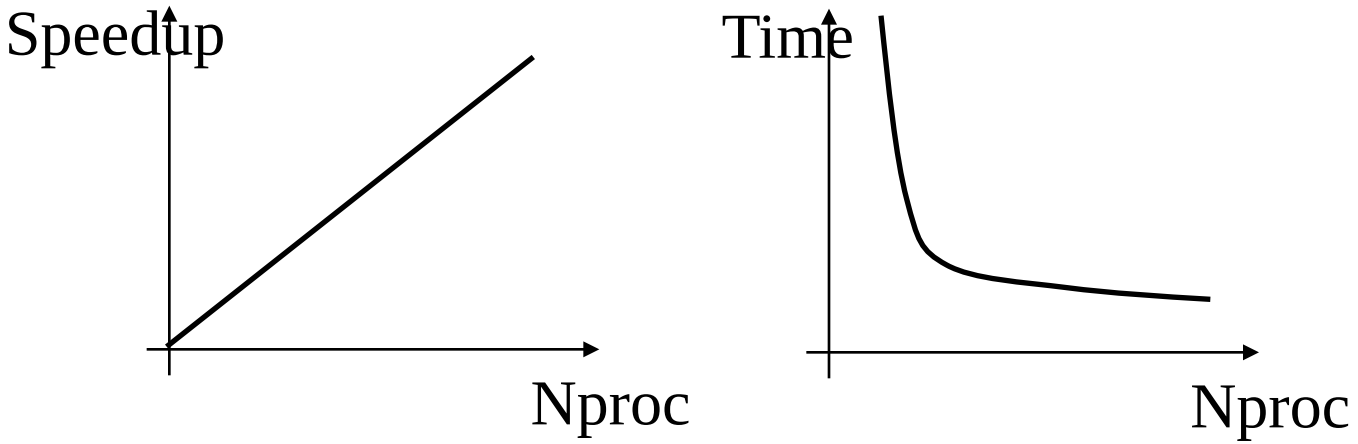
```
do igroup=1,ngroup
   istart=group(1,igroup)   ! first edge of this group
   iend =group(2,igroup)    ! last edge of this group
c$dir ivdep                 ! compiler directive: ignore vector dependencies
   do iedge=istart,iend
      …
   enddo
enddo
```

Parallel computing

- Basics
- Shared memory: processes and threads
- Distributed memory: message passing

Speedup

- The speedup can be computed by dividing the time required to run on 1 processor to the time it takes to run on N processors



- In some cases, due to data locality and the memory cache, *superlinear* speedups can be achieved (e.g. running on 2 processors you get a speedup of 2.2)

Amdahl's law

- Amdahl's law states that the speedup of a code can be computed as

$$S = \frac{1}{\alpha + (1 - \alpha) / p}$$

- α is the portion of the code that cannot be parallelized
- *p* is the number of processors

- This implies that the speedup is limited by the slowest (serial) portion of the code
- If 1% of your code cannot be parallelized, does it make sense to run on 100 processors ? And 1,000 ? And 10,000 ?

Code scalability

The performance and scalability of a parallel code will be determined by the following factors:

- Load balancing
- Communication speed (system bandwidth)
- Computation / communication ratio (problem size and data locality)
- I/O

Coding for shared memory machines

- Parallel programming on shared memory computers can follow two basic strategies:

1. By creating independent *processes*
2. By creating independent *threads*

- Threads share the same memory space, therefore there is no need for explicit communications

- Processes have their own memory space and must communicate to exchange data

Parallel Processes

- Parallel processes can be created with the fork() command in linux
- This command creates a new process (child) with an exact copy of the memory of the calling (parent) process
- The function fork() returns the process id (pid) of the child to the parent process and 0 to the child process
- The code execution continues on both processes with the next line below the fork() call

```
pid=fork()
if( pid==0 )  print “I am the child”
else          print “I am the parent”
```

Communications

Communications between parallel Unix processes can be done in a number of ways:

