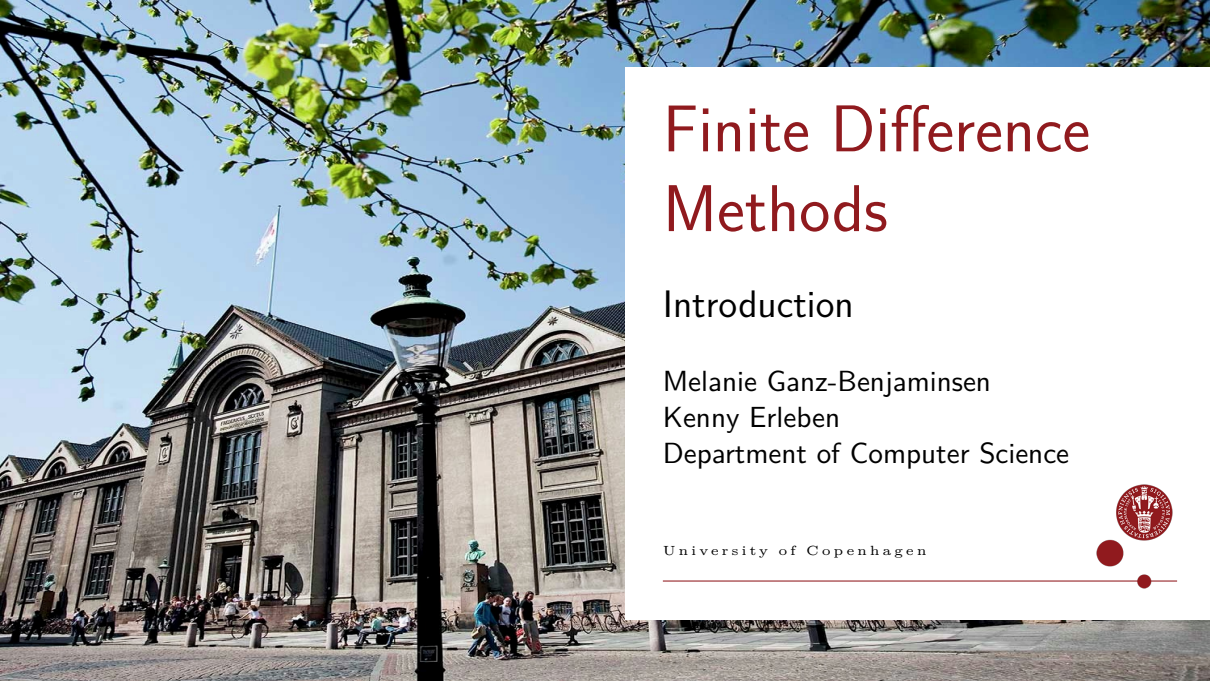


Finite Difference Methods

Introduction

Melanie Ganz-Benjaminsen
Kenny Erleben
Department of Computer Science

University of Copenhagen

Finite Difference Methods Quick Facts

- Finite-difference methods (FDM) are numerical methods for solving differential equations by approximating them with difference equations.
- The idea can be described as replacing derivatives by finite differences approximations.
- FDMs are thus discretization methods. Today, FDMs are one of the dominant approaches to numerical solutions of partial differential equations (PDEs).

Finite Difference Methods Quick Facts

- FDMs are often the first class of methods taught to students and form the foundation of numerical methods for many research fields.
- FDMs build on Taylor series expansions to derive approximation formulas for spatial and temporal derivatives. The remainder terms allow for a straightforward analysis of the discretization error. This error analysis is often referred to as the “big O” or “little o” error of a specific given method.

Finite Difference Methods Quick Facts

- FDMs are particularly intuitive when using regular grids for representing unknowns. Hence, this will be our starting point too. Often one simply starts from a given PDE and replaces derivatives with finite difference approximations using a cookbook of replacement rules.
- FDMs do extend beyond regular grids to unstructured meshes, and one may perform both stability analysis and formal verification to put a derived method on a strong theoretical foundation. For starters, we ignore all this and simply take a more practical cook-book approach where we will elaborate on problems as we encounter them.

Finite Difference Methods Quick Facts

- Although we suggest that one simply just go ahead and replace with finite difference approximations, one should be careful. Not all combinations of "rules" work. If one is not careful then the resulting discrete linear systems either have a non-unique solution or no solution. We adopt a naive trial-and-error approach where we construct our "schemes" and then investigate through experiments (verification) if they give a unique solution.

What are PDEs again?

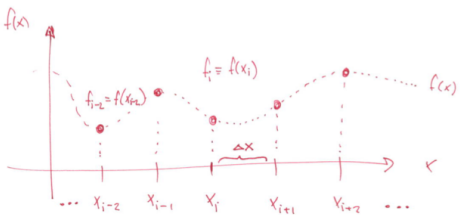
A partial differential equation (PDE) is a differential equation that contains unknown multivariable functions and their partial derivatives, e.g.

$$\frac{\partial u(x,y)}{\partial x} = 0$$

This PDE implies that the function u(x,y) is independent of x. However, the equation gives no information on the function's dependence on the variable y. Hence the general solution of this equation is

$$u(x,y) = f(y)$$

Using a Regular Grid



We have some function $f(x) : \mathbb{R} \mapsto \mathbb{R}$ on a regular grid.

The sampling positions are given by $x_i = \Delta x \cdot i$ where i is an integer index (label) for the i^{th} node.

We write

$$f_i \equiv f(x_i)$$

as a shorthand.

Image and Signal Processing Analogy

It can be quite insightful to think of the grid nodes x_i as sample points in signal or image f because

- It gives us an idea of how to pick Δx such that we can reconstruct f from the sample values f_i. Remember the Nyquist-Shannon sampling theorem.
- Many convolutions or filters applied to images and signals can be written as PDEs. Hence thinking of the grid as an image gives us a strong analogy to the fields of image and signal processing.

Adding the First set of Rules to our Cook-book

In the following we will derive the simplest and probably most wide-spread "rules" that our cookbook will contain. The rules are named

- Forward difference approximation
- Backward difference approximation
- Central difference approximation

Later these 3 basic rules will extend trivially to higher order derivatives and to higher dimensions.

