# Multiscale Finite Volume Method

Karl Louis Glänzer / Jonathan Ulmer

August 18, 2025

# Outline

# Numeral Problem I

## 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

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

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

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.

$$\int_{Q_i} \nabla \cdot (D(x)\nabla c) = \int_{Q_i} f(x)\,\mathrm{d}x \qquad i = 1, \ldots, N \tag{3}$$

$$\int_{\partial Q_i} D(x)\nabla c \cdot \vec{n}\,\mathrm{d}S = \int_{Q_i} f(x)\,\mathrm{d}x \qquad i = 1, \ldots, N \tag{4}$$

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.

$$\int_{\partial Q_i} g(c^+, c^-) \cdot \vec{n} \, \mathrm{d}S = \int_{Q_i} f(x) \, \mathrm{d}x \qquad i = 1, \ldots, N \tag{5}$$

$$\int_{\partial Q_i} g(c^+, c^-) \cdot \vec{n} \, \mathrm{d}S = |Q_i| f(x_i) \qquad i = 1, \ldots, N \tag{6}$$

# Numerical flux approximation I

## 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.

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

Furthermore, we introduce transmissitivities $T_\pm$ between both cells.

$$g(c^+, c^-) = T_\pm * \left(c^+ - c^-\right)$$

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

# Numerical flux approximation II

## 2D Flux

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

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

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

and in the same manner we introduce 2D transmissions $T^x_{i+1j}$, $T^y_{ij+1}$

$$g_x(c_{i+1j}, c_{ij}) = T^x_{i+1j} (c_{i+1j} - c_{ij})$$
$$g_y(c_{ij+1}, c_{ij}) = T^y_{ij+1} (c_{i+1j} - c_{ij})$$

# Linearization I

## 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

$$\int_{\partial Q_i} g(c^+, c^-) \cdot \vec{n} \, \mathrm{d}S = |Q_i| f(x_i) \qquad i = 1, \ldots, N$$

$$\sum_{n \in \partial Q} \vec{g}(c_{i+j+1}, c_{i+j}) \cdot \vec{n} = |Q_i| \overline{f}(x_i)$$

## 1D

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

# Linearization II

## 2D replace with image

- In two dimensions there are four outward cell normals

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

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

- we use the 2D flux (8)