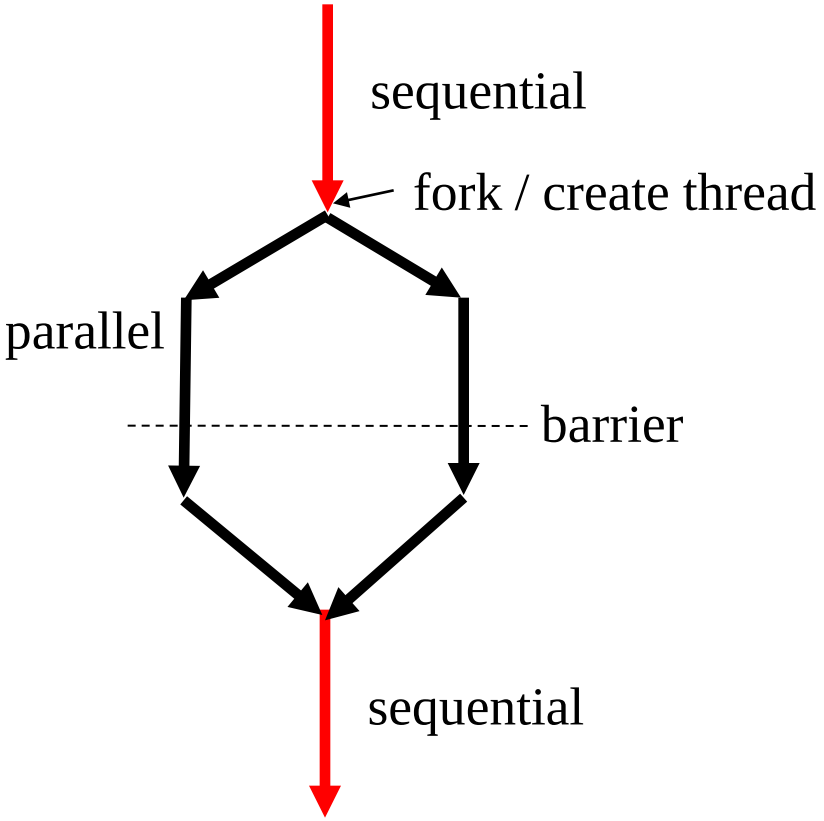
- Signals
- Shared memory + semaphores
- Pipes
- Sockets
- Files

Threads

- The creation of a parallel thread is similar to the fork() but in this case the memory space is not duplicated, it is inherited by the child thread
- The thread creation function will return the id of the child to the parent and 0 to the child
- There are different implementations of threads, the most commonly used is *pthreads*

```
if( thread_create()==0 )   for i=n/2+1,n  do_work(i)
else                       for j=1,n/2    do_work(j)
```

Parallel threads / processes



OpenMP

- OpenMP is a set of compiler directives to write parallel programs for shared memory machines
- Although it uses parallel threads, the programming is different from explicit thread programming for example with pthreads
- OpenMP is available for C/C++ and Fortran
- The number of threads used in a parallel calculation can be set via environment variables or with explicit function calls within the code

Thread management

Environment variables:

- OMP_NUM_THREADS

Functions:

- numt=omp_get_num_threads()
- myid=omp_get_thread_num()

Parallel constructs

C/C++:

```
#pragma omp parallel shared(a), private(i)
{
   i=omp_get_thread_num();
   do_work(a,i);
}
```

Fortran:

```
!$omp parallel shared(a) private(i)
    i=omp_get_thread_num()
    do_work(a,i)
$!omp end parallel
```

Parallel loops

C/C++:

```
#pragma omp parallel for shared(a,n), private(i)
{
   for(i=0; i<n; i++) a[i]=i*i;
}
```

Fortran:

```
!$omp parallel do shared(a,n) private(i)
   do i=1,n
      a(i)=i*i
   enddo
$!omp end parallel do
```

