

## Lecture 4 – Advanced Fortran 95/03

- **There are different data types used in Fortran 95/03**
  - Some compilers will delineate REAL variables into KINDS
  - KINDS are described in Chapman
  - Most of the time, you do not have to worry about the KINDS but will instead decide in advance if you wish to work in *single* or *double precision* and then declare variables accordingly
  - REAL(kind=4) (single precision), REAL(kind=8) (double precision)
- **Most modern scientific Fortran programs use *derived data type variables* to better describe their meaning**
  - Convenient way to group together all information regarding a particular item and their relationships
  - Used extensively in object-oriented programming (OOP)

## Derived Data Types

- **Usage:**

```
TYPE :: type_name  
component definitions  
...  
END TYPE type_name
```

- **EXAMPLE:**

```
TYPE :: PERSON  
  CHARACTER (LEN=14) :: first_name  
  CHARACTER :: middle_initial  
  CHARACTER (LEN=14) :: last_name  
  CHARACTER (LEN=14) :: phone  
  INTEGER :: age  
  CHARACTER :: sex  
  CHARACTER (LEN=11) :: ssn  
END TYPE PERSON
```

## Derived Data Types

- **We could initialize variables or arrays as data types**

EXAMPLE:    TYPE (PERSON) :: JOHN, JANE

              TYPE (PERSON) :: DIMENSION(100) PEOPLE

      ....

          JOHN = PERSON('John','R','Jones','323-6439','21','M','123-45-6789')

          JANE = PERSON('Jane','C','Doe','332-3060','19','F','999-99-9999')

          PEOPLE(1) = PERSON('Sam','L','Van','666-2354','33','M','987-65-4321')

- **We can refer to individual parameters of a derived data type (such as first\_name, age, or ssn in the above example) by simply referring to type%parameter**

EXAMPLE:               JOHN%age = 21

                        JANE%last\_name = 'Doe'

- **You can now begin to see how all of the parameters (or characteristics) of a variable can be linked together by using derived data types**
  - Working with the parameters of derived data types is easy but working with the data types as a whole is not!

## Derived Data Types in Modules

- **Derived data types can be defined in Modules and then “used” in different routines**
  - Input into Modules is the same as previously described
- **You can see how derived data types might be very useful in**
  - Data bases
  - Data structures used for scientific simulations
    - Unstructured-grids for instance:

```
TYPE GRID_CELL
  REAL :: cell_number
  CHARACTER(LEN=4) :: cell_type
  REAL :: number_of_faces
END TYPE GRID
```
  - Object oriented programming (OOP)

## Dynamic Memory Allocation

- **One reason that the C computer language became so popular was its ability to dimension arrays dynamically.**
  - This eliminates the need to re-compile codes in order to handle different problem sizes
- **Fortran did not allow for dynamic memory allocation until the 90/95 versions (several years after C was able to)**
- **An array can be declared with variable dimensions through use of the ALLOCATABLE statement**

`REAL, ALLOCATABLE, DIMENSION (:,:) :: ARRAY1`

This declares ARRAY1 with a variable dimension to be defined with an ALLOCATE statement

## ALLOCATE and DEALLOCATE

- Once an array is declared **ALLOCATABLE**, the dimensions can be defined and the array created

```
EXAMPLE: INTEGER, PARAMETER :: IDIM=200
          INTEGER, PARAMETER :: JDIM=100
          REAL, ALLOCATABLE DIMENSION (:,:) :: ARRAY1
          ...
          ALLOCATE(ARRAY1(IDIM,JDIM),STAT=STATUS)
          ...use in computation
          DEALLOCATE(ARRAY1,STAT=STATUS)
```

Where STATUS = 0 for successful allocation and a positive number if the allocation fails

## Checking Allocated Arrays

- **Arrays can not be re-allocated once they have been allocated. They must first be deallocated and the allocated with new parameters.**
- **A way to check to see if an array has be allocated is to use the logical intrinsic function *ALLOCATED***

EXAMPLE: REAL, ALLOCATEABLE, DIMENSION (:): INPUT

...

IF(ALLOCATED(INPUT)) THEN

    READ(8,\*) INPUT

ELSE

    WRITE(\*,\*) 'Warning: INPUT not allocated'

END IF

## Pointers

- **Pointers are primarily used when variables and arrays are created and destroyed dynamically and we do not know in advance the number of each that will be needed**
- **A pointer is a variable that contains the address in memory of another variable where data is stored**
  - Pointer → address of variable
  - Variable → data value
- **Pointers are useful in GPU computing (discussed later in the course)**



## Pointers

- **A Fortran variable is declared to be a pointer by including the `POINTER` attribute in its declaration statement**

EXAMPLE:            `REAL, POINTER :: P1`

- **A pointer is only allowed to point to variables of its declared type**
- **Pointers may also be used for derived data types**

EXAMPLE: `TYPE(VECTOR), POINTER :: VECTOR_P1`

- **Pointers may also point to arrays**

EXAMPLE: `INTEGER, DIMENSION(:), POINTER :: P1`

`REAL, DIMENSION(:,,:), POINTER :: P2`

## Targets

- A pointer can point to any variable or array of the pointer's type as long as the variable has been declared to be a *target*
- A *target* is a data object whose address has been made available for use with pointers
- A Fortran variable or array may be declared to be a target by including the TARGET attribute in its declaration

EXAMPLE: REAL, TARGET :: A1 = 7

INTEGER, DIMENSION(10), TARGET :: INT\_ARRAY

## Pointers → Targets

- **Pointers may be associated with a given target by using a *pointer assignment statement***

EXAMPLE:            `pointer => target`

This stores the memory address of target in the pointer

- **We can also assign the value of one pointer to another pointer by using the assignment statement**

EXAMPLE:            `pointer2 => pointer`

- **Even though the address is stored in a pointer, reference to the pointer (say with a read or write statement) will result in the variable value that it is pointing to**

## Disassociation and Status of Pointers

- **Pointers can be disassociated from their targets by using the *nullify* statement**

EXAMPLE:            NULLIFY(pointer1, pointer2,...)

- **The status of the association of a pointer can be determined using the *associated* statement**

EXAMPLE:            status = ASSOCIATED(pointer1)

- where status is TRUE if the pointer is associated with any target

or                    status = ASSOCIATED(pointer1,target1)

- where status is TRUE if the pointer is associated with the particular target included in the ASSOCIATED statement

## Pointers and Arrays

- **Pointers can also point to a targeted array. The pointer must declare the type and the rank of the array that it will point to:**

EXAMPLE:           REAL, DIMENSION(50,50), TARGET :: MYDATA  
                      REAL, DIMENSION(:, :), POINTER :: POINTER1  
                      POINTER1 => MYDATA

- **Pointers can also point to a subset of an array**

EXAMPLE:           REAL, DIMENSION(50,50), TARGET :: MYDATA  
                      REAL, DIMENSION(:, :), POINTER :: POINTER1  
                      POINTER1 => MYDATA(1:20,10:50)

## Dynamic Memory Allocation of Array Pointers

- **A powerful feature of pointers is that they can be used to dynamically create variables or arrays (similar to what we did before but without the need for **ALLOCATABLE** statements)**

EXAMPLE:

```
PROGRAM MEM
IMPLICIT NONE
INTEGER :: I, ISTAT
INTEGER, DIMENSION(:) POINTER :: PTR1,PTR2
ALLOCATE(PTR1(1:10),STAT=ISTAT)
ALLOCATE(PTR2(1:10),STAT=ISTAT)
! NOTE THAT WE DID NOT HAVE TO DECLARE PTR1 OR PTR2
! AS ALLOCATABLE ARRAYS
PTR1 = (/ I,I=1,10) /)
PTR2 = (/ I,I=11,20) /)
```

## Pointers with Derived Data Types

- **Another powerful use of pointers is with components of derived data types.**

- Pointers in derived data types may even point to the derived data type being defined (circular!)
- This feature allows us to construct types of dynamic data structures linked together by successive pointers during program execution. Such a structure is called a *linked list*
- A linked list is a series of variables of a derived data type with the pointer from each variable pointing to the next variable in the list

```
EXAMPLE:      TYPE :: REAL_VALUE
                REAL :: VALUE
                TYPE(REAL_VALUE), POINTER :: PNTR
            END TYPE
```