Forward Finite Difference Approximations

We wish to compute $\frac{\partial f}{\partial x}$. We use a 1st order Taylor expansion

$$f(x_{i+1}) = f(x_i + \Delta x) = f(x_i) + \frac{\partial f_i}{\partial x}\Delta x + \mathbf{o}(\Delta x)$$

From this we get the *forward difference (FD) approximation*

$$\frac{\partial f_i}{\partial x} \approx \frac{f_{i+1} - f_i}{\Delta x}$$

Backward Finite Difference Approximations

We may also write the 1st order Taylor expansion as

$$f(x_{i-1}) = f(x_i) - \frac{\partial f_i}{\partial x}\Delta x + \mathbf{o}(\Delta x)$$

Leading to the *backward difference (BD) approximation*

$$\frac{\partial f_i}{\partial x} \approx \frac{f_i - f_{i-1}}{\Delta x}$$

Central Finite Difference Approximations

Adding together the forward and backward difference approximations

$$\frac{\partial f_i}{\partial x} \approx \frac{f_{i+1} - f_i}{\Delta x}$$
$$\frac{\partial f_i}{\partial x} \approx \frac{f_i - f_{i-1}}{\Delta x}$$

Yields

$$2\frac{\partial f_i}{\partial x} \approx \frac{f_{i+1} - f_i}{\Delta x} + \frac{f_i - f_{i-1}}{\Delta x}$$

Thus, we have the central difference (CD) approximation

$$\frac{\partial f_i}{\partial x} \approx \frac{f_{i+1} - f_{i-1}}{2\Delta x}$$

Higher order Derivatives

We wish to compute $\frac{\partial^2 f}{\partial x^2}$. Using CD approximation

$$\frac{\partial^2 f_i}{\partial x^2} \approx \frac{\frac{\partial}{\partial x} f_{i+\frac{1}{2}} - \frac{\partial}{\partial x} f_{i-\frac{1}{2}}}{\Delta x}$$

Recursively

$$\frac{\partial^2 f_i}{\partial x^2} \approx \frac{\frac{f_{i+1}-f_i}{\Delta x} - \frac{f_i-f_{i-1}}{\Delta x}}{\Delta x}$$

Results in the *central difference approximation*

$$\frac{\partial^2 f_i}{\partial x^2} \approx \frac{f_{i+1} - 2f_i + f_{i-1}}{\Delta x^2}$$

(Note the trick – we used halfway/mid points to evaluate the finite differences!)

Going to Higher Dimensions

Now we define $f(x, y) : \mathbb{R}^2 \mapsto \mathbb{R}$. Our regular grid is now a 2D grid with sampled positions $\mathbf{x}_{i,j} = (\Delta xi, \Delta yj)^T$ and we write $f_{i,j} \equiv f(\mathbf{x}_{i,j}) = f(x_i, y_j)$. Straightforward

$$\frac{\partial f_{i,j}}{\partial y} \approx \frac{f_{i,j+1} - f_{i,j-1}}{2\Delta y}$$

Similar for forward, backward, and higher order central difference approximations wrt. y. What about $\frac{\partial^2 f}{\partial x \partial y}$?

$$\frac{\partial^2 f_{i,j}}{\partial x \partial y} = \frac{\partial}{\partial x}\left(\frac{\partial f_{i,j}}{\partial y}\right) = \frac{f_{i+1,j+1} - f_{i+1,j-1} - f_{i-1,j+1} + f_{i-1,j-1}}{4\Delta x \Delta y}$$

A 2D Toy Example

Let $u(x, y) : \mathbb{R}^2 \mapsto \mathbb{R}$ and let the PDE be

$$\nabla^2 u - \kappa^2 u = f(x, y),$$

where $\kappa > 0 \in \mathbb{R}$ and f are known. We call such a PDE that defines what happens in the domain of a governing equation.

The term $\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ is approximated by 2nd order CD and the term u is simply the sample value at node (i, j).

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} - \kappa^2 u_{i,j} = f_{i,j}$$

This new approximate form we call a finite difference approximation equation or formula.

2D Example Continued

We may rewrite

$$\underbrace{\frac{1}{\Delta x^2}}_{c_{i-1,j}} u_{i-1,j} + \underbrace{\frac{1}{\Delta y^2}}_{c_{i,j-1}} u_{i,j-1} + \underbrace{\left(-\kappa^2 - \frac{2}{\Delta x^2} - \frac{2}{\Delta y^2}\right)}_{c_{i,j}} u_{i,j} + \underbrace{\frac{1}{\Delta y^2}}_{c_{i,j+1}} u_{i,j+1} + \underbrace{\frac{1}{\Delta x^2}}_{c_{i+1,j}} u_{i+1,j} = f_{i,j}$$

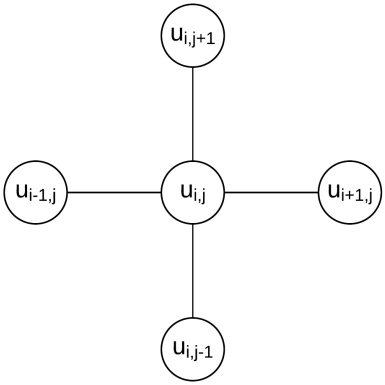
Or

$$u_{i,j} = \frac{f_{i,j} - c_{i-1,j}u_{i-1,j} - c_{i,j-1}u_{i,j-1} - c_{i,j+1}u_{i,j+1} - c_{i+1,j}u_{i+1,j}}{c_{i,j}}$$

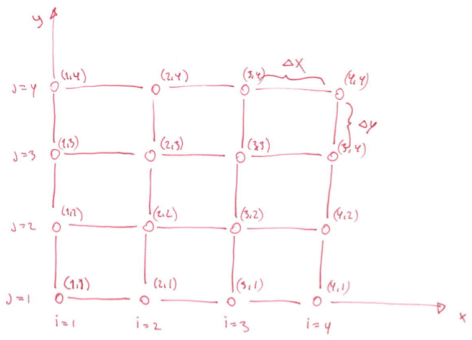
This is called an *update formula* for the unknown $u_{i,j}$. c's are called the coefficients and f are referred to as the source term or right-hand side term.

The Stencil

From the update formula of the unknown $u_{i,j}$ we discover what other unknown values it depends on. This defines what is called a *stencil* and can be graphically illustrated as follows



The Computational Domain



We need to update all nodes in our domain.

This is called a computational grid or computational mesh.

