

# Multiscale Finite Volume Method

Karl Louis Glänzer / Jonathan Ulmer

August 18, 2025

# Outline

- 1 Standard Diffusion equation
- 2 Example diffusion terms
- 3 Finite Volume 1D
- 4 1D Results
- 5 2D results
- 6 Error

## Abstract

in our work, we solve the diffusion equation (1) with zero Dirichlet boundary and constant flux  $f(x) = 1$ . In order to investigate the effects of the multiscale method on the solution, we introduce a varied sample of diffusivity terms

$$\nabla \cdot (D(x)\nabla c) = f(x) \quad \text{in } \Omega \quad (1)$$

$$c(x) = 0 \quad \text{on } \partial\Omega \quad (2)$$

Our work provides a way to approximate a solution  $c$  to the PDE (1) with the Finite Volume method, and a multiscale adaptation.

# Derivation of the 1D Finite Volume Method I

The Finite Volume method considers the differential equation in Integral form over disjunct ( $Q_i \cap Q_j = \emptyset, i \neq j$ ) reference cells  $Q_i$ ,  $\bigcup_{i=1}^N Q_i = \Omega$  and calculates the integral over them, with an integral over the reference cell boundaries using Stokes integration.

$$\int_{Q_i} \nabla \cdot (D(x) \nabla c) = \int_{Q_i} f(x) \, dx \quad i = 1, \dots, N \quad (3)$$

$$\int_{\partial Q_i} D(x) \nabla c \cdot \vec{n} \, dS = \int_{Q_i} f(x) \, dx \quad i = 1, \dots, N \quad (4)$$

The Finite Volume Method then considers the solution piecewise constant on  $Q$ . This creates discontinuities on the cell boundaries, where the values are not uniquely defined. The Finite Volume method therefore introduces a numerical flux in the Ansatz and solves the integral over the flux instead. Since the assumed solution is constant we approximate the source term

# Derivation of the 1D Finite Volume Method II

$f(\vec{x})$  with its value on the cell center  $x_i$  of  $Q_i$  and calculate the integrals directly.

$$\int_{\partial Q_i} g(c^+, c^-) \cdot \vec{n} dS = \int_{Q_i} f(x) dx \quad i = 1, \dots, N \quad (5)$$

$$\int_{\partial Q_i} g(c^+, c^-) \cdot \vec{n} dS = |Q_i| f(x_i) \quad i = 1, \dots, N \quad (6)$$

## 1D Flux

We employ the flux approximation introduced in the MMM Lecture. Since we only investigated diffusion terms with an analytical representation, we are able to calculate this value directly.

$$g(c^+, c^-) = -D(x^{\frac{1}{2}+}) \frac{c^+ - c^-}{h} \quad (7)$$

Furthermore, we introduce transmissivities  $T_{\pm}$  between both cells.

$$g(c^+, c^-) = T_{\pm} * (c^+ - c^-)$$
$$T_{\pm} = -D(x^{\frac{1}{2}+}) \frac{1}{h}$$

## 2D Flux

We define the flux term in 2 Dimensions very similar to those in one dimension.

$$g_x(c_{i+1,j}, c_{ij}) = -\Delta_y D(x_{i+\frac{1}{2},j}) \frac{c_{i+1,j} - c_{ij}}{\Delta_x} \quad (8)$$

$$g_y(c_{i,j+1}, c_{ij}) = -\Delta_x D(x_{i,j+\frac{1}{2}}) \frac{c_{i,j+1} - c_{ij}}{\Delta_y} \quad (9)$$

and in the same manner we introduce 2D transmissions  $T_{i+1,j}^x, T_{ij+1}^y$

$$g_x(c_{i+1,j}, c_{ij}) = T_{i+1,j}^x (c_{i+1,j} - c_{ij})$$

$$g_y(c_{i,j+1}, c_{ij}) = T_{ij+1}^y (c_{i,j+1} - c_{ij})$$

## Image

We implemented our finite Volume solver on a rectangular grid. therefore the normals on the boundaries are constant, and the flux integral (6) simplifies to a sum

$$\int_{\partial Q_i} g(c^+, c^-) \cdot \vec{n} dS = |Q_i| f(x_i) \quad i = 1, \dots, N$$
$$\sum_{n \in \partial Q} \vec{g}(c_{i+j+1}, c_{i+j}) \cdot \vec{n} = |Q_i| \bar{f}(x_i)$$