Contains a real number and a pointer to another variable of the same type

## Uses of Link Lists

- **Link lists are useful when the size of the data set is dynamic and it is unknown at the start of the program what the final size will be**
  - Grid embedding adaptation is an example
  - Convergence histories is another example
  - Results of search algorithms
- **Link lists allow us to add elements to an array, one at a time**



## Pointers in Procedures

- **Pointers can also be used as dummy arguments in procedures (subroutines) or passed as arguments to procedures. This is used in GPU computing (to be discussed).**
  - If a procedure has dummy arguments with either the POINTER or TARGET attributes, then the procedure must have an explicit interface
  - If a dummy argument is a pointer, then the actual argument passed to the procedure must be a pointer of the same type, kind, and rank
  - A pointer dummy argument cannot have an INTENT attribute
    - In Fortran03, the INTENT now refers to the pointer and not the target
- **A function result can also be a pointer.**

# Object Oriented Programming (OOP)

- **Modern programming methods include object oriented styles that include:**
  - Modular program design (intensive use of subroutines)
  - Use of re-usable programming “objects”
  - Abstract data types
  - General data structures
  - Pointers and targets
- **OOP methods have been used in the C language for several years and are now fully-functional in Fortran**
  - Object-based modular structure
  - Data abstraction
  - Automatic memory management
  - Classes
  - Inheritance
  - Polymorphism (generic functionality) and dynamic binding

# Data Types

- **We've mentioned some data types used in programming**
  - Intrinsic types: character, logical, floating point, complex, real
  - User defined data types: derived data types
- **Now let's discuss data types and some definitions in the context of OOP**
- **Abstract Data Types**
  - Coupling or encapsulation of the data with a select group of functions defining everything that can be done with the data → **abstract data type**
  - **Abstraction** (generalization):
    - pertains to only the essential features of the data
    - Features are defined in a manner that is independent of any specific programming language
    - Instances are defined by their behavior and the implementation is secondary

## Classes

- **Class:** an extension of an abstract data type by providing additional member routines to serve as constructors
- **The constructor ensures that the class is created with acceptable values assigned to all its data attributes.**
  - Example a type: 

```
type (face_information) :: face
    integer :: face_index
    integer :: face_idim, face_jdim
    real(kind=8), allocatable, dimension(:,:) :: xface,yface
end type face_information
```

when combined with a constructor using “contains” to create the information contained in the type and all included in a module, it becomes a class

## Object, Instance, Encapsulation, and Data Hiding

- **Object**: combines various classical data types into a set that defines a new variable type or structure.
- **Instance**: every object from a class, by providing the necessary data, is called an instance of the class.
- **Encapsulation**: the coupling or encapsulation of data and its functions into a unified entity. This is performed by including the class in a module.
- **Data hiding**: the protection of information in one part of a program from access and from being changed in other parts of the program. In the module, the “contains” statement couples the data, specifications, and operators before it to the functions and subroutines that follow it in the module.

## Inheritance and Polymorphism

- **Inheritance:** we can employ one or more previously defined classes (of data and functionality) to organize additional classes. Functionality programmed into the earlier classes may not need to be recoded to be usable in the later class.
  - This is performed in Fortran with “use” followed by the name of the module statement block that defined the class.
- **Polymorphism:** ability of a function to respond differently when supplied with arguments that are objects of different types

## Example

- **In this class (and likely during your careers) you will develop (or use) a differential equation solver for modeling some physical phenomena.**
  - Heat conduction, structural stress/strain, flow
- **In such solvers, there are several kernels that make up the algorithms and several algorithms that make up the code**
  - Grid cell face area
  - Grid cell volume
  - i-face, j-face, k-face fluxes
  - Integration of fluxes
  - Timestep size
  - Update of variables, etc.

## Procedural vs OOP

- Most people that learn how to program codes start off learning the **procedural programming style**
- Let's say you wanted to start programming your project-1 code to be described next.
- The pseudo-code for a procedural program to solve the heat conduction equation might look something like the following:



## Procedural Programming Example-1

program heat

! this program solves the 2D heat conduction equation on a structured grid

create global computational grid

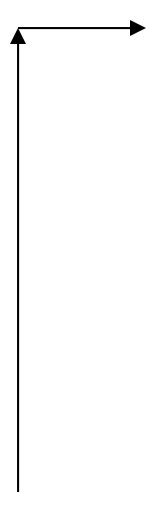
define computational and physical dimensions

allocate memory for grid and temperature variables

initialize temperature on computational grid


set the boundary conditions on temperature

solve the heat conduction equation



- calculate constant-i face primary projected lengths
- calculate constant-j face primary projected lengths
- calculate primary cell volume
- calculate primary cell-centered time-step size
- calculate constant-i face temperature fluxes
- calculate constant-j face temperature fluxes
- calculate primary cell center temperature first-derivatives

## Procedural Programming Example-1



```
calculate constant-i face secondary projected lengths
calculate constant-j face secondary projected lengths
calculate secondary cell volume
calculate secondary node-centered time-step size
calculate constant-i face temperature derivative fluxes
calculate constant-j face temperature derivative fluxes
calculate secondary node-center temperature second-derivatives
calculate right-hand side of heat conduction equation
    sum second derivatives and multiply by time-step size
    these are the changes in temperature
apply boundary conditions
update temperature and test for convergence
if (notconverged) iterate
    else write out final solution in standard (plot3d) format
end program
```

## Procedural Programming Example-1

- **Each line of the pseudo-code could be either programmed directly in the main program (that I do not recommend), or written as a subroutine that is called from the main program.**
- **When subroutines are used, you need to get information exchanged between the subroutine and the main program**
  - This can be done using arguments in the subroutine call OR a module that can be “used” in both the subroutine and the main program
  - Also note that subroutines that pertain to the calculation of geometry variables do not need to be placed inside of the iteration loop

## Procedural Programming Example-2

program heat

use variable\_module

! this program solves the 2D heat conduction equation on a structured grid

call computational grid creation

    define computational and physical dimensions

    allocate memory for grid and temperature variables and store in  
    variable\_module

call temperature initialization

call boundary condition initialization

call constant-i face primary projected lengths

call constant-j face primary projected lengths

calculate primary cell volume

calculate constant-i face secondary projected lengths

calculate constant-j face secondary projected lengths

calculate secondary cell volume

## Procedural Programming Example-2

solve the heat conduction equation

→ call primary cell-centered time-step size

call constant-i face temperature fluxes

call constant-j face temperature fluxes

call primary cell center temperature first-derivatives

call secondary node-centered time-step size

call constant-i face temperature derivative fluxes

call constant-j face temperature derivative fluxes

call secondary node-center temperature second-derivatives

call right-hand side of heat conduction equation

    sum second derivatives and multiply by time-step size

        these are the changes in temperature

call boundary conditions

call update temperature and test for convergence

if (notconverged) iterate

    else write out final solution in standard (plot3d) format

end program

## **Procedural Programming Example-2**

- **Then the subroutines are linked to the main program during the linking step, after compilation of all subroutines, in the creation of the code**
- **The subroutines can be included in the same file as the main program or in separate files**
- **Procedural programming is the most straightforward method for engineers. However, it lacks the generality (abstractions) and organization that OOP offers.**

## OOP Example-3

program heat

! this program solves the 2D heat conduction equation on a structured grid

use grid\_module

use face\_module

use cellvolume\_module

use facefluxes\_module

use secondaryface\_module

use secondarycellvolume\_module

use secondaryfacefluxes\_module

use boundarycondition\_module

create global computational grid (class)

define computational and physical dimensions

allocate memory for grid and temperature variables and store in module

call temperature initialization (subroutine)

call boundary condition initialization (subroutine)

call constant-i face primary projected lengths (class)

call constant-j face primary projected lengths (class)