Update Stencil = Fixed Point Problem

The update stencils mean that $\forall i, j$ we have a linear equation

$$u_{i,j} = \underbrace{\frac{f_{i,j} - c_{i-1,j}u_{i-1,j} - c_{i,j-1}u_{i,j-1} - c_{i,j+1}u_{i,j+1} - c_{i+1,j}u_{i+1,j}}{c_{i,j}}}_{\equiv g(u_{i,j})}$$

$$u_{i,j} = g(u_{i,j})$$

Where $g(\cdot)$ is an affine function. This is a fixed-point problem and one way to iterative solve this is to make a guess $u_{i,j}^0 = 0$ and then compute

$$u_{i,j}^{k+1} = g(u_{i,j}^k)$$

Starting with $k = 0$ and continuing updating until $u_{i,j}^k$ does not change.

Computing a Solution – Almost

Usually, the updates are done one-by-one

```
while not converged
  for j=1:4
    for i=1:4
      u(i,j)= ( f(i,j)-c(i-1,j)u(i-1,j)-...
                ...-c(i+1,j)u(i+1,j) ) / c(i,j)
    next i
   next j
end
```

Here i is termed the running index.

Trouble with the Model

The model so far presented for our Toy problem is ill-posed,

- We need boundary conditions (BCs) to tell us how u (in the example) behaves on the boundary of the domain.
- We may also need initial conditions (in case u is time-dependent which it is not in our example) to tell what the value of u would be for $t = 0$.

The missing BCs are tied to our update code works too.

Dealing with Boundary Conditions

For most of the examples, we will only consider Dirichlet and von Neumann boundary conditions, these are defined as follows:

$$u(x, y) = k_d, \ \forall x, y \in \Gamma,$$
$$\frac{\partial u}{\partial x} = k_n, \ \forall x, y \in \Gamma.$$

Here Γ is the set of points on the boundary. Now let us discuss how to modify our update formulas to incorporate boundary conditions.

Dirichlet is also sometimes called fixed boundary conditions and Neumann is referred to as Natural boundary conditions.

Techniques for handling Boundary Conditions

Before we go into details we will zoom out. Overall in our opinion, there are two main techniques for dealing with boundary conditions

- Elimination of unknown variables, that means the boundary variables are removed from the set of unknowns, and as such are no longer present in the problem.
- Adding ghost variables to make handling of boundary conditions simpler from a coding viewpoint.

Their implementations might take different paths. For instance, the elimination technique can be done either by adding if-statements to the "update" loop or using a Schur complement when working with matrices. In the following, we will explore both techniques and various ways to implement them.

One sided Differences on The Boundary

One remedy is to apply one-sided approximations on the boundary.

- Make sure the physical boundaries coincide with the border of the boundary cell.
- Use the boundary conditions to eliminate "unknown" terms

As an example, we apply a von Neumann boundary condition $\frac{\partial u}{\partial x} = 0$. For the left border, we have

$$\begin{aligned}\frac{\partial^2 u_{1,j}}{\partial x^2} &= \frac{\frac{\partial}{\partial x} u_{1+\frac{1}{2},j} - \frac{\partial}{\partial x} u_{\frac{1}{2},j}}{\Delta x} \\ &= \frac{\frac{u_{2,j}-u_{1,j}}{\Delta x} - \frac{\partial}{\partial x} u_{\frac{1}{2},j}}{\Delta x} = \frac{u_{2,j} - u_{1,j}}{\Delta x^2}\end{aligned}$$

Since $\frac{\partial u_{\frac{1}{2},j}}{\partial x} = 0$. Clearly, the modified stencil now only uses "valid" nodes that exist in the domain.

The Ugliness of One-sided Differences on the Boundary

As we just saw when plugging boundary conditions into the finite difference approximations of our governing equation we get a modified stencil that only uses valid domain nodes. There are unfortunately some challenges we must overcome in order to use this approach:

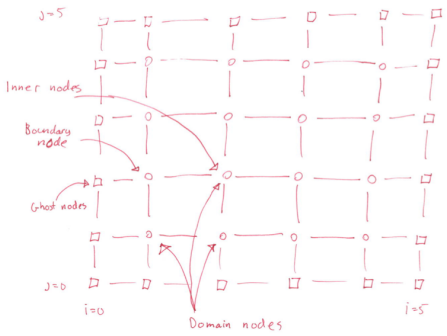
- We must crunch a lot of math to derive all updated formulas for all the boundaries in our problem.
- When implementing the update method on Page 20 we would need to use if-statements or similar logic to determine which update formulas to use for different types of nodes in the domain as boundary nodes would need their modified counterparts of the governing equation.

Ghost nodes provide us with a different approach for implementing the boundary conditions that make the coding effort somewhat simpler.

Using Ghost Nodes/Cells

To use the idea of ghost nodes we append extra nodes to our domain as shown here.

Doing this is sometimes called “padding”. Now when naively running the update code from page 20 we can evaluate any central difference approximation in our domain without accessing non-existing grid nodes in the computer memory. However, how should we update the values stored in the actual ghost nodes?



Setting the Ghost Nodes

Imagine once again that we have $\frac{\partial u}{\partial x} = 0$ on our domain boundary.
A central difference approximation would imply

$$\frac{\partial u_{\frac{1}{2},j}}{\partial x} = \frac{u_{1,j} - u_{0,j}}{\Delta x} = 0$$

From this, we get an update-formula for the ghost node $u_{0,j}$

$$u_{0,j} = u_{1,j}$$

Thus, prior to evaluating update formulas on the domain, we make an initial pass over the ghost nodes and apply update formulas to set these to respect our boundary conditions.

It is about Solving Linear Systems

We observe that approximation formulas and their reformulated update stencils are linear systems of equations.

It means the iterative update scheme (the fixed point approach) is an iterative solver for solving a linear system.

Benefit: It does not need storage to formulate a matrix.

Disadvantage: It may need many iterations to converge.

Idea: If we create a matrix form of all the equations then we can apply a different solver.

Like LU, QR or Cholesky factorizations, or even Conjugate Gradients or other types of solvers.