# Multiscale I

# Diffusion I

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

# 1D I

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

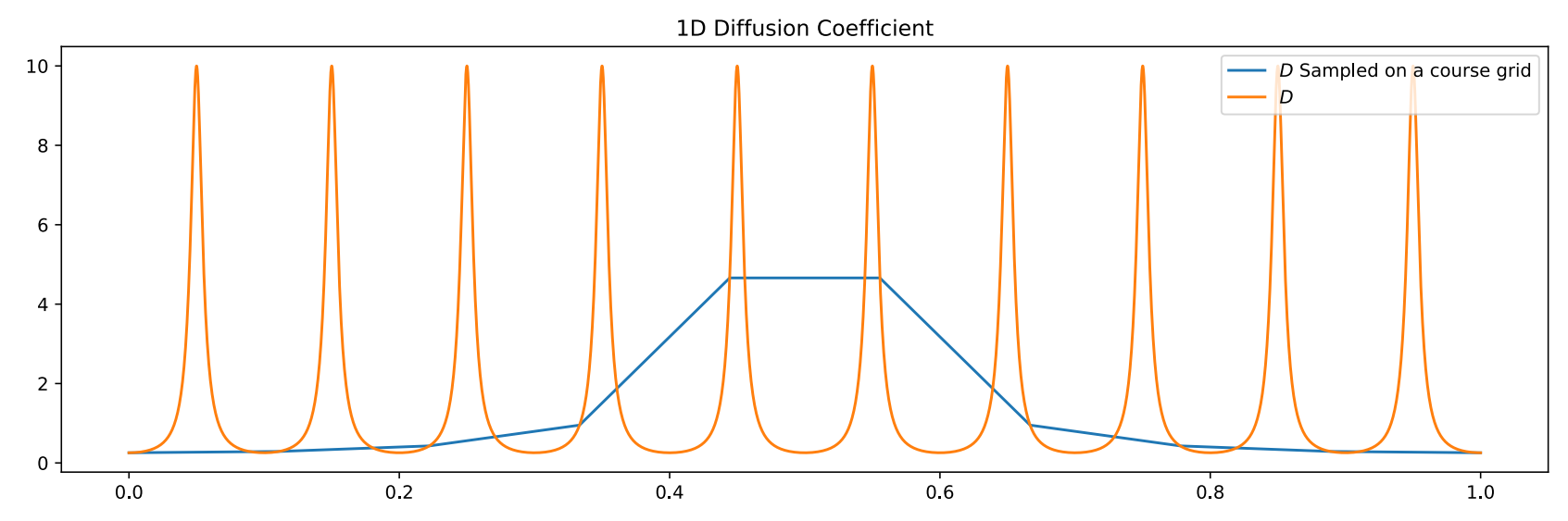



Figure: Oscillating Diffusivity for 1D finite volume method

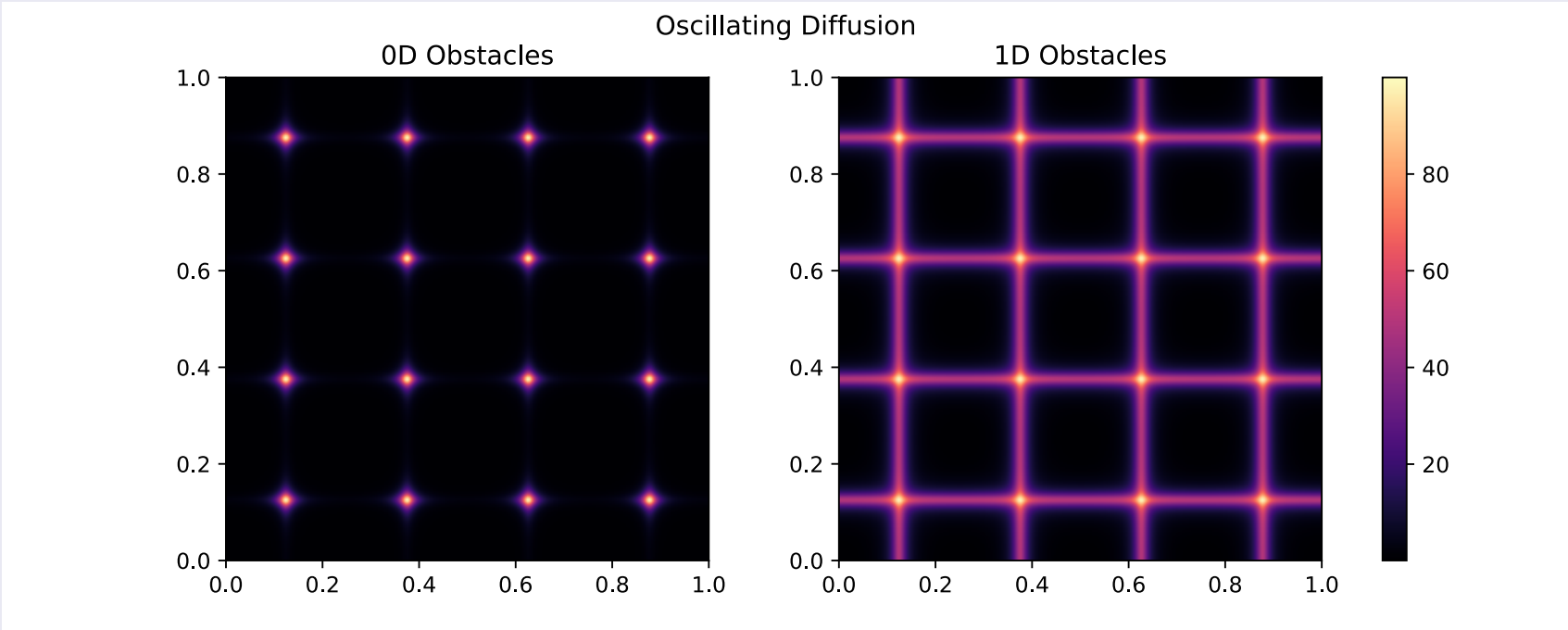
# 2D Oscillation I

## 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 oscillation(x, eps=eps) + oscillation(y, eps=eps)
```

# 2D Oscillation II

## 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

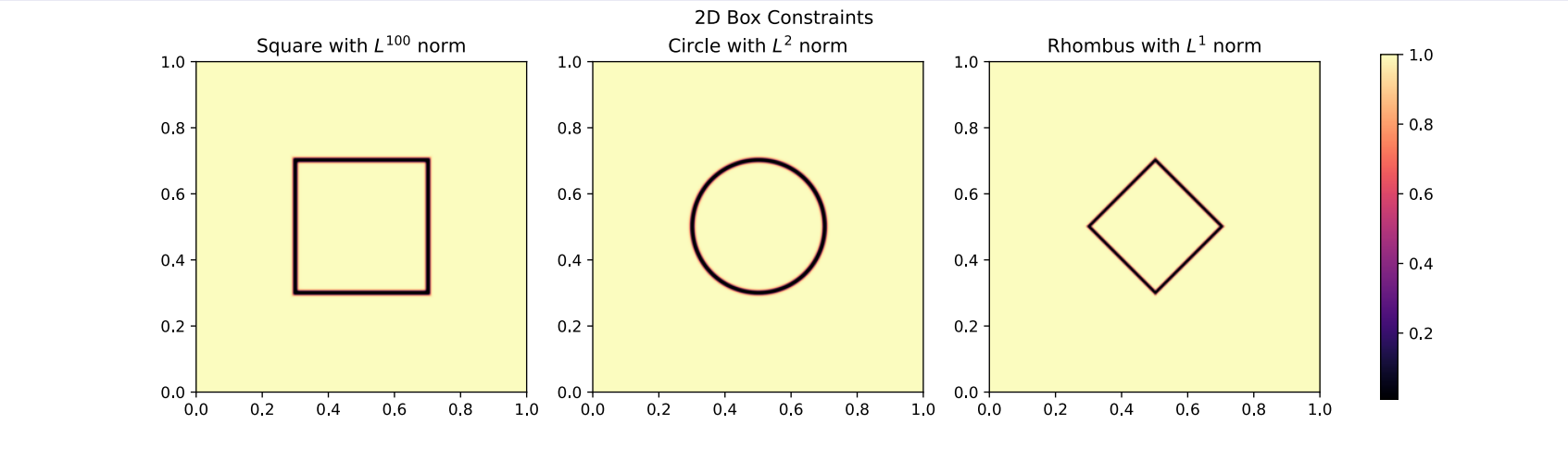



Figure: Constraints restricting flow from the center of the Domain

# 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
```