Other constructs

- master:
  specifies that a block is executed only by the master of a thread group
- critical:
  specifies that a block is executed by a single thread at a time
- barrier:
  waits until all threads have reached this point
- reduction:
  the reduction operator, a copy of temporary values are stored in each thread and at the end of the parallel construct these variables are associated with the specified operator

Caution

- Need to be careful with memory consistency

```
#pragma parallel for shared(a,n), private(i)
{
    for(i=1; i<n; i++) a[i]=a[i-1];
}
```

- Sometimes you will need to duplicate memory or change the execution logic in order to achieve the correct result in parallel

Coding for distributed memory machines

- There are two basic modes of programming for distributed memory computers:

1. A single program is loaded and executed in each node simultaneously

2. A master program is launched on one of the nodes that then launches slave programs into the other nodes

- Then the processes running on different nodes communicate with each other via *message passing*

- The two most common libraries for message passing are *PVM* (Parallel Virtual Machine) and *MPI* (Message Passing Interface)

Operations

- Process control
- Global operations
- Communications
- Group operations

Process control

- Initialize the parallel node
- Launch a code on other nodes
- Obtain task or process id
- Obtain parent task id
- Obtain number of processes
- Finalize a parallel task
- Synchronize parallel tasks

Process control

PVM

- pvm_init()
- pvm_exit()
- myid=pvm_mytid()
- parid=pvm_parent()

- pvm_barrier()

- pvm_spawn()

MPI

- MPI_Init()
- MPI_Finalize()
- MPI_COMM_RANK(…myid…)
- MPI_COMM_SIZE(…nproc…)

- MPI_Barrier()

Global operations

- Reduction operations
  - sum, min, max, product of a distributed array
- Synchronization operations
  - Cause all processes to stop until all processes have reached this point
- Broadcast operations
  - Transmit data from a node to all the other nodes

Global Operations

PVM

- pvm_reduce()
- Operations: sum, max, min, product

MPI

- MPI_Reduce()
- Operations: sum

Example: PVM

```
main() {
   mid=pvm_mytid();
   pid=pvm_parent();
   if( pid<0 ) {master=1; cid=pvm_spawn();}
   else        {master=0;}
   print “my id=” mid
   print “parent id=” pid
   if( master==1) print “master”
   else           print “slave”
  pvm_exit()
}
```

Example: MPI

```
Main() {
   MPI_init()
   MPI_Comm_size(MPI_COMM_WORLD,&np)
   MPI_Comm_rank(MPI_COMM_WORLD,&mid)
   print “my id=” mid
   print “# proc=” np
   MPI_Finalize()
}
```

Communications

- Communications can be blocking or non-blocking
- Operations include:
  - Packing information into a communication buffer
  - Sending data to another node
  - Receiving data from another node
  - Unpacking data from a communication buffer
  - Checking whether a message has arrived

Communications

PVM

- pvm_initsend()
- pvm_pack*()
- pvm_send()
- pvm_psend()
- pvm_mcast()

- pvm_recv()
- pvm_upk*()
- pvm_probe()