## OOP Example-3

calculate primary cell volume (class)

calculate constant-i face secondary projected lengths (class)

calculate constant-j face secondary projected lengths (class)

calculate secondary cell volume (class)

solve the heat conduction equation

call primary cell-centered time-step size (subroutine)

call constant-i face temperature fluxes (class)

call constant-j face temperature fluxes (class)

call primary cell center temperature first-derivatives (class)

call secondary node-centered time-step size (class)

call constant-i face temperature derivative fluxes (class)

call constant-j face temperature derivative fluxes (class)

call secondary node-center temperature second-derivatives (class)



## OOP Example-3

call right-hand side of heat conduction equation (class)

    sum second derivatives and multiply by time-step size (subroutine)

        these are the changes in temperature

call boundary conditions (class)

call update temperature and test for convergence (subroutine)

if (notconverged) iterate

    else write out final solution in standard (plot3d) format

end program

## OOP Example-3

- Each class consists of a module of the allocatable array(s) pertaining to that particular class and **contains** the subroutine that creates the data. For instance, constant-i face lengths class would be:

```
module constant-i_face_lengths
real(kind=8), allocatable, dimension(:, :) :: dx_i, dy_i
contains
use dimensions_module !(contains imax, jmax)
do j = 1, jmax-1
  do i = 1, imax
    dx_i(i, j) = x(i, j+1) - x(i, j)
    dy_i(i, j) = y(i, j+1) - y(i, j)
  enddo
enddo
end module constant-i_face_lengths
```

## Projects

- **During the creation of your projects, you can start by using procedural programming until you are comfortable with programming techniques.**
- **However, once you become comfortable with the concepts and techniques of OOP, you can use them in your projects.**

## Homework 2 (cont)

- **Read Chapters 11-15 in Fortran95/03 (Chapman)**

## **Programming a Single-Block Simulation Code**

- **Engineers develop and use various simulation codes to perform analysis and design**
- **As the complexity of physical problems increases, the likelihood that these analyses are used on parallel computers greatly increases**
  - Knowing how to develop and/or use parallel computers to do these engineering tasks will give you skills that will enhance your career
- **One of the objectives of this course is to take you through the steps that are used to develop a parallel simulation code**
  - Develop a single-block structured simulation code
  - Develop a data-structure to support multiple blocks
  - Develop a multi-block structured simulation code
  - Develop a domain-decomposition capability
  - Develop a parallel, multi-block structured simulation code

## Programming a Simulation Code

- **Numerical engineering analyses range in complexity from**
  - simple algebraic tasks → solving a differential equation → solving multiple differential equations → solving multiple sets of differential equations for different disciplines
- **Many simple algebraic analyses, such as data reduction, are “embarrassingly parallel” in that the data of a given point does not depend on other points**
- **However, most differential equation solvers rely on finite-differencing, finite-volume, or finite-element numerical schemes that require that the data at a given point is dependent on its surrounding set of points.**
- **You will learn much more by developing a parallel simulation code for this type of problem**

## Programming a Simulation Code

- **For this class, you will be asked to program a parallel simulation code that solves the heat conduction equation.**
  - A single, second-order, partial differential equation for the thermal field of a structure
  - If you have NOT had any background/experience in numerical methods, don't worry....I will go through the Poisson/Laplace equation with you and give the basics needed to perform this task

## Programming a Simulation Code

- **Many engineering analyses require the solution of a single differential equation.**
- **A good example of this is the Poisson/Laplace equation used for:**
  - Stream-function (incompressible fluid dynamics)
  - Potential flow (incompressible, irrotational fluid dynamics)
  - Heat conduction (heat transfer)
  - Wave or Helmholtz equation (acoustics)
  - Grid generation (smoothing a computational grid)



## Example: The Stream Function For Irrotational (Inviscid) Flow

- **For irrotational flow, the curl of the velocity is zero**

$$\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = 0 \quad \text{let's define:} \quad u = \frac{\partial \psi}{\partial y} \quad \text{and} \quad v = -\frac{\partial \psi}{\partial x}$$

- **Substituting the definitions of the velocity components in terms of the stream function leads to:**

$$\nabla^2 \psi = \frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = 0$$

- **Laplace's equation with boundary conditions:**

- At infinity:  $\psi = U_{\infty} y + \text{const}$

- At solid body:  $\psi = \text{const}$

## What is the Stream Function Physically?

- Lines of constant  $\psi$  are streamlines of the flow!
- If an elemental arc-length of a 2-D streamline is to be parallel to the velocity vector, then the velocity components must be in proportion to:

$$\frac{dx}{u} = \frac{dy}{v}$$

or 
$$u \, dy - v \, dx = 0$$

- Substituting the stream function for the velocity components leads to:

$$\frac{\partial \psi}{\partial x} dx + \frac{\partial \psi}{\partial y} dy = 0 = d\psi \quad \text{or}$$

$$\psi = \text{constant along a streamline}$$

## Example: The Velocity Potential

- **We can define another function, called the velocity potential,  $\phi$ , that satisfies the irrotationality condition (instead of continuity as the stream function did).**
  - Note that the stream function only forced us to 2-dimensional problems but with no other constraints
    - (must use 2 stream functions for 3-D flow!)
  - Introduction of the velocity potential restricts us to irrotational flow but we can deal with 3-D flows directly!
    - However, there aren't many 3-D irrotational flows in reality

$$u = \frac{\partial \phi}{\partial x} \quad v = \frac{\partial \phi}{\partial y} \quad w = \frac{\partial \phi}{\partial z} \quad \text{or} \quad V = \nabla \phi$$

$$\text{curl } V = \nabla \times V = 0$$

$$\begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ \frac{\partial \phi}{\partial x} & \frac{\partial \phi}{\partial y} & \frac{\partial \phi}{\partial z} \end{vmatrix} = \left( \frac{\partial^2 \phi}{\partial y \partial z} - \frac{\partial^2 \phi}{\partial z \partial y} \right) \hat{i} + \left( \frac{\partial^2 \phi}{\partial z \partial x} - \frac{\partial^2 \phi}{\partial x \partial z} \right) \hat{j} + \left( \frac{\partial^2 \phi}{\partial x \partial y} - \frac{\partial^2 \phi}{\partial y \partial x} \right) \hat{k} = 0$$

## Velocity Potential

- **We know that the velocity potential automatically satisfies the irrotationality condition.**
- **Let's see what happens to continuity:**
  - For incompressible flow:

$$\nabla \cdot V = \nabla \cdot (\nabla \phi) = 0 \quad \text{or}$$

$$\nabla^2 \phi = 0 = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2}$$

- **Laplace's equation which is fairly easy to solve**

## Orthogonality of Streamlines and Potential Lines

- **By comparing the stream function with the velocity potential, we see**

$$u = \frac{\partial \psi}{\partial y} = \frac{\partial \phi}{\partial x}$$

$$v = -\frac{\partial \psi}{\partial x} = \frac{\partial \phi}{\partial y}$$

**or**  $d\psi = \left(\frac{\partial \psi}{\partial y} dy\right)\hat{i} + \left(\frac{\partial \psi}{\partial x} dx\right)\hat{j} = du\,dy - dv\,dx = 0$  along line of constant  $\psi$

$$d\phi = \left(\frac{\partial \phi}{\partial x} dx\right)\hat{i} + \left(\frac{\partial \phi}{\partial y} dy\right)\hat{j} = du\,dx + dv\,dy = 0 \text{ along line of constant } \phi$$

- **The condition for orthogonality between  $\psi$  and  $\phi$  is that**

$$\left(\frac{dy}{dx}\right)_{\phi=\text{constant}} = -\frac{u}{v} = -\frac{1}{(dy/dx)_{\psi=\text{constant}}}$$

## Example: Heat Conduction

- The temperature distribution through a solid is governed by the transient heat conduction equation

$$\rho c_p \frac{\partial T}{\partial t} = k \left[ \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right]$$