# Program Structure IV

## Boundary

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

## Assembly of the linear system

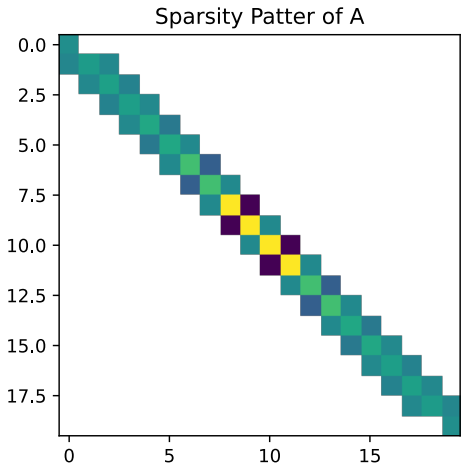
# Program Structure V

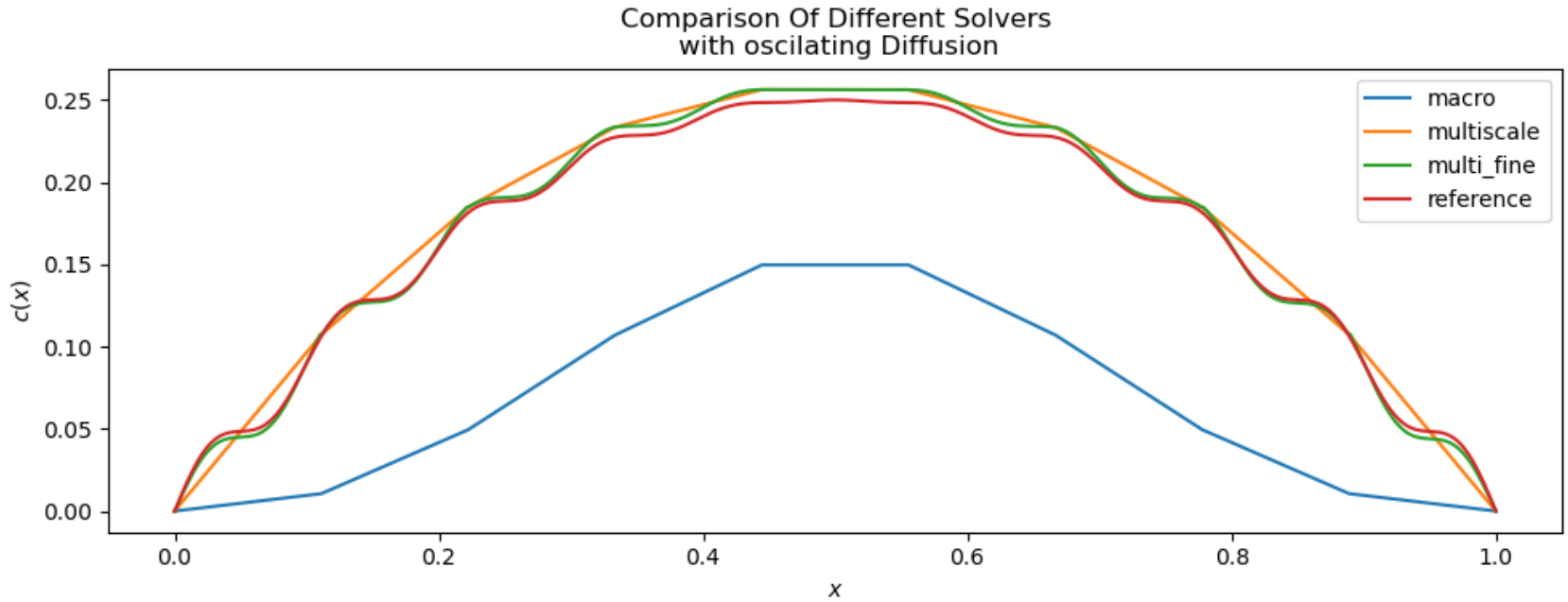
## Matrix Assembly

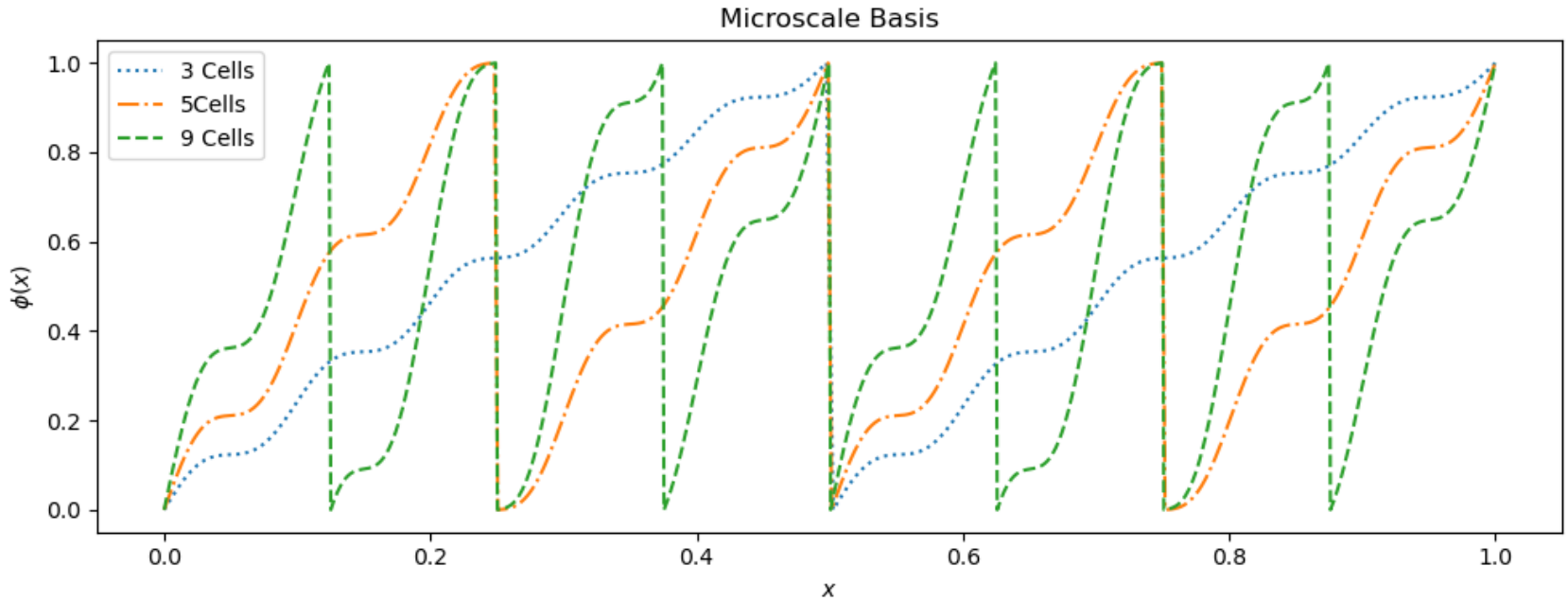
```
def assemble_matrix(self)-> None:
    diagp1 = np.zeros(self.N)
    diagp1[2:] =  self._T[1:]
    diagm1 = np.zeros(self.N)