Connection to Matrix Form

We may write up all finite difference approximation formulas simultaneously for all unknowns $u(i,j)$

$$\begin{aligned} c_{10}u_{10} + c_{01}u_{01} + c_{11}u_{11} + c_{21}u_{21} + c_{12}u_{12} &= f_{11} \\ &\vdots \quad \vdots \\ c_{43}u_{43} + c_{34}u_{34} + c_{44}u_{44} + c_{54}u_{54} + c_{45}u_{45} &= f_{44} \end{aligned}$$

Next we may concatenate all u-values into one vector $\mathbf{u}$ which we partition into a part of $\mathbf{u}_D$ that holds domain nodes and a part $\mathbf{u}_G$ that holds ghost nodes.

$$\mathbf{A}\mathbf{u} = \begin{bmatrix} \mathbf{A}_{DD} & \mathbf{A}_{DG} \end{bmatrix} \begin{bmatrix} \mathbf{u}_D \\ \mathbf{u}_G \end{bmatrix} = \mathbf{f}.$$

Here $\mathbf{A}$ is a matrix containing the coefficients c_{ij} and $\mathbf{f}$ is a concatenation of all f_{ij} terms. We call this process a Matrix Assembly.

Connection to Matrix Form Continued

Similarly, we may write up the update-formulas for the boundary conditions as a simultaneous system of equations

$$\begin{aligned} b_{01}u_{01} + b_{11}u_{11} &= k_{01} \\ \vdots \quad & \vdots \\ b_{54}u_{54} + b_{44}u_{44} &= k_{54} \end{aligned}$$

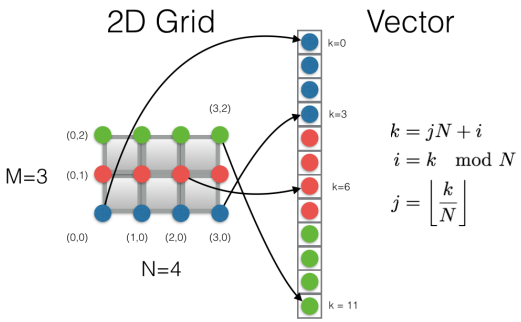
To keep it general we introduced b_{ij} as symbols for the coefficients in the update formulas and k_{ij} for the right hand side term. As before we may introduce the matrix notation

$$\mathbf{Bu} = \begin{bmatrix} \mathbf{B}_{GD} & \mathbf{B}_{GG} \end{bmatrix} \begin{bmatrix} \mathbf{u}_D \\ \mathbf{u}_G \end{bmatrix} = \mathbf{k}.$$

Connection to Matrix Form Continued

For mapping the update formulas into matrices we need to design a memory layout that maps from a pair of 2D grid indices (i, j) into a linear vector index k.

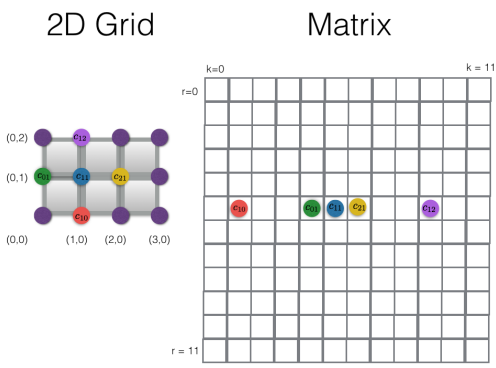
We simply just stack the grid rows on top of each other.



Connection to Matrix Form Continued

The same mapping is used to map the coefficients of a stencil into rows of a Matrix.

The stencil of node $(1, 1)$ maps to row 5 and the coefficients of the stencil will be placed appropriately into the columns of row 5.



Connection to Matrix Form Continued

Let the grid be of $M \times N$ nodes then the coefficient $c_{a,b}$ from the finite difference approximation for the (i, j) node maps into the $\mathbf{A}$-matrix as

$$\mathbf{A}_{(j\,N+i),(b\,N+a)} = c_{a,b}$$

Similar for the b-coefficients and the $\mathbf{B}$-matrix.

Notice that $u_{i,j}$ maps into

$$\mathbf{u}_{(j\,N+i)} = u_{i,j}$$

Similar to the right-hand-side vectors.

The Final Story on Matrix Form

Discretization of the governing equations and the boundary conditions results in two simultaneous linear systems

$$\begin{bmatrix}\mathbf{A}_{DD} & \mathbf{A}_{DG}\end{bmatrix}\begin{bmatrix}\mathbf{u}_D \\ \mathbf{u}_G\end{bmatrix} = \mathbf{f},$$

$$\begin{bmatrix}\mathbf{B}_{GD} & \mathbf{B}_{GG}\end{bmatrix}\begin{bmatrix}\mathbf{u}_D \\ \mathbf{u}_G\end{bmatrix} = \mathbf{k}.$$

That we can write up as a single full linear system,

$$\begin{bmatrix}\mathbf{A}_{DD} & \mathbf{A}_{DG} \\ \mathbf{B}_{GD} & \mathbf{B}_{GG}\end{bmatrix}\begin{bmatrix}\mathbf{u}_D \\ \mathbf{u}_G\end{bmatrix} = \begin{bmatrix}\mathbf{f} \\ \mathbf{k}\end{bmatrix}$$

When we have assembled this system we can simply solve this to find our solution.

Real Sub-Blocks or Just Index Sets?

Looking at

$$\begin{bmatrix} \mathbf{A}_{DD} & \mathbf{A}_{DG} \\ \mathbf{B}_{GD} & \mathbf{B}_{GG} \end{bmatrix} \begin{bmatrix} \mathbf{u}_D \\ \mathbf{u}_G \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{k} \end{bmatrix}$$

There are two ways to consider how to implement this.

- One may take the blocked partitioning literally, and constructing the sub-matrixes $\mathbf{A}$ and $\mathbf{B}$ and assemble these into the the full matrix as shown above.
- Alternative one may simply consider the blocks above as index sets into a full matrix. This is nice as it allows one to pre-allocate the full matrix and simply index it into needed entries. Hence, the blocks are an imaginary partitioning of the full system and the sub-block matrices do not really exist.

In implementations we often use the index approach, but when explaining the math we take the sub-block viewpoint.