## 1D

- In one dimension there are only two outward normals  $n \in \{-1, 1\}$ ,
- we use the 1D flux (7)



## 2D replace with image

- In two dimensions there are four outward cell normals

$$n_{\text{north}} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$n_{\text{south}} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

$$n_{\text{east}} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$n_{\text{west}} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

- we use the 2D flux (8)



We investigate of single and multiscale solvers with different Diffusion functions, that we introduce in the following sections

Since the Aim of multiscale Finite Volume, is to improve the results for highly fluctuating diffusivities, we test with the following oscillating function

## Code

```
def oscillation(x, eps = 0.1):  
    return 1 / (2+1.9 * np.cos(2 * np.pi* x / eps))
```

## Diffusivity

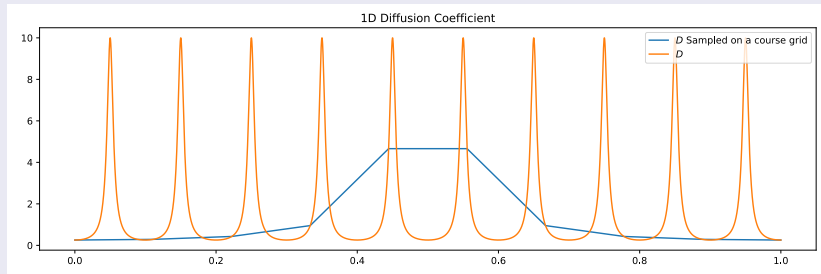


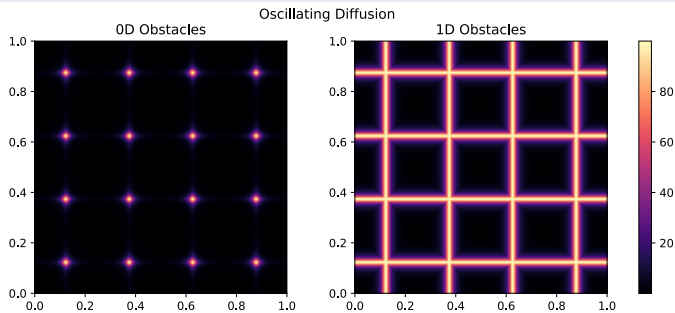
Figure: Oscillating Diffusivity for 1D finite volume method

## Code

```
def osc2D_point(x,y , eps = 0.25):  
    return oscillation(x, eps=eps) * oscillation(y, eps=eps)  
def osc2D_line(x,y , eps = 0.25):  
    return np.maximum(oscillation(x, eps=eps) , oscillation(y,  
        ↪ eps=eps))
```

None

## Diffusion



# 2D Box Condition I

To test numerical stability of our methods we introduce a box constrain condition, that traps some concentration in the center.

## Diffusivity

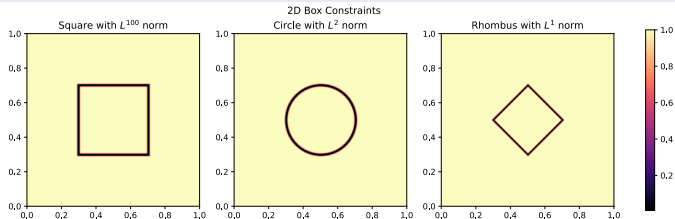


Figure: Constraints restricting flow from the center of the Domain



# Program Structure I

For convenience in Explanation and Execution, we bundle all required information for solving a 1D system into a python class, which is structured as follows

# Program Structure II

## Class Structure

```
class FVSolver:
    N : int
    resolution : int
    h : np.float64
    x : NDArray[np.float64]
    D : Callable
    f : NDArray[np.float64]
    c : NDArray[np.float64]
    micro_basis : NDArray[np.float64]
    _T : NDArray[np.float64]

<<Init>>
<<Assemble Matrix>>
<<Boundary>>
<<Solve>>
<<Microscale Transmissions>>
<<Reconstruct Microscale Solution>>
```