**With boundary conditions:**

At edges:  $T(x, y) = \text{known}$  or

$$\frac{\partial T}{\partial n}(x, y) = \text{known}$$

- For steady state:**

**Laplace's equation again**

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

## Finite Difference Representation

- One can represent derivatives of differential equations as finite-differences at some level of accuracy based upon series expansions
- Usually, we would use standard second-order accurate (error of  $O[\Delta x^3, \Delta y^3]$ ), central difference operators



$$\frac{\partial \phi}{\partial x} = \frac{\phi_{i+1,j} - \phi_{i-1,j}}{2(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})} + O[\Delta x^3] \quad \text{central}$$

$$= \frac{\phi_{i+1,j} - \phi_{i,j}}{(x_{i+1,j} - x_{i,j})} + O[\Delta x^2] \quad \text{forward}$$

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})^2} + O[\Delta x^4]$$

## Finite Difference Representation

- So we could take a Laplace equation, like the steady heat conduction equation, and write it in terms of finite-differences (for simple plates)

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

$$\left[ \frac{T_{i+1,j}^n - 2T_{i,j}^n + T_{i-1,j}^n}{(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})^2} + \frac{T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n}{(y_{i,j+\frac{1}{2}} - y_{i,j-\frac{1}{2}})^2} \right] = 0$$

We can take the  $T_{i,j}$  term and write it at the new time level giving

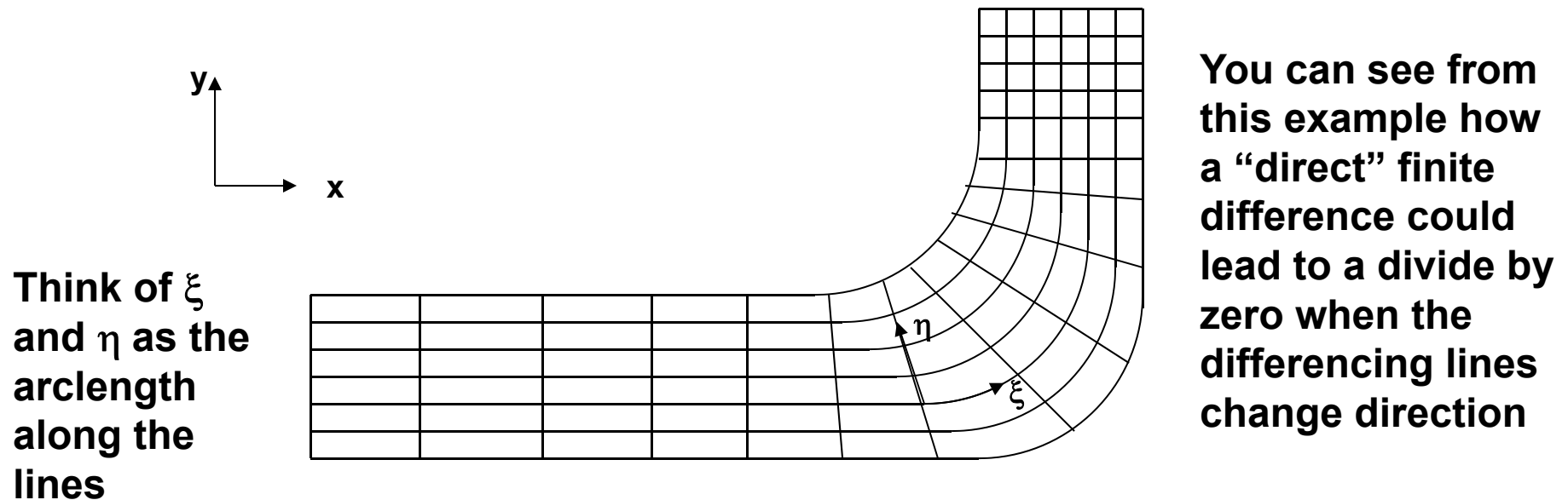
$$T_{i,j}^{n+1} = \frac{1}{2} \left[ \frac{T_{i+1,j}^n + T_{i-1,j}^n}{(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})^2} + \frac{T_{i,j+1}^n + T_{i,j-1}^n}{(y_{i,j+\frac{1}{2}} - y_{i,j-\frac{1}{2}})^2} \right] \left[ \frac{(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})^2 (y_{i,j+\frac{1}{2}} - y_{i,j-\frac{1}{2}})^2}{(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})^2 + (y_{i,j+\frac{1}{2}} - y_{i,j-\frac{1}{2}})^2} \right]$$

and we could write the RHS terms to all exist at time level  $n$  (making the scheme **EXPLICIT**). This is the point-Jacobi numerical method.



## Finite Difference Representation

- This “direct” type of finite differencing works fine if the geometry is very simple (rectangular).
- If the geometry is distorted or has large variations in angle, then the finite difference equations must be written in the curvilinear coordinate system OR the equations must be integrated and discretized using a finite-volume procedure.



## Finite Difference Representation

- **So in the curvilinear coordinate system  $(\xi, \eta)$  where  $\xi$  and  $\eta$  are functions of  $x$  and  $y$**

$$\begin{aligned}\frac{\partial}{\partial \xi} &= \frac{\partial}{\partial x} \frac{\partial x}{\partial \xi} + \frac{\partial}{\partial y} \frac{\partial y}{\partial \xi} \\ \frac{\partial}{\partial \eta} &= \frac{\partial}{\partial x} \frac{\partial x}{\partial \eta} + \frac{\partial}{\partial y} \frac{\partial y}{\partial \eta}\end{aligned}\quad \text{or} \quad \begin{bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix}$$

**Jacobian, J**

- **And we can find the inverse to the Jacobian matrix to give**

$$J^{-1} = \frac{1}{|J|} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial y}{\partial \xi} \\ -\frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \xi} \end{bmatrix} \quad |J| = \frac{\partial x}{\partial \xi} \frac{\partial y}{\partial \eta} - \frac{\partial x}{\partial \eta} \frac{\partial y}{\partial \xi}$$

## Finite Difference Representation

- So we have

$$\begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} = \frac{1}{|J|} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial y}{\partial \xi} \\ -\frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \xi} \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{bmatrix}$$

or

$$\begin{aligned} \frac{\partial}{\partial x} &= \frac{1}{|J|} \left[ \frac{\partial}{\partial \xi} \frac{\partial y}{\partial \eta} - \frac{\partial}{\partial \eta} \frac{\partial y}{\partial \xi} \right] \\ \frac{\partial}{\partial y} &= \frac{1}{|J|} \left[ -\frac{\partial}{\partial \xi} \frac{\partial x}{\partial \eta} + \frac{\partial}{\partial \eta} \frac{\partial x}{\partial \xi} \right] \end{aligned} \qquad |J| = \frac{\partial x}{\partial \xi} \frac{\partial y}{\partial \eta} - \frac{\partial x}{\partial \eta} \frac{\partial y}{\partial \xi}$$

## Finite Difference Representation

- **And for second derivatives, we can use the chain rule of differentiation to get:**

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{1}{|J|^2} \left[ (y_\eta)^2 \phi_{\xi\xi} - 2y_\xi y_\eta \phi_{\xi\eta} + (y_\xi)^2 \phi_{\eta\eta} + (y_\eta y_{\xi\eta} - y_\eta y_{\eta\eta}) \phi_\xi + (y_\xi y_{\xi\eta} - y_\eta y_{\xi\xi}) \phi_\eta \right]$$

$$\frac{\partial^2 \phi}{\partial y^2} = \frac{1}{|J|^2} \left[ (x_\eta)^2 \phi_{\xi\xi} - 2x_\xi x_\eta \phi_{\xi\eta} + (x_\xi)^2 \phi_{\eta\eta} + (x_\eta x_{\xi\eta} - x_\xi x_{\eta\eta}) \phi_\xi - (x_\xi x_{\xi\eta} - x_\eta x_{\xi\xi}) \phi_\eta \right]$$

- **You can see that the governing equation becomes much more complicated in the transformed coordinate system!**