The Schur Reduction

By doing blocked row-operations

$$\begin{bmatrix} \mathbf{A}_{DD} & \mathbf{A}_{DG} \\ \mathbf{B}_{GD} & \mathbf{B}_{GG} \end{bmatrix} \begin{bmatrix} \mathbf{u}_D \\ \mathbf{u}_G \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{k} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{A}_{DD} & \mathbf{A}_{DG} \\ \mathbf{B}_{GG}^{-1}\mathbf{B}_{GD} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}_D \\ \mathbf{u}_G \end{bmatrix} = \begin{bmatrix} \mathbf{f}_D \\ \mathbf{B}_{GG}^{-1}\mathbf{c} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{A}_{DD} - \mathbf{A}_{DG}\mathbf{B}_{GG}^{-1}\mathbf{B}_{GD} & \mathbf{0} \\ \mathbf{B}_{GG}^{-1}\mathbf{B}_{GD} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}_D \\ \mathbf{u}_G \end{bmatrix} = \begin{bmatrix} \mathbf{f}_D - \mathbf{A}_{DG}\mathbf{B}_{GG}^{-1}\mathbf{k} \\ \mathbf{B}_{GG}^{-1}\mathbf{k} \end{bmatrix}$$

Observe that the first row gives,

$$\left(\mathbf{A}_{DD} - \mathbf{A}_{DG}\mathbf{B}_{GG}^{-1}\mathbf{B}_{GD}\right)\mathbf{u}_D = \mathbf{f} - \mathbf{A}_{DG}\mathbf{B}_{GG}^{-1}\mathbf{k}$$

The Schur Reduction

We can use this equation to solve for $\mathbf{u}_D$ and then substitute into the second row to compute $\mathbf{u}_G$ (which we do not care about),

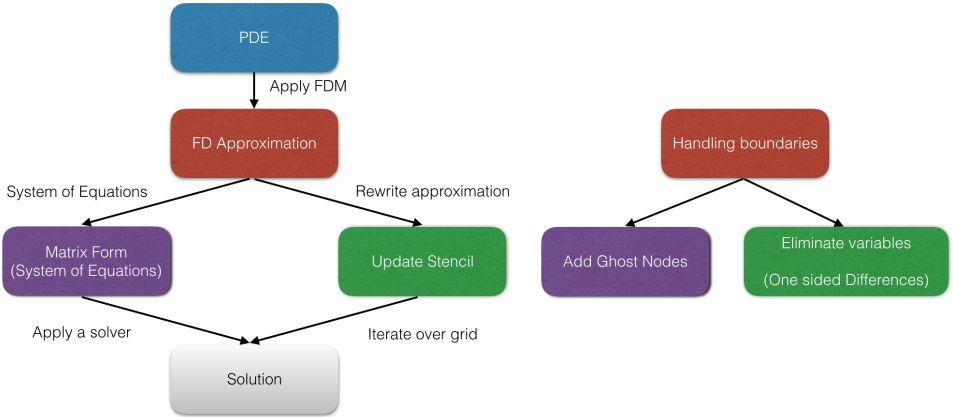
$$\mathbf{u}_G = \mathbf{B}_{GG}^{-1}(\mathbf{k} - \mathbf{B}_{GD}\mathbf{u}_D)$$

This is called the Schur method (or Schur reduction).

The Schur reduction shows that in the end, all ghost values disappear.

- In most cases one would not solve the system like this – but rather apply an iterative or direct solver to the full system.
- But using a Schur complement can be a benefit if there are far fewer domain variables than ghost variables, or to get a better conditioned system.

Overview of Approaches



A Note on Iterative Methods

Solve the linear system $\mathbf{Ax} = \mathbf{b}$ using a splitting method

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ (\mathbf{L} + \mathbf{D} + \mathbf{U})\,\mathbf{x} &= \mathbf{b} \\ (\mathbf{L} + \mathbf{D})\,\mathbf{x}^{k+1} &= \mathbf{b} - \mathbf{U}\mathbf{x}^{k} \\ \mathbf{x}^{k+1} &= (\mathbf{L} + \mathbf{D})^{-1}\left(\mathbf{b} - \mathbf{U}\mathbf{x}^{k}\right) \end{aligned}$$

This is a Gauss-Seidel (GS) method. Write the i^{th} component of the last equation

$$\mathbf{x}_i^{k+1} = \frac{\mathbf{b}_i - \sum_{j>i} \mathbf{A}_{ij}\mathbf{x}_j^{k} - \sum_{j<i} \mathbf{A}_{ij}\mathbf{x}_j^{k+1}}{\mathbf{A}_{ii}}$$

A Note on Iterative Methods

Observe if $j > i$ then $\mathbf{x}_j^{k+1} = \mathbf{x}_j^k$ Thus, doing in place update

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \frac{\mathbf{b}_i - \sum \mathbf{A}_{ij}\mathbf{x}_j}{\mathbf{A}_{ii}}$$

Time Integration

Up to this point, we have mainly studied spatial derivatives. Time derivatives are special. Often one would go with the guidelines

- Use either FD (called explicit time integration) or BD (called implicit time integration) rules for temporal derivatives.
- Use CD for spatial derivatives for the interior of domains and FD/BD on boundaries as appropriate.

These are merely guidelines, they are not bulletproof. In the following, we will introduce yet a new rule to our cookbook, which is extremely useful for dealing with time integration of advection terms.

Semi Lagrangian Time Integration

Basic Idea:

- Convert nodes into particles
- Trace particles back in time
- Find the values in the past
- Copy the values to the present

This is often used to deal with advection terms

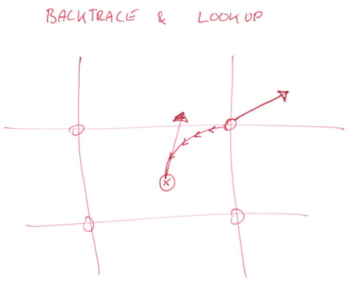
$$\frac{\partial \phi}{\partial t} = -\left(\mathbf{u} \cdot \nabla\right) \phi$$

where ϕ is a scalar field and $\mathbf{u}$ is the associated velocity field and for now we additionally assume an incompressible flow meaning $\nabla \cdot \mathbf{u} = 0$.

Advection describes how much ϕ is pushed around by $\mathbf{u}$.

Step 1: Traceback in time

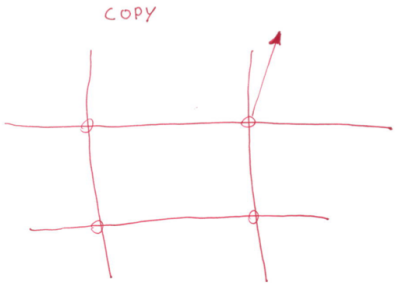
Convert grid nodes to particles and follow the trajectory of the particle back in time.



Use some interpolation method to look up the value.

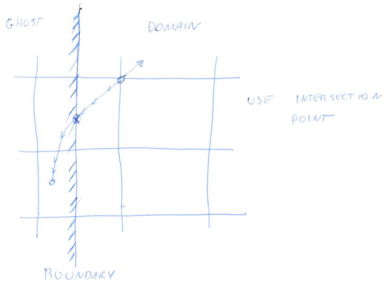
Step 2: Copy forward in time

Copy the value stored at the past particle position to the current time grid node position.