# Program Structure III

## Initialization

```
def __init__(self , N :int , D :Callable , domain=(0.,1.))->None:
    self.h = (domain[1] - domain[0]) / (N-1)
    self.N = N
    self.D = D
    self.x = np.linspace(domain[0] , domain[1] , N)
    self._T = -1/self.h * D((self.x[:-1] + self.x[1:])*0.5)
    self.f = self.h* np.ones(N)
```

## Solving

```
def solve(self):
    self.c = spsolve(self._A.tocsr() , self.f)
    return self.c
```

## Boundary

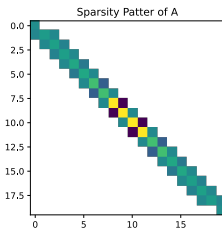
```
def set_boundary(self , bc=(0.,0.)):  
    self.f[0] = bc[0]  
    self.f[-1] = bc[1]
```

## Assembly of the linear system

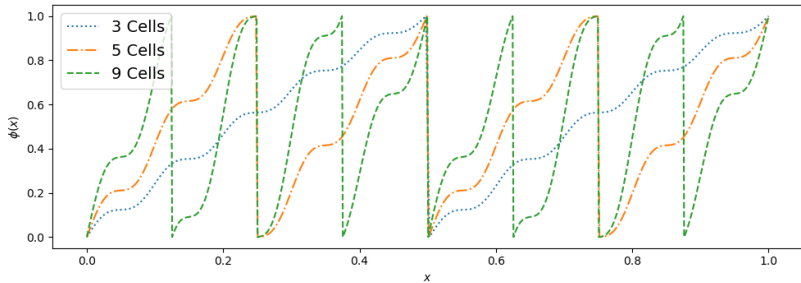
## Matrix Assembly

```
def assemble_matrix(self)-> None:
    diagp1 = np.zeros(self.N)
    diagp1[2:] = self._T[1:]
    diagm1 = np.zeros(self.N)
    diagm1[:-2] = self._T[:-1]
    diag0 = np.ones(self.N)
    diag0[1:-1] = -1 * (self._T[1:] + self._T[:-1])
    self._A = spdiags([diagm1 , diag0 , diagp1] , np.array( [-1, 0,
↪ 1] ))
```