    diagm1[:-2] =  self._T[:-1]
    diag0 = np.ones(self.N)
    diag0[1:-1] = -1 * (self._T[1:] + self._T[:-1])
    self._A = spdiags([diagm1 , diag0 , diagp1] , np.array( [-1,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

# Program Structure VI

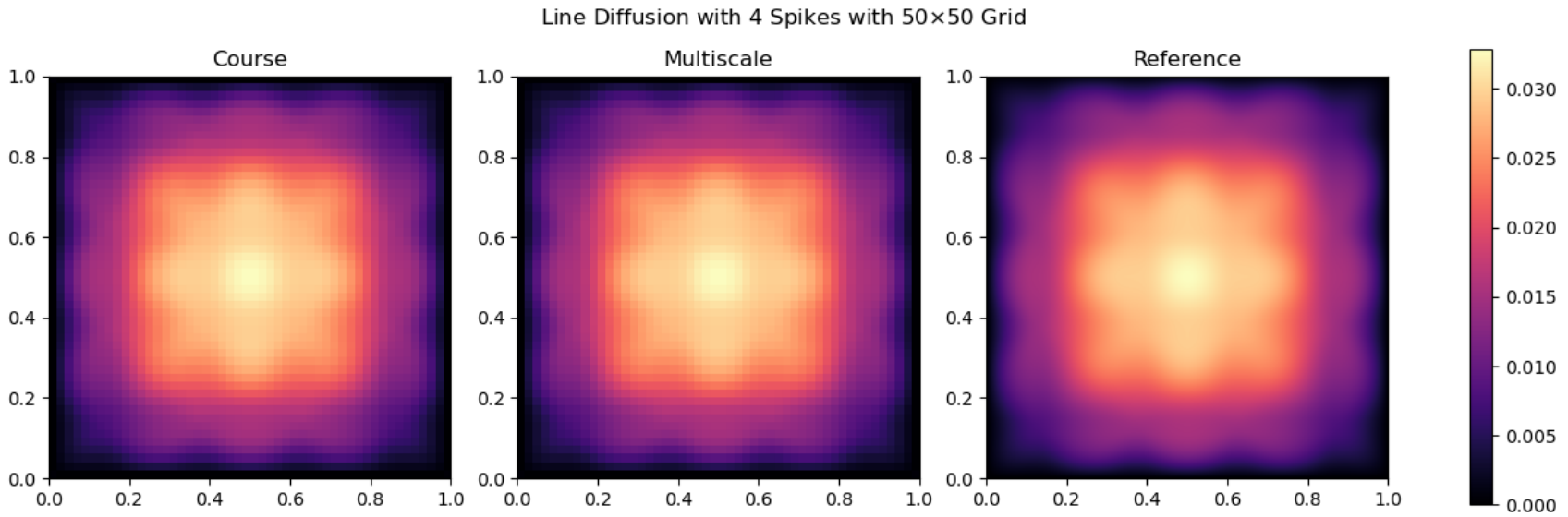
## Sparsity Pattern of the linear system

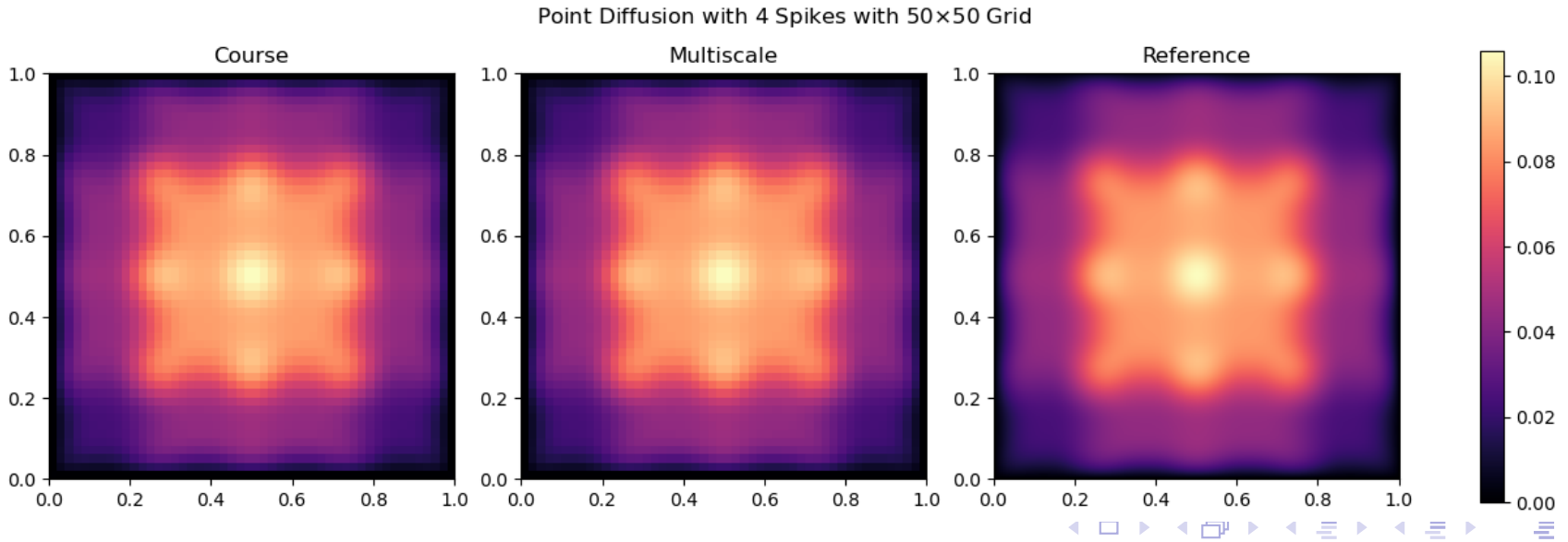