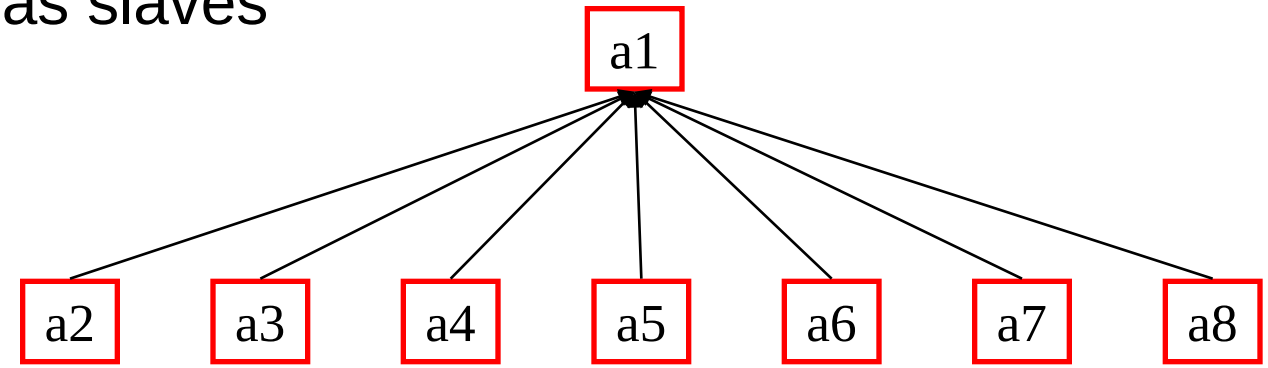
MPI

- MPI_Send()
- MPI_Isend() – non-blocking
- MPI_Bcast()

- MPI_Recv()
- MPI_Irecv() – non-blocking
- MPI_Iprobe()

Parallel reduction

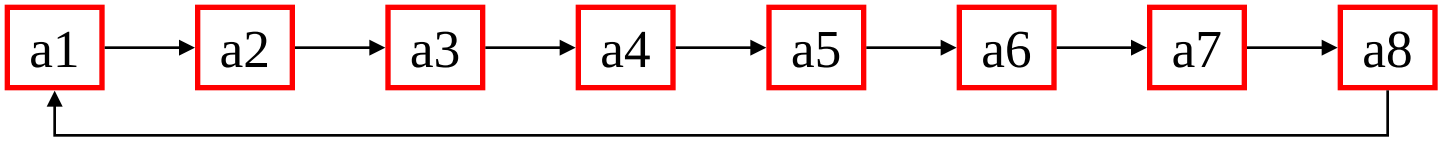
- Problem: add a distributed set of numbers sum=a1+a2+a3+a4+a5+a6+a7+a8
- Possible solution: one node acts as master and the others as slaves




```
if( i==1 ) {
   sum=a;
   for(i=2,n) {receive(a,any); sum+=a;}
   print sum
}
else {send(a.1);}
```

Parallel Reduction

- Ring communication strategy: sequential processing

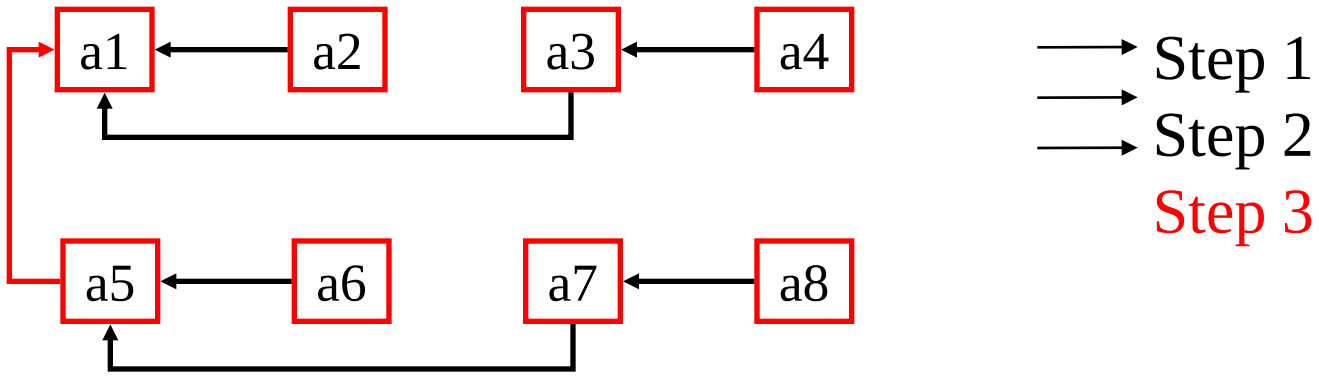



```
if( i==1 ) {send(a,i+1); receive(sum,n); print sum;}
if( i==n ) {receive(sum,i-1); sum+=a; send(sum,1);}
else       {receive(sum,i-1); sum+=a; send(sum,i+1);}
```