## Sparsity Pattern of the linear system

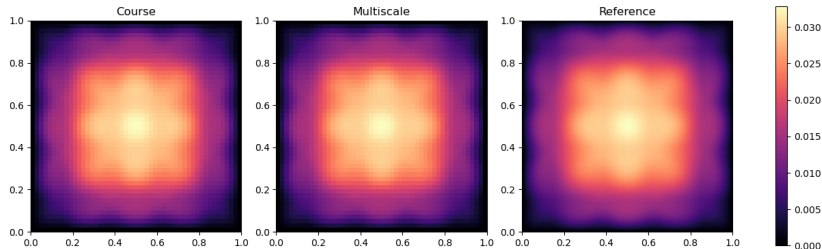


# Microscale Basis

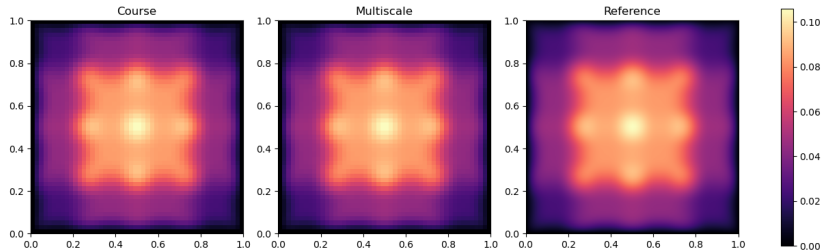


# Oscillations I

Line Diffusion with 4 Spikes with 50×50 Grid

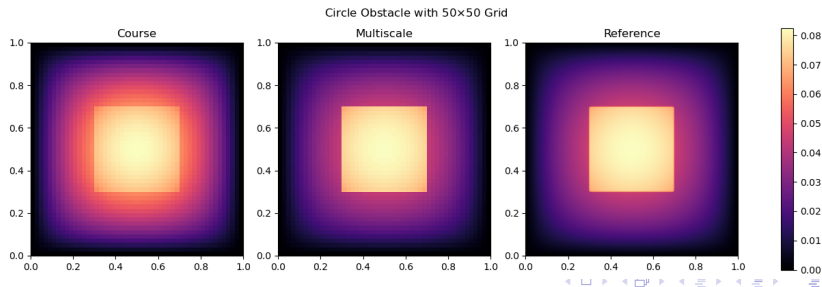
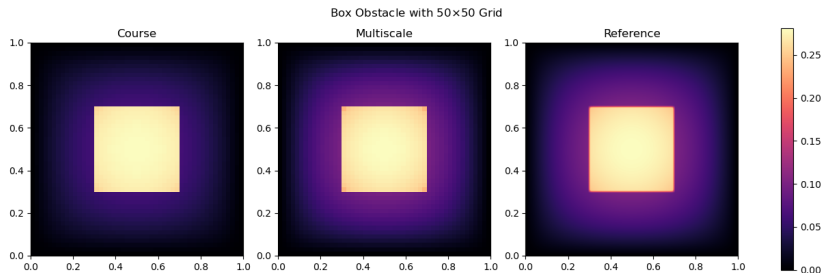