Comparison Of Different Solvers
with oscilating Diffusion
macro
multiscale
multi_fine
reference
c(x)
x
0.00
0.05
0.10
0.15
0.20
0.25
0.0
0.2
0.4
0.6
0.8
1.0

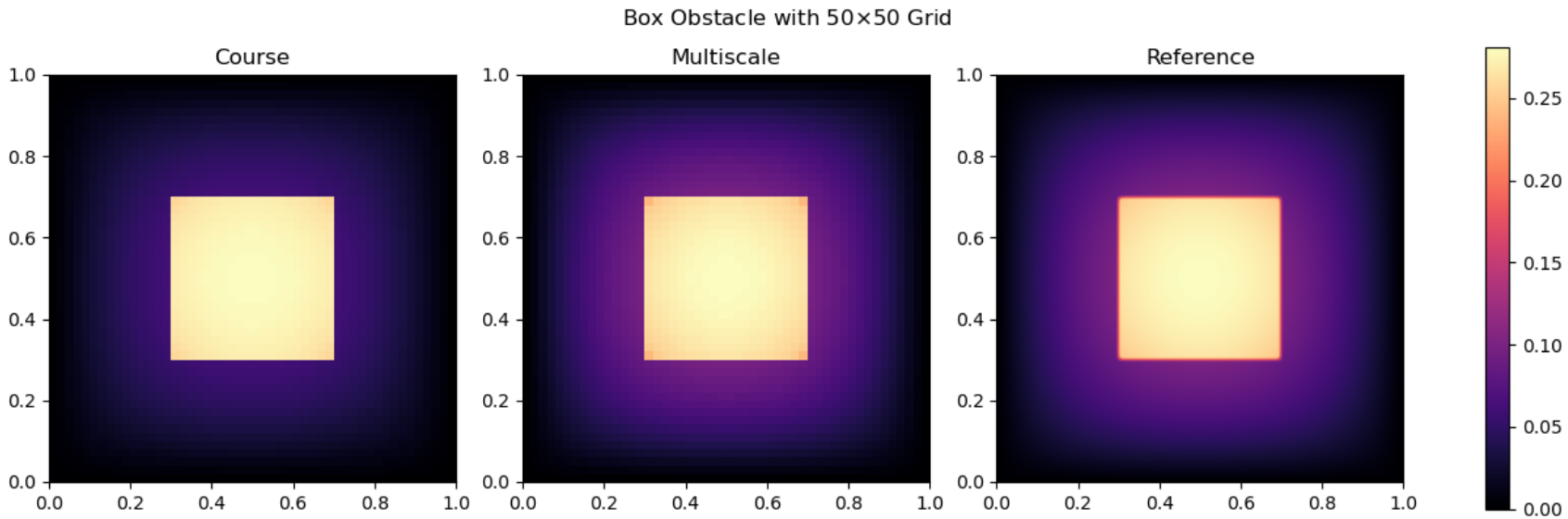

Microscale Basis
3 Cells
5Cells
9 Cells
ϕ(x)
x
0.0
0.2
0.4
0.6
0.8
1.0

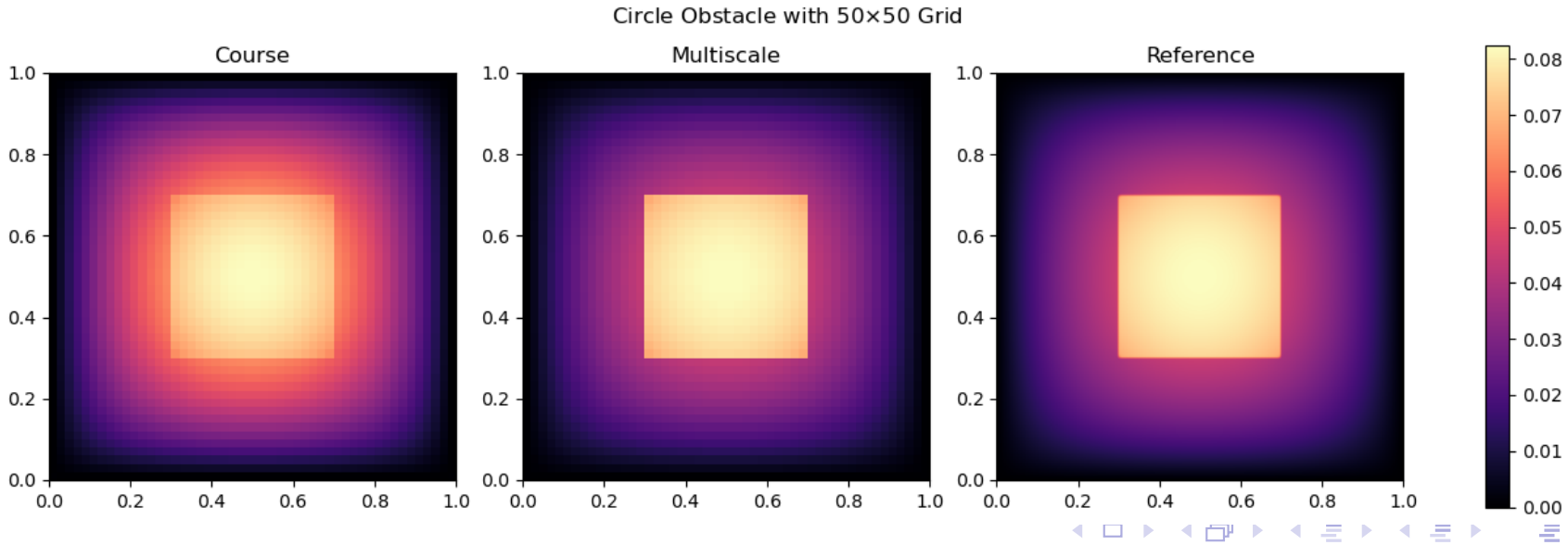
# Oscillations I



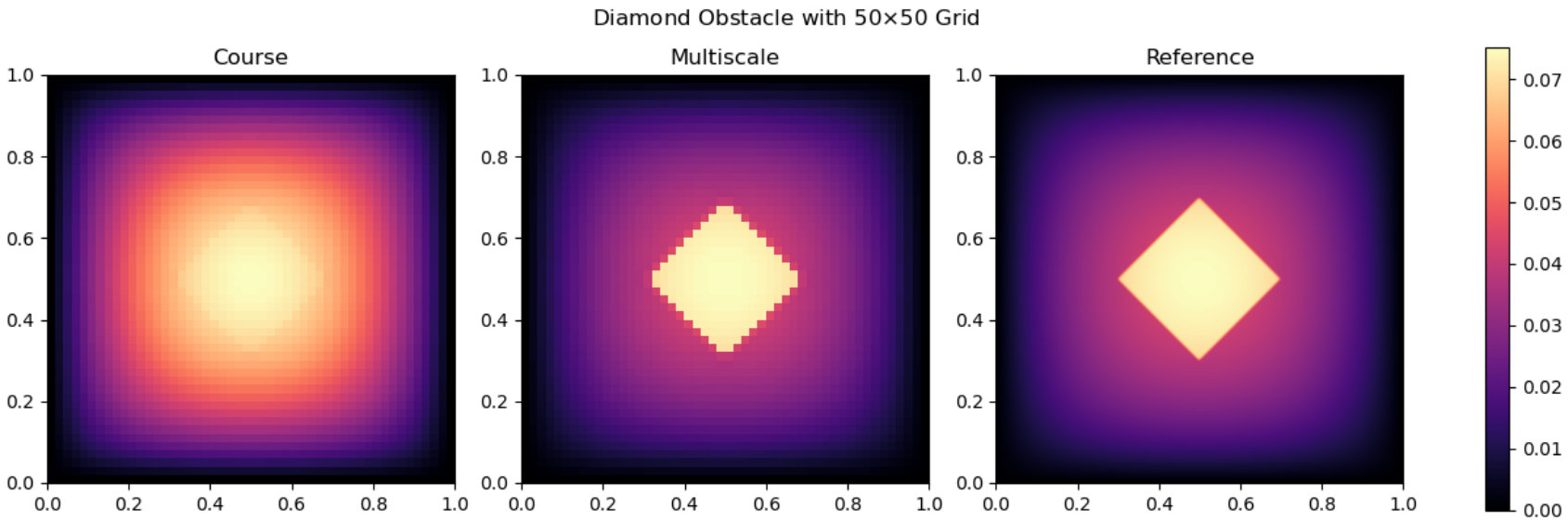



# Box Conditions I

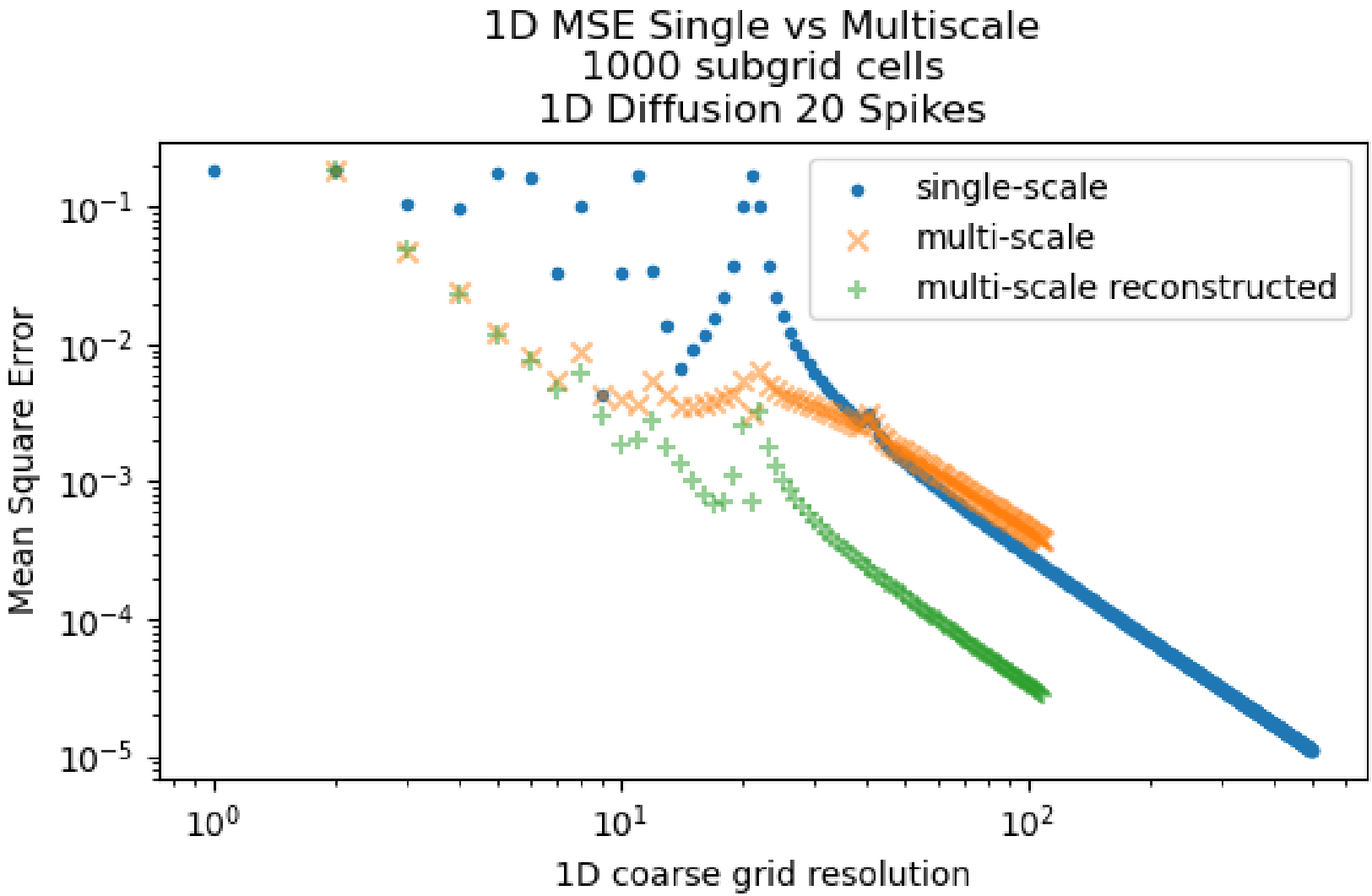