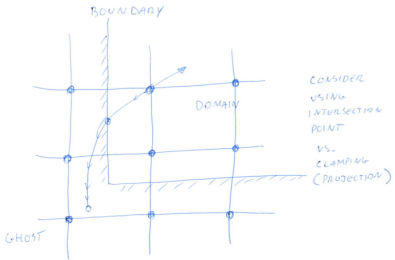
Hitting the Wall

If we trace beyond a wall then stop at the intersection point between trajectory and wall.



Clamping - The Fast Way to handle the Wall

Clamping (meaning associated with the closest boundary point) might be faster but may have side effects



Depends on time-step size and wall shape.

The Math of It

From a grid viewpoint, we are sitting fixed in space and looking at ϕ as it moves through space and we should think of ϕ as a spatially varying function $\phi(\mathbf{x}, t)$. Thus, the full derivative from the grid view becomes

$$\frac{D\phi(\mathbf{x}, t)}{Dt} = \sum_i \frac{\partial \phi}{\partial \mathbf{x}_i} \frac{\partial \mathbf{x}_i}{\partial t} + \frac{\partial \phi}{\partial t} = (\mathbf{u} \cdot \nabla)\, \phi + \frac{\partial \phi}{\partial t}$$

Now imagine instead that we take a particle viewpoint, that is we move with a particle and can observe as we move along how the quantity $\phi(t)$ of the particle will change in time as

$$\frac{D\phi(t)}{Dt} = k\,.$$

In the particle view, we do not observe $\mathbf{x}$ of the particle because we are moving with it. That is why ϕ now appears to only depend directly on t and not on $\mathbf{x}$.

The Math of It

This means that the particle view $\frac{D\phi(t)}{Dt} = k$ is the same as the grid view $\frac{D\phi(\mathbf{x},t)}{Dt}$,

$$\frac{D\phi(t)}{Dt} = \frac{D\phi(\mathbf{x},t)}{Dt}.$$

That is

$$k = (\mathbf{u} \cdot \nabla)\,\phi + \frac{\partial \phi}{\partial t}$$

In our case $k = 0$ as we consider ϕ to be conserved. We could encounter problems with sinks or sources that would cause $k \neq 0$.

The cool thing about this interchangeable viewpoint insight is that we can go with the viewpoint that makes the numerical method easier to solve. In our case, we swap the grid view with the particle view and solve the time integration problem with the particle view, before finally switching back to the grid view again.

The Implicitness of It

Having to solve

$$\frac{D\phi(t)}{Dt} = k$$

We may choose an implicit first-order time integration (BD in time)

$$\phi^t = \phi^{t-\Delta t} + \Delta t k$$

In the case of advection from Page 42 we have $k = 0$. That means we must find $\phi^{t-\Delta t}$ in order to find ϕ^t.

One can think of this that we wish to trace back in time to find the imaginary particle that will hit the location (grid node) we are currently updating.

Tracing Back in Time

From the definition of a velocity field as the time derivative of position, we have

$$\frac{\partial \mathbf{x}}{\partial t} = \mathbf{u}$$

For the sake of example, we here apply a first-order backward finite difference approximation

$$\frac{\mathbf{x}^t - \mathbf{x}^{t-\Delta t}}{\Delta t} = \mathbf{u}$$

Using this approximation equation we can find the particle position (update formula) at $t - \Delta t$,

$$\mathbf{x}^{t-\Delta t} = \mathbf{x}^t - \Delta t \mathbf{u}$$

Here $\mathbf{x}^t$ will be the grid node position we want to know ϕ^t for.

Advecting the Field Forward

Knowing the location of the particle in the past that will hit our current grid node, we can now advect the ϕ field forward in time by doing the update ϕ by

$$\phi^t \leftarrow \phi^{t-\Delta t}$$

After the update, we simply store the new ϕ^t value at the grid node to swap back the grid viewpoint. Our problem in this approach is that we do not know $\phi^{t-\Delta t}$ as $\mathbf{x}^{t-\Delta t}$ might not be located at a grid node.

Advecting the Field Forward

Hence to find $\phi^{t-\Delta t}$ we interpolate from the enclosing grid nodes at location $\mathbf{x}^{t-\Delta t}$. One important observation from doing an interpolation is that:

- Any interpolated value will be in the range of the enclosing nodal values. Never outside the range.

Thus, we never add more "ϕ" to the grid nodes. Instead we always "smooth" out "ϕ". We say we numerically lose or dissipate ϕ.

Expanding The Cookbook

Next, we will briefly extend our cookbook with 3 more ingredients. They are often quite useful when working with computational fluid dynamics although they are not limited to this application domain. The techniques are

- Fractional step method
- Pressure projection
- Staggered grids

The Fractional Step Method

Given some PDE

$$\frac{\partial \phi(\mathbf{x}, t)}{\partial t} = f_1(\mathbf{x}, \mathbf{t}, \phi, \nabla\phi, \ldots) + \cdot + f_n(\mathbf{x}, \mathbf{t}, \phi, \nabla\phi, \ldots)$$

with initial condition $\phi(\cdot, t) = \phi^t$. Split into sequence of sequential subproblems – steps. First step solve

$$\frac{\partial \phi_1(\mathbf{x}, t)}{\partial t} = f_1(\mathbf{x}, \mathbf{t}, \phi_1, \nabla\phi_1, \ldots).$$

We use the solution $\phi_1^{t+\Delta t}$ of first step as initial condition for the second step $\phi_2(\cdot, t) = \phi_1^{t+\Delta t}$

$$\frac{\partial \phi_2(\mathbf{x}, t)}{\partial t} = f_2(\mathbf{x}, \mathbf{t}, \phi_2, \nabla\phi_2, \ldots)$$

and so forth...

The Fractional Step Method Continued

...until we use the result ϕ_{n-1} as initial condition for the final step $\phi_n(\cdot, t) = \phi_{n-1}^{t+\Delta t}$

$$\frac{\partial \phi_n(\mathbf{x}, t)}{\partial t} = f_n(\mathbf{x}, \mathbf{t}, \phi_n, \nabla \phi_n, \ldots)$$

The solution of the final step is the solution for the original PDE. $\phi(\cdot, t + \Delta t) = \phi_n^{t+\Delta t}$

Why does this work?