Point Diffusion with 4 Spikes with 50×50 Grid

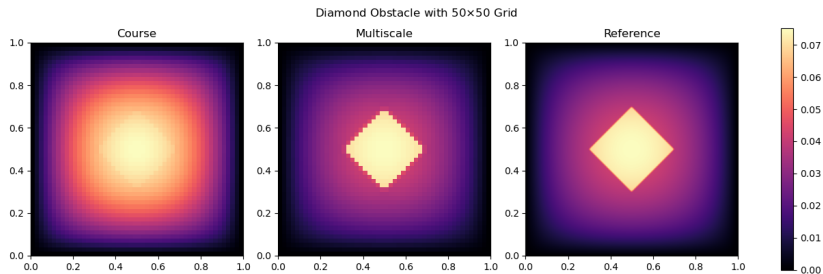




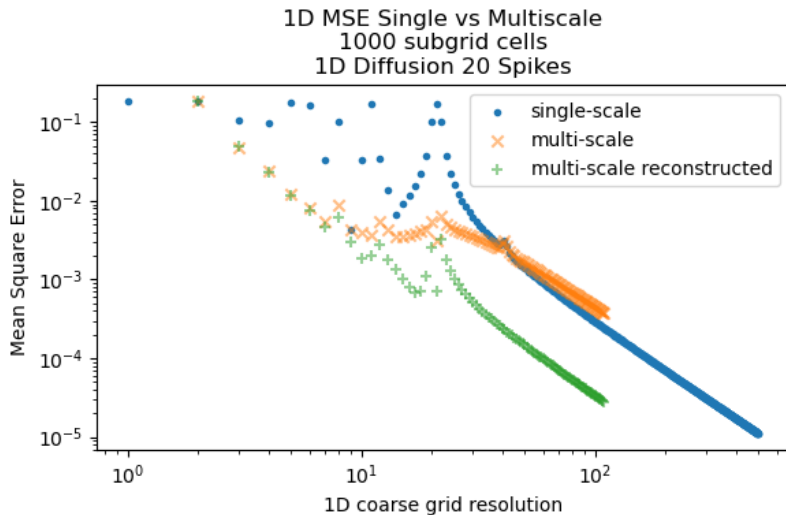
# Box Conditions I



# Box Conditions II

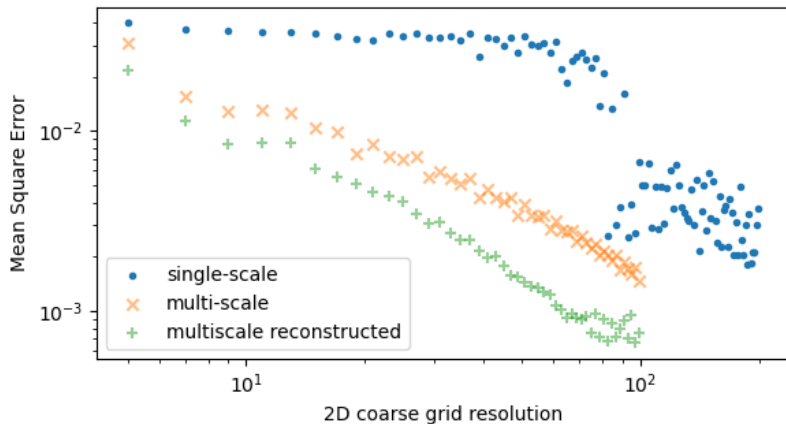


# 1D Error I



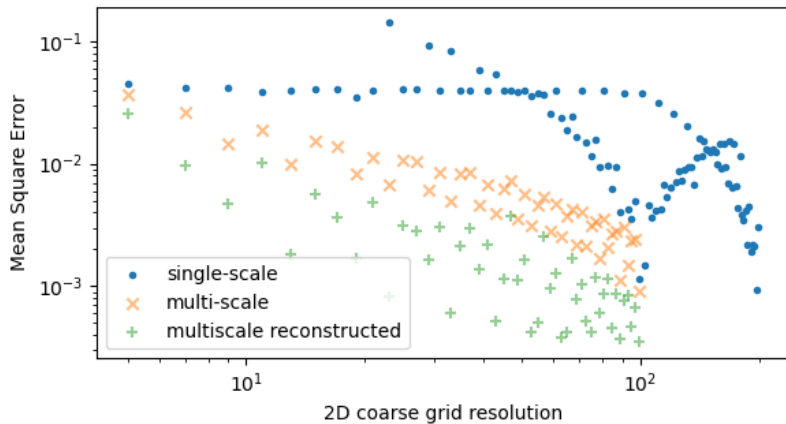
# 2D Error I

2D MSE Single vs Multiscale  
1000 subgrid cells  
Circle Diffusion



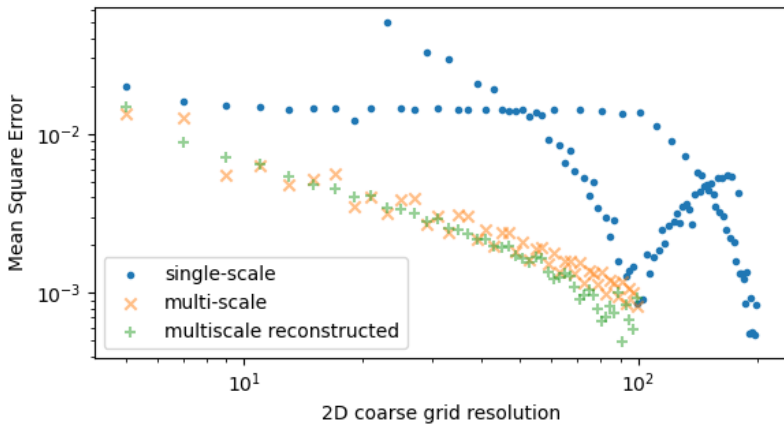
# 2D Error II

2D MSE Single vs Multiscale  
1000 subgrid cells  
Box Diffusion



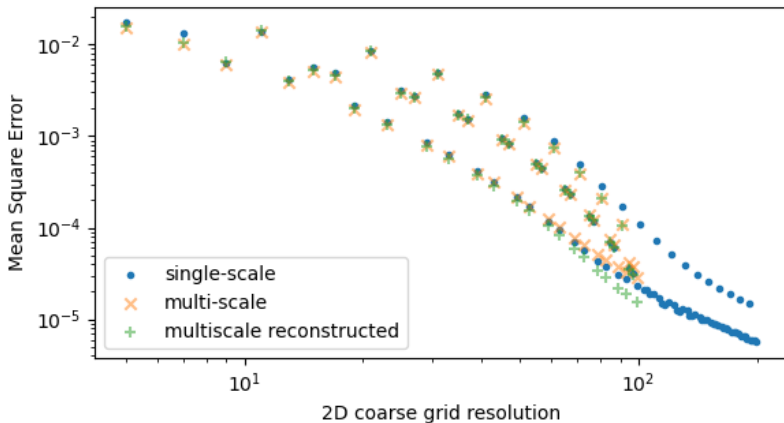
# 2D Error III

2D MSE Single vs Multiscale  
1000 subgrid cells  
Diamond Diffusion



# 2D Error IV

2D MSE Single vs Multiscale  
1000 subgrid cells  
Line Diffusion 5 Spikes



# 2D Error V

2D MSE Single vs Multiscale  
1000 subgrid cells  
Point Diffusion 5 Spikes