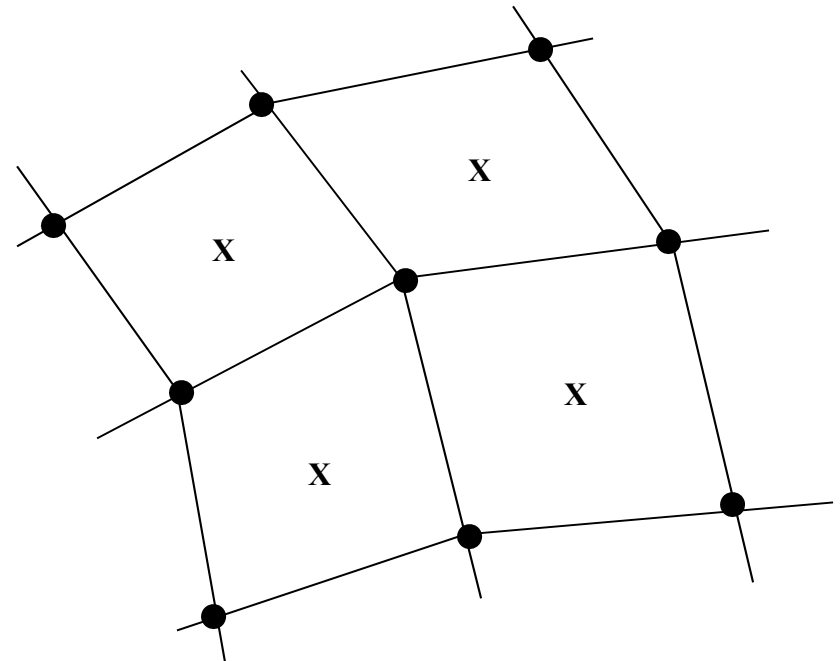
## Finite Volume Representation

- We can avoid dealing with Jacobians directly by *integrating* the governing equations instead of using finite-differences. This results in a control (finite) volume procedure.
- From Green's theorem (integrating in the counter-clockwise direction):

$$\int_{Vol} \left[ \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right] dVol = 0$$

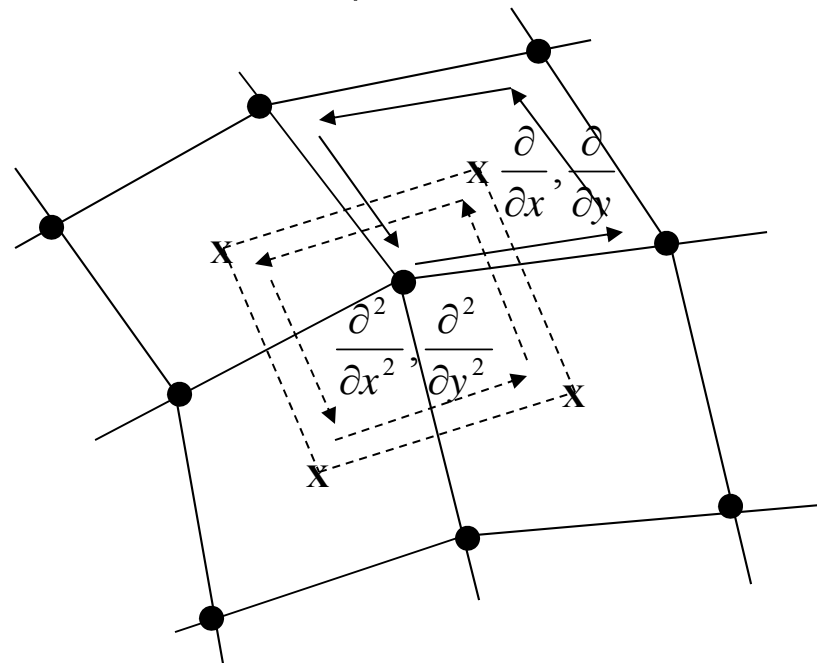
$$\frac{\partial \phi}{\partial x} = \frac{1}{Vol} \oint_{CV} \phi \, dy$$

$$\frac{\partial \phi}{\partial y} = -\frac{1}{Vol} \oint_{CV} \phi \, dx$$



## Finite Volume Representation

- **We can apply the finite volume discretization to the steady heat conduction equation by applying Green's integral twice**
  - Around the primary control volume (with the nodes as the vertices) to find the first derivative
  - Then around the secondary control volume (with the cell-centers as the vertices) to find the second derivative



## Finite Volume Representation

- So let's just look at the x-derivative terms for now

$$\left. \frac{\partial T}{\partial x} \right|_{i+\frac{1}{2}, j+\frac{1}{2}} = \frac{1}{2Vol_{i+\frac{1}{2}, j+\frac{1}{2}}} \left[ \begin{aligned} & (T_{i,j} + T_{i+1,j})(y_{i+1,j} - y_{i,j}) + (T_{i+1,j} + T_{i+1,j+1})(y_{i+1,j+1} - y_{i+1,j}) + \\ & (T_{i,j+1} + T_{i+1,j+1})(y_{i,j+1} - y_{i+1,j+1}) + (T_{i,j} + T_{i,j+1})(y_{i,j} - y_{i,j+1}) \end{aligned} \right]$$

$$\left. \frac{\partial T}{\partial x} \right|_{i-\frac{1}{2}, j+\frac{1}{2}} = \frac{1}{2Vol_{i-\frac{1}{2}, j+\frac{1}{2}}} \left[ \begin{aligned} & (T_{i-1,j} + T_{i,j})(y_{i,j} - y_{i-1,j}) + (T_{i,j} + T_{i,j+1})(y_{i,j+1} - y_{i,j}) + \\ & (T_{i-1,j+1} + T_{i,j+1})(y_{i-1,j+1} - y_{i,j+1}) + (T_{i-1,j} + T_{i-1,j+1})(y_{i-1,j} - y_{i-1,j+1}) \end{aligned} \right]$$

$$\left. \frac{\partial T}{\partial x} \right|_{i-\frac{1}{2}, j-\frac{1}{2}} = \frac{1}{2Vol_{i-\frac{1}{2}, j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i-1,j-1} + T_{i,j-1})(y_{i,j-1} - y_{i-1,j-1}) + (T_{i,j-1} + T_{i,j})(y_{i,j} - y_{i,j-1}) + \\ & (T_{i-1,j} + T_{i,j})(y_{i-1,j} - y_{i,j}) + (T_{i-1,j-1} + T_{i-1,j})(y_{i-1,j-1} - y_{i-1,j}) \end{aligned} \right]$$

$$\left. \frac{\partial T}{\partial x} \right|_{i+\frac{1}{2}, j-\frac{1}{2}} = \frac{1}{2Vol_{i+\frac{1}{2}, j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i,j-1} + T_{i+1,j-1})(y_{i+1,j-1} - y_{i,j-1}) + (T_{i+1,j-1} + T_{i+1,j})(y_{i+1,j} - y_{i+1,j-1}) + \\ & (T_{i,j} + T_{i+1,j})(y_{i,j} - y_{i+1,j}) + (T_{i,j-1} + T_{i,j})(y_{i,j-1} - y_{i,j}) \end{aligned} \right]$$

## Finite Volume Representation

- And likewise, the second derivative is:

$$\left. \frac{\partial^2 T}{\partial x^2} \right|_{i,j} = \frac{1}{Vol_{i,j}} \left[ \frac{\left( \frac{\partial T}{\partial x} \Big|_{i+\frac{1}{2},j+\frac{1}{2}} + \frac{\partial T}{\partial x} \Big|_{i-\frac{1}{2},j+\frac{1}{2}} \right)}{2} \left( y_{i+\frac{1}{2},j+\frac{1}{2}} - y_{i-\frac{1}{2},j+\frac{1}{2}} \right) + \frac{\left( \frac{\partial T}{\partial x} \Big|_{i-\frac{1}{2},j+\frac{1}{2}} + \frac{\partial T}{\partial x} \Big|_{i-\frac{1}{2},j-\frac{1}{2}} \right)}{2} \left( y_{i-\frac{1}{2},j+\frac{1}{2}} - y_{i-\frac{1}{2},j-\frac{1}{2}} \right) + \right. \\ \left. \frac{\left( \frac{\partial T}{\partial x} \Big|_{i-\frac{1}{2},j-\frac{1}{2}} + \frac{\partial T}{\partial x} \Big|_{i+\frac{1}{2},j-\frac{1}{2}} \right)}{2} \left( y_{i-\frac{1}{2},j-\frac{1}{2}} - y_{i+\frac{1}{2},j-\frac{1}{2}} \right) + \frac{\left( \frac{\partial T}{\partial x} \Big|_{i+\frac{1}{2},j-\frac{1}{2}} + \frac{\partial T}{\partial x} \Big|_{i+\frac{1}{2},j+\frac{1}{2}} \right)}{2} \left( y_{i+\frac{1}{2},j-\frac{1}{2}} - y_{i+\frac{1}{2},j+\frac{1}{2}} \right) \right]$$