As an example consider using first order explicit time integration then we have

$$\begin{aligned}
\phi_1^{t+\Delta t} &= \phi^t + \Delta t f_1^t \\
\phi_2^{t+\Delta t} &= \phi_1^{t+\Delta t} + \Delta t f_2^t \\
&\vdots \quad \vdots \\
\phi_n^{t+\Delta t} &= \phi_{n-1}^{t+\Delta t} + \Delta t f_n^t \\
\phi^{t+\Delta t} &= \phi_n^{t+\Delta t}
\end{aligned}$$

Back substitution yields

$$\phi^{t+\Delta t} = \phi^t + \Delta t \sum_{i=1}^{n} f_i^t$$

But this is first order explicit time integration of the original PDE.

Why is this clever?

- We can take one big problem and break it into several smaller problems
- We can apply a specific solution technique for each sub problem independently of the other sub problems. Example use implicit time integration for one term but explicit time integration for another etc..
- It makes solution technique more modular and allow us to easy change choices later (different discretization choices)
- One major downside! It adds a sequential nature to the overall method.

The Pressure Projection Trick

We got the PDE

$$\rho \frac{\partial \mathbf{u}}{\partial t} = -\nabla p$$

For $\mathbf{u} = (u, v)^T$ combined with the equation

$$\nabla \cdot \mathbf{u} = 0$$

The Pressure Projection Trick

We do an implicit first order time integration and then we take the divergence of the result

$$\mathbf{u}^{t+1} = \mathbf{u}^t - \frac{\Delta t}{\rho}\nabla p^{t+\Delta t}$$

$$\nabla \cdot \mathbf{u}^{t+1} = \nabla \cdot \mathbf{u}^t - \frac{\Delta t}{\rho}\nabla \cdot \nabla p^{t+\Delta t}$$

$$\nabla^2 p' = \nabla \cdot \mathbf{u}^t$$

where $p' = \frac{\Delta t}{\rho} p^{t+\Delta t}$. This is a Poisson equation.

Pressure Projection – Final recipe

Step 1, compute

$$\nabla^2 p' = \nabla \cdot \mathbf{u}^t$$

Step 2, compute

$$\mathbf{u}^{t+1} = \mathbf{u}^t - \nabla p'$$

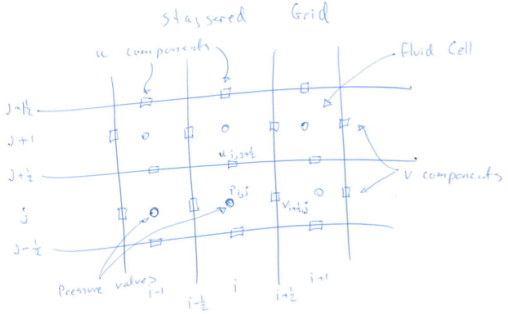
That is it.

A Short Story about a Staggered Grid

In computational fluid dynamics, it is custom to

- Store velocity components at face centers
- Store scalar values (density, pressure) at cell centers

Example



Benefits of Staggered Grid

- Make it very easy to add boundary conditions like $u = 0$ and $v = 0$
- Make it easy to compute pressure gradients for u and v velocity components.
- Results in more "accurate" central difference approximations (Study group work: can you explain why it gets more accurate?)
- Drawback – may make implementation a bit messier

Assignment

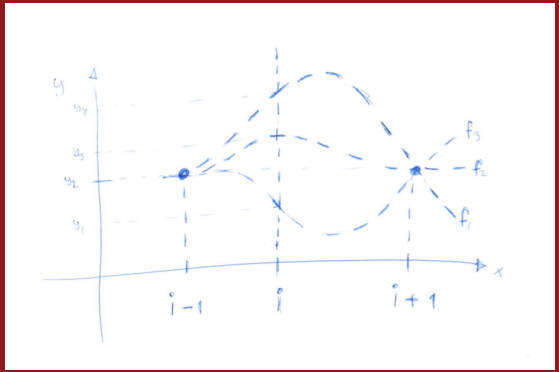
- Show the order of error for the FD, BD, CD approximations from previous slides (Hint find the term $\mathbf{o}(...)$ or $\mathcal{O}(...)$) Notice that you have to rederive the CD formula differently. Write up the 2nd order Taylor series forward and backward expansions. Then subtract the backward Taylor expansion from the forward Taylor expansion. Recall that by definition we write

$$f(\Delta x) \in \mathbf{o}(\Delta x^n) \Rightarrow \lim_{\Delta x \to 0} \frac{f(\Delta x)}{\Delta x^n} = 0$$

  For any power n.
- Based on your order analysis answer what is the benefit of the CD approximation compared to the FD or BD approximations?

Assignment

- Do you see any drawbacks to the CD approximation? (Hint compute FD, BD and CD for $\frac{\partial f_1(x_i)}{\partial x}$, $\frac{\partial f_2(x_i)}{\partial x}$, and $\frac{\partial f_3(x_i)}{\partial x}$ shown in the figure)



Assignment

Show in full detail that $\frac{\partial^2 f_{i,j}}{\partial x \partial y} = \frac{\partial^2 f_{i,j}}{\partial y \partial x}$ holds when using finite difference approximations as outlined/defined on Page 14.

Assignment

- Explain why the name running index on page 20 makes sense.
- What problems can you find with the pseudo-code on slide page 20 if you actual run the code as it is shown?

Assignment

Use a 4-by-4 grid for the small 2D Toy example from page 15 and apply the boundary conditions

$$\frac{\partial u}{\partial x} = 0$$

On all vertical boundaries and

$$\frac{\partial u}{\partial y} = 0$$

On all horizontal boundaries. Let $f(x, y) = x + y$ and $\kappa = 2$. Use the ghost node approach.

- Derive update formulas for the ghost nodes
- Derive update formulas for all domain nodes

Assignment

We will now continue the example from Page 67.

- Explain how the approximation equations from the governing equations and boundary conditions are mapped into a matrix using index sets.
- Write a Matlab code that assembles the corresponding matrix rows for the domain nodes.
- Write a Matlab code that assembles the corresponding matrix rows for the ghost nodes

Assignment

Use your code solution from Page 68.