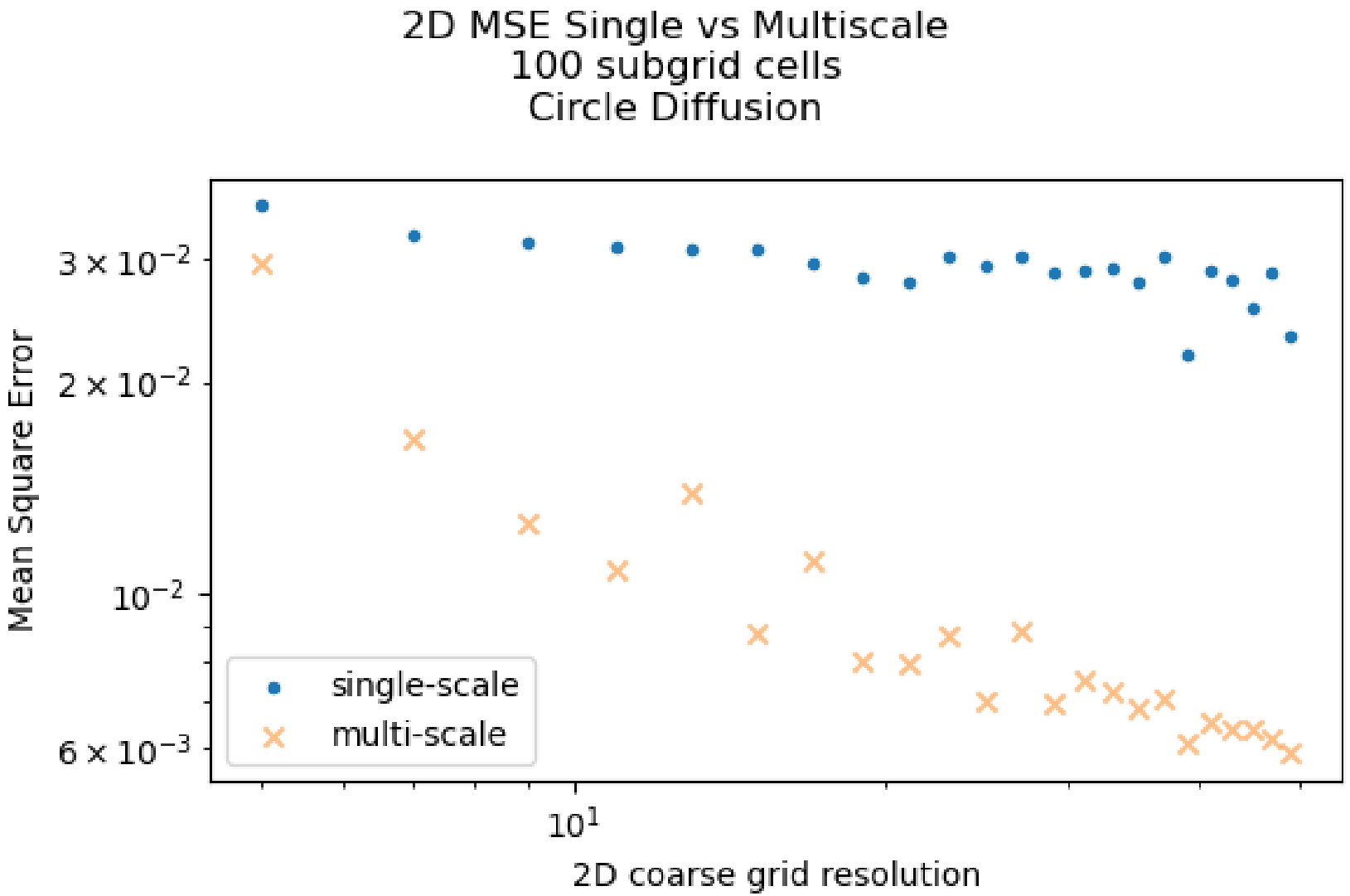
# Box Conditions II



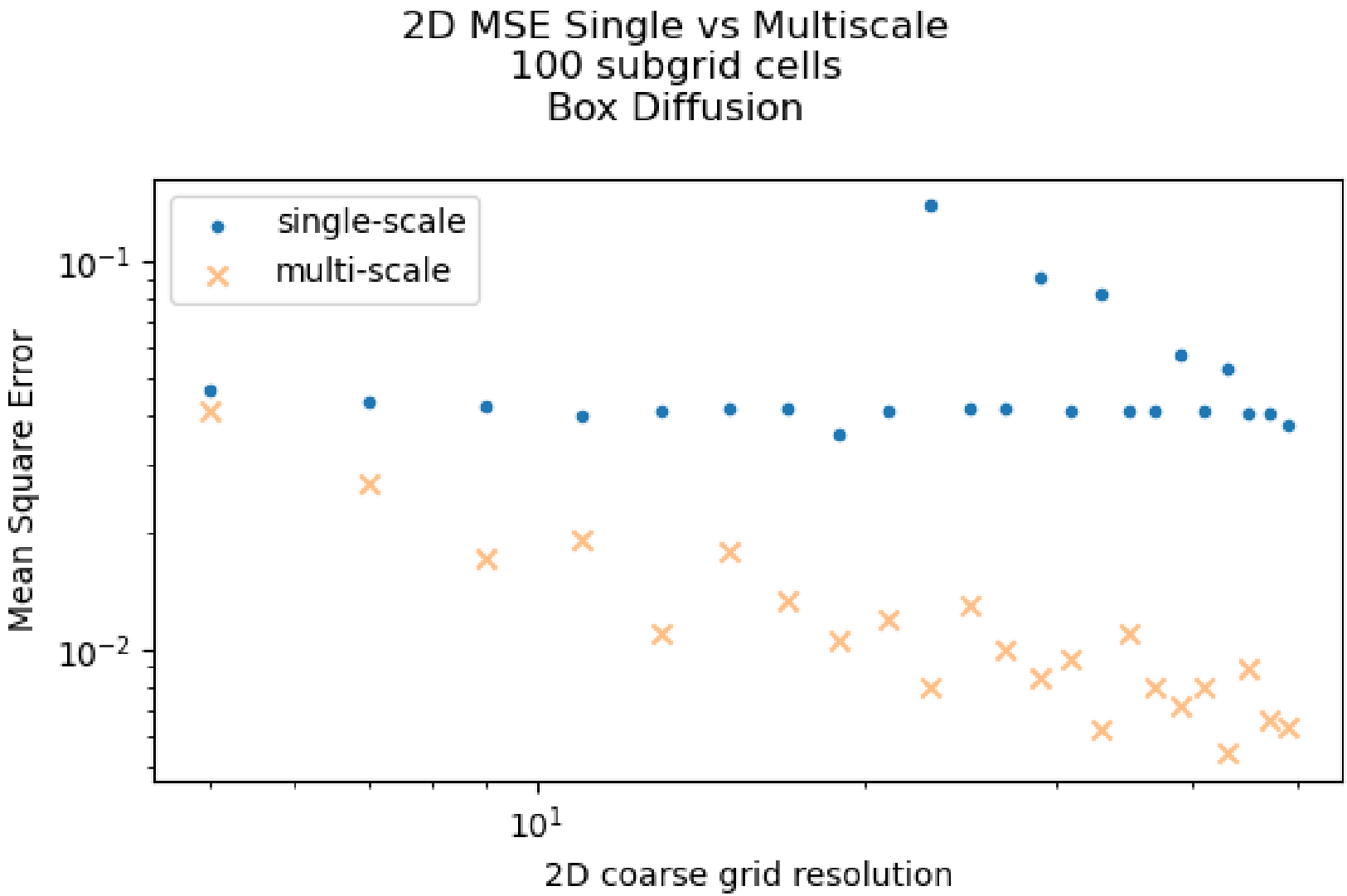
# 1D Error I


1D MSE Single vs Multiscale
1000 subgrid cells
1D Diffusion 20 Spikes
single-scale
multi-scale
multi-scale reconstructed
Mean Square Error
$10^{-1}$
$10^{-2}$
$10^{-3}$
$10^{-4}$
$10^{-5}$
$10^{0}$
$10^{1}$
$10^{2}$
1D coarse grid resolution

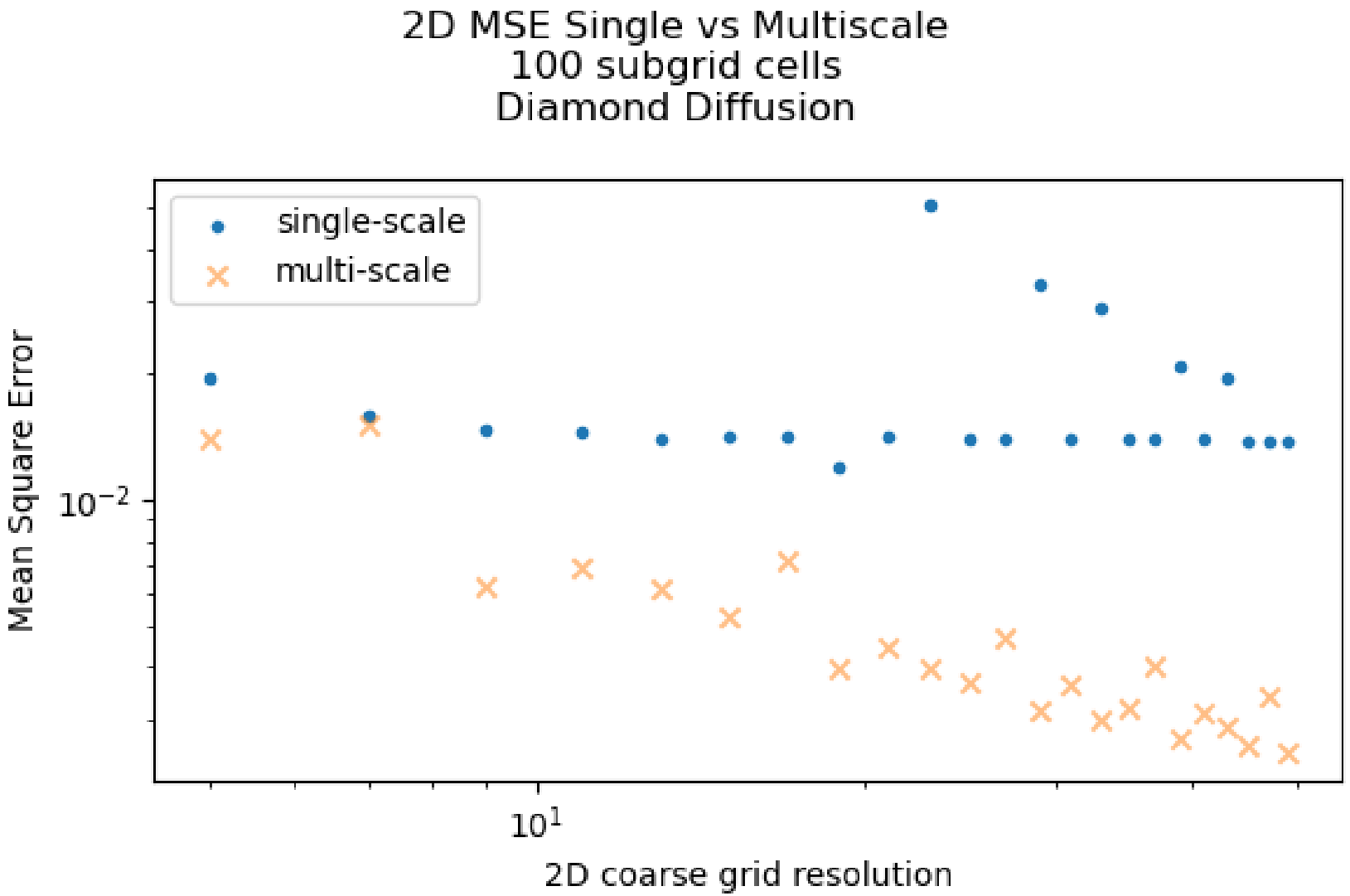