## Finite Volume Representation

- If we substituted the first derivatives, we get:

$$\frac{\partial^2 T}{\partial x^2} \Big|_{i,j} = \frac{1}{2Vol_{i,j}} \left[ \begin{aligned} & \left( \frac{1}{2Vol_{i+\frac{1}{2},j+\frac{1}{2}}} \left[ \begin{aligned} & (\mathbf{T}_{i,j} + T_{i+1,j})(y_{i+1,j} - y_{i,j}) + (T_{i+1,j} + T_{i+1,j+1})(y_{i+1,j+1} - y_{i+1,j}) + \\ & (T_{i,j+1} + T_{i+1,j+1})(y_{i,j+1} - y_{i+1,j+1}) + (\mathbf{T}_{i,j} + T_{i,j+1})(y_{i,j} - y_{i,j+1}) \end{aligned} \right] + \right. \\ & \left. \frac{1}{2Vol_{i-\frac{1}{2},j+\frac{1}{2}}} \left[ \begin{aligned} & (T_{i-1,j} + \mathbf{T}_{i,j})(y_{i,j} - y_{i-1,j}) + (\mathbf{T}_{i,j} + T_{i,j+1})(y_{i,j+1} - y_{i,j}) + \\ & (T_{i-1,j+1} + T_{i,j+1})(y_{i-1,j+1} - y_{i,j+1}) + (T_{i-1,j} + T_{i-1,j+1})(y_{i-1,j} - y_{i-1,j+1}) \end{aligned} \right] \right) \left( y_{i+\frac{1}{2},j+\frac{1}{2}} - y_{i-\frac{1}{2},j+\frac{1}{2}} \right) + \\ & \left( \frac{1}{2Vol_{i-\frac{1}{2},j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i-1,j} + \mathbf{T}_{i,j})(y_{i,j} - y_{i-1,j}) + (\mathbf{T}_{i,j} + T_{i,j+1})(y_{i,j+1} - y_{i,j}) + \\ & (T_{i-1,j+1} + T_{i,j+1})(y_{i-1,j+1} - y_{i,j+1}) + (T_{i-1,j} + T_{i-1,j+1})(y_{i-1,j} - y_{i-1,j+1}) \end{aligned} \right] + \right. \\ & \left. \frac{1}{2Vol_{i-\frac{1}{2},j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i-1,j-1} + T_{i,j-1})(y_{i,j-1} - y_{i-1,j-1}) + (T_{i,j-1} + \mathbf{T}_{i,j})(y_{i,j} - y_{i,j-1}) + \\ & (T_{i-1,j} + \mathbf{T}_{i,j})(y_{i-1,j} - y_{i,j}) + (T_{i-1,j-1} + T_{i-1,j})(y_{i-1,j-1} - y_{i-1,j}) \end{aligned} \right] \right) \left( y_{i-\frac{1}{2},j+\frac{1}{2}} - y_{i-\frac{1}{2},j-\frac{1}{2}} \right) + \\ & \left( \frac{1}{2Vol_{i-\frac{1}{2},j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i-1,j-1} + T_{i,j-1})(y_{i,j-1} - y_{i-1,j-1}) + (T_{i,j-1} + \mathbf{T}_{i,j})(y_{i,j} - y_{i,j-1}) + \\ & (T_{i-1,j} + \mathbf{T}_{i,j})(y_{i-1,j} - y_{i,j}) + (T_{i-1,j-1} + T_{i-1,j})(y_{i-1,j-1} - y_{i-1,j}) \end{aligned} \right] + \right. \\ & \left. \frac{1}{2Vol_{i+\frac{1}{2},j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i,j-1} + T_{i+1,j-1})(y_{i+1,j-1} - y_{i,j-1}) + (T_{i+1,j-1} + T_{i+1,j})(y_{i+1,j} - y_{i+1,j-1}) + \\ & (\mathbf{T}_{i,j} + T_{i+1,j})(y_{i,j} - y_{i+1,j}) + (T_{i,j-1} + \mathbf{T}_{i,j})(y_{i,j-1} - y_{i,j}) \end{aligned} \right] \right) \left( y_{i-\frac{1}{2},j-\frac{1}{2}} - y_{i+\frac{1}{2},j-\frac{1}{2}} \right) + \\ & \left( \frac{1}{2Vol_{i+\frac{1}{2},j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i,j-1} + T_{i+1,j-1})(y_{i+1,j-1} - y_{i,j-1}) + (T_{i+1,j-1} + T_{i+1,j})(y_{i+1,j} - y_{i+1,j-1}) + \\ & (\mathbf{T}_{i,j} + T_{i+1,j})(y_{i,j} - y_{i+1,j}) + (T_{i,j-1} + \mathbf{T}_{i,j})(y_{i,j-1} - y_{i,j}) \end{aligned} \right] + \right. \\ & \left. \frac{1}{2Vol_{i+\frac{1}{2},j+\frac{1}{2}}} \left[ \begin{aligned} & (\mathbf{T}_{i,j} + T_{i+1,j})(y_{i+1,j} - y_{i,j}) + (T_{i+1,j} + T_{i+1,j+1})(y_{i+1,j+1} - y_{i+1,j}) + \\ & (T_{i,j+1} + T_{i+1,j+1})(y_{i,j+1} - y_{i+1,j+1}) + (\mathbf{T}_{i,j} + T_{i,j+1})(y_{i,j} - y_{i,j+1}) \end{aligned} \right] \right) \left( y_{i+\frac{1}{2},j-\frac{1}{2}} - y_{i+\frac{1}{2},j+\frac{1}{2}} \right) \end{aligned} \right] \end{aligned}$$

**It is easier  
to do this  
as 2 steps,  
however!!  
I'll explain  
this more  
later.**

## Finite Volume Representation

- Now, if we write the value of  $T_{i,j}$  to be at iteration level  $n+1$  and all other values of  $T$  to be at iteration  $n$ , and we add a similar term for the  $y$ -second derivative, we have developed a general form of the point-Jacobi numerical method (in finite-volume form)
- For the previous second derivative of  $T$  wrt  $x$ , we get:

# Point-Jacobi Finite Volume Representation

X- second  
derivative  
term only:

$$T_{i,j}^{n+1} \begin{bmatrix} \frac{(y_{i+1,j} - y_{i,j+1})}{Vol_{i+\frac{1}{2},j+\frac{1}{2}}} + \\ \frac{(y_{i,j+1} - y_{i-1,j})}{Vol_{i-\frac{1}{2},j+\frac{1}{2}}} + \\ \frac{(y_{i-1,j} - y_{i,j-1})}{Vol_{i-\frac{1}{2},j-\frac{1}{2}}} + \\ \frac{(y_{i,j-1} - y_{i+1,j})}{Vol_{i+\frac{1}{2},j-\frac{1}{2}}} \end{bmatrix} = \frac{1}{2Vol_{i,j}} \left[ \begin{aligned} & \left( \frac{1}{2Vol_{i+\frac{1}{2},j+\frac{1}{2}}} \left[ \begin{aligned} & (T_{i+1,j})(y_{i+1,j} - y_{i,j}) + (T_{i+1,j} + T_{i+1,j+1})(y_{i+1,j+1} - y_{i+1,j}) + \\ & (T_{i,j+1} + T_{i+1,j+1})(y_{i,j+1} - y_{i+1,j+1}) + (T_{i,j+1})(y_{i,j} - y_{i,j+1}) \end{aligned} \right] + \right. \\ & \left. \frac{1}{2Vol_{i-\frac{1}{2},j+\frac{1}{2}}} \left[ \begin{aligned} & (T_{i-1,j})(y_{i,j} - y_{i-1,j}) + (T_{i,j+1})(y_{i,j+1} - y_{i,j}) + \\ & (T_{i-1,j+1} + T_{i,j+1})(y_{i-1,j+1} - y_{i,j+1}) + (T_{i-1,j} + T_{i-1,j+1})(y_{i-1,j} - y_{i-1,j+1}) \end{aligned} \right] \right) \left( y_{i+\frac{1}{2},j+\frac{1}{2}} - y_{i-\frac{1}{2},j+\frac{1}{2}} \right) + \\ & \left( \frac{1}{2Vol_{i-\frac{1}{2},j+\frac{1}{2}}} \left[ \begin{aligned} & (T_{i-1,j})(y_{i,j} - y_{i-1,j}) + (T_{i,j+1})(y_{i,j+1} - y_{i,j}) + \\ & (T_{i-1,j+1} + T_{i,j+1})(y_{i-1,j+1} - y_{i,j+1}) + (T_{i-1,j} + T_{i-1,j+1})(y_{i-1,j} - y_{i-1,j+1}) \end{aligned} \right] + \right. \\ & \left. \frac{1}{2Vol_{i-\frac{1}{2},j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i-1,j-1} + T_{i,j-1})(y_{i,j-1} - y_{i-1,j-1}) + (T_{i,j-1})(y_{i,j} - y_{i,j-1}) + \\ & (T_{i-1,j})(y_{i-1,j} - y_{i,j}) + (T_{i-1,j-1} + T_{i-1,j})(y_{i-1,j-1} - y_{i-1,j}) \end{aligned} \right] \right) \left( y_{i-\frac{1}{2},j+\frac{1}{2}} - y_{i-\frac{1}{2},j-\frac{1}{2}} \right) + \\ & \left( \frac{1}{2Vol_{i-\frac{1}{2},j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i-1,j-1} + T_{i,j-1})(y_{i,j-1} - y_{i-1,j-1}) + (T_{i,j-1})(y_{i,j} - y_{i,j-1}) + \\ & (T_{i-1,j})(y_{i-1,j} - y_{i,j}) + (T_{i-1,j-1} + T_{i-1,j})(y_{i-1,j-1} - y_{i-1,j}) \end{aligned} \right] + \right. \\ & \left. \frac{1}{2Vol_{i+\frac{1}{2},j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i,j-1} + T_{i+1,j-1})(y_{i+1,j-1} - y_{i,j-1}) + (T_{i+1,j-1} + T_{i+1,j})(y_{i+1,j} - y_{i+1,j-1}) + \\ & (T_{i+1,j})(y_{i,j} - y_{i+1,j}) + (T_{i,j-1})(y_{i,j-1} - y_{i,j}) \end{aligned} \right] \right) \left( y_{i-\frac{1}{2},j-\frac{1}{2}} - y_{i+\frac{1}{2},j-\frac{1}{2}} \right) + \\ & \left( \frac{1}{2Vol_{i+\frac{1}{2},j-\frac{1}{2}}} \left[ \begin{aligned} & (T_{i,j-1} + T_{i+1,j-1})(y_{i+1,j-1} - y_{i,j-1}) + (T_{i+1,j-1} + T_{i+1,j})(y_{i+1,j} - y_{i+1,j-1}) + \\ & (T_{i+1,j})(y_{i,j} - y_{i+1,j}) + (T_{i,j-1})(y_{i,j-1} - y_{i,j}) \end{aligned} \right] + \right. \\ & \left. \frac{1}{2Vol_{i+\frac{1}{2},j+\frac{1}{2}}} \left[ \begin{aligned} & (T_{i+1,j})(y_{i+1,j} - y_{i,j}) + (T_{i+1,j} + T_{i+1,j+1})(y_{i+1,j+1} - y_{i+1,j}) + \\ & (T_{i,j+1} + T_{i+1,j+1})(y_{i,j+1} - y_{i+1,j+1}) + (T_{i,j+1})(y_{i,j} - y_{i,j+1}) \end{aligned} \right] \right) \left( y_{i+\frac{1}{2},j-\frac{1}{2}} - y_{i+\frac{1}{2},j+\frac{1}{2}} \right) \end{aligned} \right] \end{bmatrix}$$

## Finite Volume Representation

- You could derive a similar term for the y-second derivative and add this to the previous term for the x-second derivative to get a general form of the point-Jacobi iteration.
- The resulting discretization would work for any arbitrary orientation of the block

## A Similar Poisson Problem

- Now, instead of solving the “steady” heat conduction equation, let’s solve for the temperature field as a transient problem

$$\rho c_p \frac{\partial T}{\partial t} = k \left[ \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right]$$
$$\frac{\partial T}{\partial t} = \alpha \left[ \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right]$$

- This changes the “characteristics” of the equation and forces us to use different numerical algorithms

## Finite Difference Representation

- If we write the transient heat conduction equation in terms of finite-differences with the RHS terms to all exist at time level n (making the scheme ***EXPLICIT***)

$$\frac{\partial T}{\partial t} = \alpha \left[ \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right]$$

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n-1}}{2\Delta t} = \alpha \left[ \frac{T_{i+1,j}^n - 2T_{i,j}^n + T_{i-1,j}^n}{(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})^2} + \frac{T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n}{(y_{i+\frac{1}{2},j} - y_{i-\frac{1}{2},j})^2} \right]$$

(Richardson's scheme)  
(2nd order accurate  
in space and time)  
(*unconditionally UNSTABLE*)

$$\frac{T_{i,j}^{n+1} - T_{i,j}^n}{\Delta t} = \alpha \left[ \frac{T_{i+1,j}^n - 2T_{i,j}^n + T_{i-1,j}^n}{(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})^2} + \frac{T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n}{(y_{i+\frac{1}{2},j} - y_{i-\frac{1}{2},j})^2} \right]$$

(Simple Explicit)  
(1st order accurate in time,  
2nd order accurate in space)  
(*conditionally stable*)

## Finite Difference Representation

- And a stability analysis states that

$$\alpha \Delta t \left[ \frac{1}{(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})^2} + \frac{1}{(y_{i+\frac{1}{2},j} - y_{i-\frac{1}{2},j})^2} \right] \leq \frac{1}{2} \text{ or}$$

$$\Delta t \leq \frac{1}{2\alpha} \left[ \frac{(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})^2 + (y_{i+\frac{1}{2},j} - y_{i-\frac{1}{2},j})^2}{(x_{i+\frac{1}{2},j} - x_{i-\frac{1}{2},j})^2 + (y_{i+\frac{1}{2},j} - y_{i-\frac{1}{2},j})^2} \right] \text{ or } \Delta t \leq \frac{CFL}{2\alpha} \left[ \frac{Vol^2}{\Delta x^2 + \Delta y^2} \right]$$

See “Computational Fluid Mechanics and Heat Transfer” by D.A. Anderson, J. C. Tannehill, and R. H. Pletcher p 115

- **If we created a computational grid that has constant spacing in each direction independently, these equations would reduce to:**

$$T_{i,j}^{n+1} = T_{i,j}^n + \Delta t \alpha \left[ \frac{T_{i+1,j}^n - 2T_{i,j}^n + T_{i-1,j}^n}{(\Delta x)^2} + \frac{T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n}{(\Delta y)^2} \right]$$

$$\Delta t \leq \frac{CFL}{2\alpha} \left[ \frac{Vol^2}{(\Delta x)^2 + (\Delta y)^2} \right]$$

where  $CFL \approx 0.5$

- Thus the temperature at a given node at the next time level is completely dependent on values of the temperature field at the current time level (explicit scheme)
- Note that Jacobians or a control volume approach would make the RHS completely general

## Finite Difference Representation

- You can see that this algorithm would be relatively simple to program. You can also see that the convergence rate is dependent on the stability limit for  $\Delta t$ .
- If we changed the value of  $T$  on the RHS of the heat conduction equation to be at the next time level, our discretization would change to (Laasonen's Method):

**Fully Implicit**

$$T_{i,j}^{n+1} = T_{i,j}^n + \Delta t \alpha \left[ \frac{T_{i+1,j}^{n+1} - 2T_{i,j}^{n+1} + T_{i-1,j}^{n+1}}{(\Delta x)^2} + \frac{T_{i,j+1}^{n+1} - 2T_{i,j}^{n+1} + T_{i,j-1}^{n+1}}{(\Delta y)^2} \right]$$
$$T_{i,j}^{n+1} \left[ 1 + \frac{2\Delta t \alpha}{(\Delta x)^2} + \frac{2\Delta t \alpha}{(\Delta y)^2} \right] - \Delta t \alpha \left[ \frac{T_{i+1,j}^{n+1} + T_{i-1,j}^{n+1}}{(\Delta x)^2} + \frac{T_{i,j+1}^{n+1} + T_{i,j-1}^{n+1}}{(\Delta y)^2} \right] = T_{i,j}^n$$

- Which is unconditionally stable (we can use any value of  $\Delta t$ !) but involves inverting a 5-diagonal matrix.