- Need N steps to complete the sum

Parallel reduction

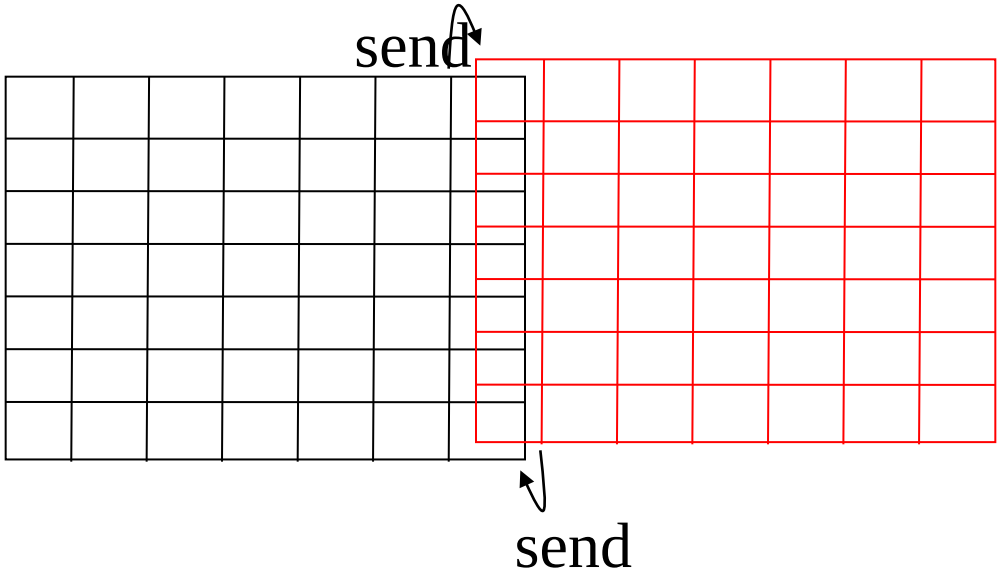
- Tree communication strategy: parallel processing



- Need $log_2 N$ steps to complete the sum

Structured Grid PDE Solution

- Suppose we are solving a PDE on a structured grid in parallel on two processors
- Split the mesh in two keeping one layer of overlapping points
- The boundary points of the white mesh are interior points for the yellow mesh and vice versa
- Need to exchange boundary values between neighboring processors before proceeding to the next timestep



Code Structure

```
loop over timesteps
   apply boundary conditions
   solve for interior points
   if( i==1 ) {send(column nx-1,i+1); receive(column nx,i+1);}
   else       {send(column 2,i-1); receive(column 1,i-1);}
end loop
```

Unstructured Grid PDE

Same idea for unstructured grids but since each sub-domain can have an arbitrary number of neighbor sub-domains, thus each processor needs:

- List of neighboring sub-domains
- List of grid points that send values to each neighbor
- List of grid points that receive values from each neighbor
- Scheduling list: order in which communications with neighbors are processed

Code Structure

```
loop over timesteps
   apply boundary conditions
   solve for interior points
   loop over list of neighbor sub-domains (j)
      jneig=lneig[j]
      send(lpsnd[jneig],jneig);
      recv(lprcv[jneig],jneig);
   end loop
end loop
```

Lneig[j]: list of neighbor sub-domains (processors)
Lpsnd[j]: list of points to send to neighbor j
Lprcv[j]: list of points to receive from neighbor j

Shared vs Distributed Memory

- Parallelizing for shared memory is easier and you always have a working code
- Parallelizing for distributed memory is harder and it doesn’t work until you have entirely parallelized your code
- For distributed memory you will probably need extra tools such as grid partitioning and parallel visualization tools
- In some cases distributed memory can yield better performance because of better data locality (the actual memory of most shared memory machines is distributed!)
- Shared memory computers are more expensive (?), clusters can be cheaper but for a fair comparison we would need to compare two systems with the same performance working for the same number of years…
- Mixed models can be also attractive: cluster of shared memory nodes (4 to 16 shared processors per node are readily available)