# 2D Error I


2D MSE Single vs Multiscale
100 subgrid cells
Circle Diffusion
Mean Square Error
$3 \times 10^{-2}$
$2 \times 10^{-2}$
$10^{-2}$
$6 \times 10^{-3}$
single-scale
multi-scale
$10^{1}$
2D coarse grid resolution

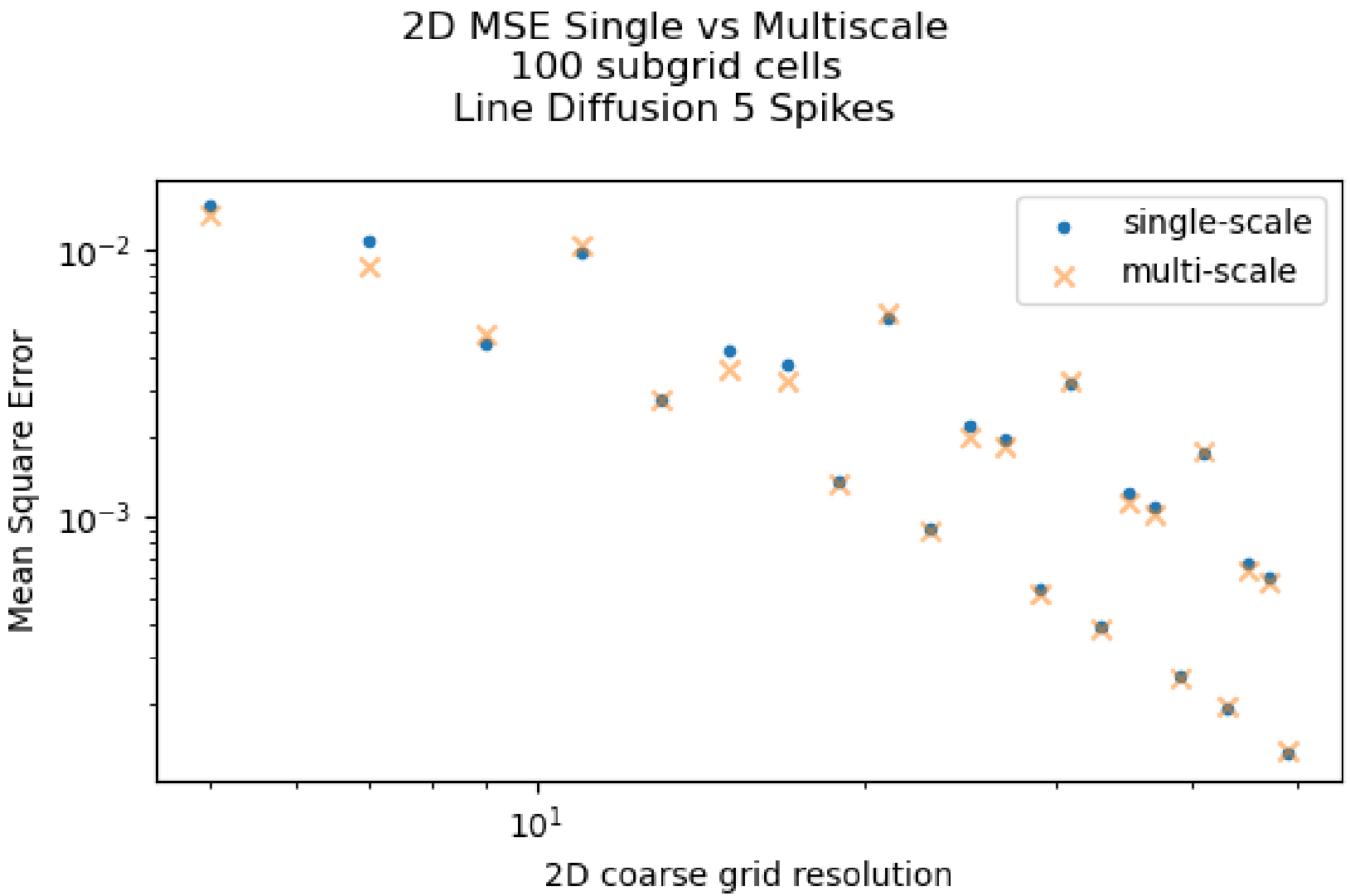
# 2D Error II


2D MSE Single vs Multiscale
100 subgrid cells
Box Diffusion
single-scale
multi-scale
$10^{-1}$
$10^{-2}$
Mean Square Error
$10^{1}$
2D coarse grid resolution

# 2D Error III


2D MSE Single vs Multiscale
100 subgrid cells
Diamond Diffusion
single-scale
multi-scale
Mean Square Error
$10^{-2}$
$10^{1}$
2D coarse grid resolution

# 2D Error IV


2D MSE Single vs Multiscale
100 subgrid cells
Line Diffusion 5 Spikes
single-scale
multi-scale
Mean Square Error
$10^{-2}$
$10^{-3}$
$10^{1}$
2D coarse grid resolution