- Examine the fill pattern of the **A** and **B** matrices from the Toy example. What can you say about them?
- Examine the eigenvalues of the full matrix system

$$\begin{bmatrix} \mathbf{A}_{DD} & \mathbf{A}_{DG} \\ \mathbf{B}_{GD} & \mathbf{B}_{GG} \end{bmatrix} \begin{bmatrix} \mathbf{u}_D \\ \mathbf{u}_G \end{bmatrix} = \begin{bmatrix} \mathbf{f}_D \\ \mathbf{k}_G \end{bmatrix}$$

  Speculate whether you can solve this linear system or not.

Assignment

Consider the 1D example of

$$\frac{\partial^2 u}{\partial x^2} = 0$$

Let the domain be from $x_a = 0$ to $x_b = 1$ (sample any $n > 3$ points in between), and use a regular mesh with $\Delta x = 1/n$ and a central difference approximation.

- Draw the computational mesh
- Write the finite difference approx. the equation for the i^{th} node
- Apply the boundary condition $\frac{\partial u}{\partial x} = 0$ at x_a and x_b ($i = 1$ and $i = n$) and use the ghost nodes x_0 and x_{n+1}.
- Assemble one whole matrix system. What structure does the matrix have, how many zero-eigenvalues, and how many zero-rows? (Hint: is it singular).

Assignment

Continue with the example from Page 70.

- Use simple geometry considerations and formulate the general solution. What is needed to find a particular solution? (Hint: How to make the matrix non-singular).
- Discuss how one would approach an efficient matrix representation. (Hint: We often use iterative linear system solvers and only need to know the value of $\mathbf{Au}$ for some given $\mathbf{u}$ and not $\mathbf{A}$.)
- For the specific example examine what would be the most efficient way to store the $\mathbf{A}$ matrix. Exactly how many numbers in memory does one really need? (More Hints: Write a small piece of Matlab code that uses the update formulas you have derived to compute $\mathbf{Au}$)

Assignment

- Derive the Gauss-Seidel update formula shown on page 40 in full detail.
- Compare the Gauss-Seidel update formula to the solver pseudo code on slide 20. (Hint: can you see a connection?)

Assignment

- Define some ϕ field on a regular 2D grid (Hint: The MatLab function peaks might be useful).
- Use the analytical velocity field $\mathbf{u}(x, y) = (y, -x)^T$
- Explain in detail how to use semi-Lagrangian time integration to implement a scheme for solving

$$\frac{\partial \phi}{\partial t} = -(\mathbf{u} \cdot \nabla)\phi$$

  on a regular mesh domain.
- Ideally ϕ should just rotate as a function of time. Implement your derived scheme and examine if this is true. (Hint: You have to carefully define an experiment and identify what causes ϕ to change).
- Change your method to use a 2nd order Runge Kutta method (RK2) for time-integration (also known as mid-point method).

Assignment

- Download the smoke2D code from the course web page and run it.
- Analyze the code and derive the used discretizations. Go over the fractional step method. Derive update formulas for each step and explain how boundary conditions are done.
- Add timers to the code and benchmark/profile the code. Analyze your profiling to find performance bottlenecks. Explain the causes/remedies of these.
- Identify visual artifacts (like jags, noise, flickering, aliasing etc..) and explain their causes. (HINT: make hypotheses about what you should expect for low/high values of different parameter values)
- Discuss if the order of steps in the fractional step method has any influence on the final simulation result. Try swapping the order in the code with your expectations.

Assignment

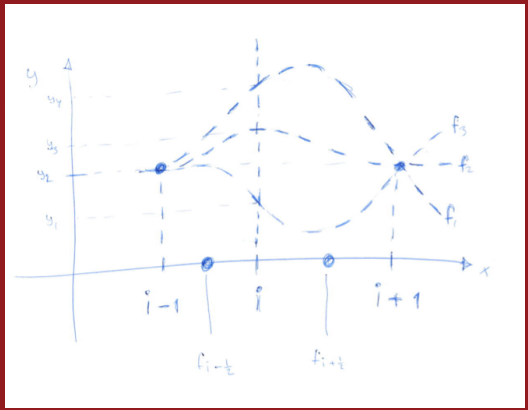
- Make convergence plots of the Gauss-Seidel solver when used for pressure projection and implicit diffusion respectively. Tune the maximum number of iterations to get the best accuracy (for the number of flops used).
- Set up experiments that will illustrate how the accuracy of the pressure projection solution affects the fluid simulation results. Based on your findings formulate a rule of thumb for how much accuracy is desirable.
- Replace the Gauss-Seidel (GS) solver with a different type of solver (see PCG, GMRES, BICG for hints). Tune parameters such that the accuracy of the solution is at least an order of magnitude better than the one you achieved with Gauss-Seidel. How does the visual quality of the simulation compare between GS and your new solver?

Assignment

- Change the boundary condition implementation such that you can simulate a cavity-driven flow by making the top of the box slide to the right. Compare your velocity profiles and streamline plots with those from: "U. Ghia, K.N. Ghia,and C.T. Shin,High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method, Journal of Computational Physics, Volume 48, issue 3, 1982".
- Add inflow and outflow boundary conditions (like $D(u_{\text{vent}}, \mathbf{n}) = k$ where k is the rate of in/outflow in the normal direction to the domain boundary). Use this to add vents to your fluid flow simulation.
- Change the code such that one can make arbitrary solid wall boundaries and test your implementation for correctness.

Assignment

- Find the CD approximations for $\frac{\partial f_1(x_i)}{\partial x}$, $\frac{\partial f_2(x_i)}{\partial x}$, and $\frac{\partial f_3(x_i)}{\partial x}$ shown in the figure



How did staggering improve your results? (Hint: Compare to page 64)

Assignment

- Change the collocated grid into a staggered grid as the one used by “Nick Foster , Ronald Fedkiw, Practical animation of liquids ” (https://doi.org/10.1145/383259.383261) and “Ronald Fedkiw, Jos Stam and Henrik Wann Jensen, Visual Simulation of Smoke ” (https://doi.org/10.1145/383259.383260).
- Replace the semi-Lagrangian time integration method with a MacCormack method and/or QUICK method. See “ Andrew Selle , Ronald Fedkiw , Byungmoon Kim , Yingjie Liu , Jarek Rossignac, An Unconditionally Stable MacCormack Method ” (https://doi.org/10.1007/s10915-007-9166-4) and “Jeroen Molemaker , Jonathan M. Cohen , Sanjit Patel , Jonyong Noh, Low viscosity flow simulations for animation ” (http://dx.doi.org/10.2312/SCA/SCA08/009-018) for details.