## Finite Difference Representation

- Rather than invert a single large matrix for the solution, we can break up the solution into 2 steps and solve:

$$T_{i,j}^{n+1/2} - T_{i,j}^n = \frac{\Delta t}{2} \alpha \left[ \frac{T_{i+1,j}^{n+1/2} - 2T_{i,j}^{n+1/2} + T_{i-1,j}^{n+1/2}}{(\Delta x)^2} + \frac{T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n}{(\Delta y)^2} \right]$$
$$T_{i,j}^{n+1} - T_{i,j}^{n+1/2} = \frac{\Delta t}{2} \alpha \left[ \frac{T_{i+1,j}^{n+1/2} - 2T_{i,j}^{n+1/2} + T_{i-1,j}^{n+1/2}}{(\Delta x)^2} + \frac{T_{i,j+1}^{n+1} - 2T_{i,j}^{n+1} + T_{i,j-1}^{n+1}}{(\Delta y)^2} \right]$$

- The solution for each step involves inverting a tri-diagonal matrix. The algorithm is unconditionally stable so we can use any value for  $\Delta t$ . This scheme is known as the *ADI (Peaceman and Rachford, and Douglas)* or *SLR* method

## Time-Marching Control-Volume Method

- Another scheme involves using the control-volume approach for the RHS (spatial) discretization along with a finite-difference representation for the LHS (temporal)

$$\int_{V_{ol,s}} \frac{\partial T}{\partial t} dV_{ol,s} = \alpha \int_{V_{ol,s}} \left[ \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right] dV_{ol,s} = \alpha \int_{V_{ol,s}} \left[ \frac{\partial}{\partial x} \frac{\partial T}{\partial x} + \frac{\partial}{\partial y} \frac{\partial T}{\partial y} \right] dV_{ol,s}$$

*Remember*

$$\frac{(T^{n+1} - T^n) V_{ol,s}}{\Delta t} = \alpha \left[ \oint_{V_{ol,s}} \frac{\partial T}{\partial x} dy - \oint_{V_{ol,s}} \frac{\partial T}{\partial y} dx \right]$$

$$\int_{V_{ol}} \frac{\partial \phi}{\partial x} dV_{ol} = \oint_{V_{ol}} \phi dy$$

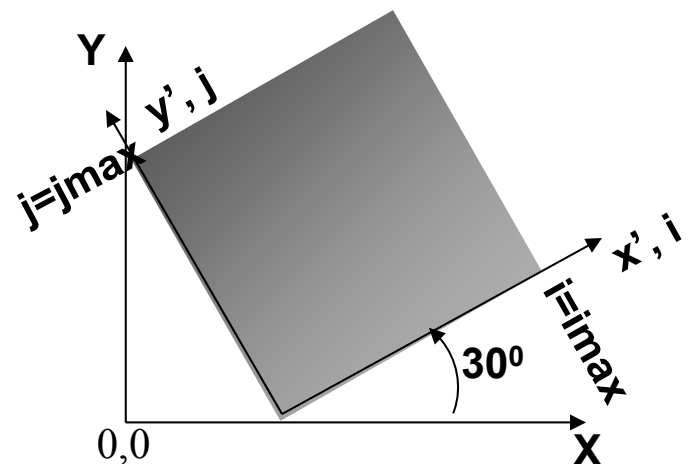
$$\int_{V_{ol}} \frac{\partial \phi}{\partial y} dV_{ol} = - \oint_{V_{ol}} \phi dx$$

$$\frac{T^{n+1} - T^n}{\Delta t} = \frac{\alpha}{V_{ol,s}} \left[ \oint_{V_{ol,s}} \left( \frac{1}{V_{ol,p}} \oint_{V_{ol,p}} T dy \right) dy + \oint_{V_{ol,s}} \left( \frac{1}{V_{ol,p}} \oint_{V_{ol,p}} T dx \right) dx \right]$$

- This approach has become very popular in CFD and simulation codes in general and is the approach that I would like for you to use in your projects

## Project-1

- **Write a computer program to solve the heat conduction equation on a single block of steel**
  - $k = 18.8 \text{ W/m K}$ ,  $\rho = 8000 \text{ kg/m}^3$ ,  $c_p = 500 \text{ J/(kg K)}$
  - Solve the transient heat conduction equation using the general, explicit control volume scheme on a non-uniform computational grid.
- **Use a computational grid:**
  - That is IMAX x JMAX in dimensions
    - Where IMAX=JMAX=101 AND IMAX=JMAX=501 (2 cases)
  - That has the following non-uniform distribution:  
 $\text{rot} = 30. \cdot 3.141592654 / 180.$   
 $x_p = \cos[0.5 \cdot \pi (\text{imax} - i) / (\text{imax} - 1)]$   
 $y_p = \cos[0.5 \cdot \pi (\text{jmax} - j) / (\text{jmax} - 1)]$   
 $x(i, j) = x_p \cdot \cos(\text{rot}) + (1. - y_p) \cdot \sin(\text{rot})$   
 $y(i, j) = y_p \cdot \cos(\text{rot}) + x_p \cdot \sin(\text{rot})$



## Project-1

- **Find the temperature distribution on a 1m x 1m block with Dirichlet boundary conditions:**
  - $T = 5. [\sin(\pi x_p) + 1.]$  at  $j = j_{\max}$
  - $T = \text{abs}(\cos(\pi x_p)) + 1.$  at  $j = 0$
  - $T = 3. y_p + 2.$  at  $i = 0$  and  $i = i_{\max}$

Note that  $x_p$  and  $y_p$  are used in these equations!
- **Iterate until the maximum residual magnitude drops below  $\text{TOLER} = 1.0 \times 10^{-5}$** 
  - Hint: The number of iterations will depend on the equation and the algorithm ( $\sim 10,000$  for 101 grid dimension)
- **Initialize the temperature field to a uniform value of 3.5**
- **Write out your converged solution to a (PLOT3D or equivalent) file that can be used for plotting or restart capability**
  - an example plot3d routine is on smartsite. Additional information can be found on the internet.

## Project-1

- **Run your code on a cluster node via the job launching scripts described in lecture-1**
  - Compiling of code can only be performed on hpc1

## Project-1

- **Due Thursday October 15<sup>th</sup> (PAY CLOSE ATTENTION TO WHAT IS ASKED FOR!):**
  - Statement of problem, equations, and algorithm(s) used
  - Listing of Fortran or C code (You may program this and all projects in C if you prefer)
  - Output of run. Print out the iteration number, the magnitude of the maximum residual, and the indices where the maximum residual was located
  - The CPU time from start of iteration to convergence. Let's use the clock routine from MPI for all projects by doing the following:

```
Include "mpif.h"
double start, finish, time

start = MPI_Wtime()
.
.
.
finish = MPI_Wtime()
time = finish - start
```

Time all parts of your code from start to finish so that we can see total performance as well as just the iteration loop so that we can see solver performance. Give both times.
- Plot of computational grid
- Plot of converged temperature fields