Heat Diffusion

Introduction

Heat mapping in Python refers to the process of visualizing data in the form of a heatmap, where values are represented by color intensity on a grid. It is commonly used to display two-dimensional data, such as correlations or temperature distributions, making it easy to observe patterns, trends, and relationships. Python libraries like Matplotlib, Seaborn, and Plotly offer built-in functions to create heatmaps. The data is usually represented in a matrix form, where each value corresponds to a specific location on the grid, and color gradients are applied to represent the magnitude of the values. This visualization technique is widely used in fields like data science, machine learning, and environmental modeling to quickly interpret large datasets.

The process of simulating heat diffusion in a two-dimensional space is described using matrix multiplication. The heat equation, which models the diffusion of heat, is extended to two dimensions. This equation involves partial derivatives with respect to both spatial dimensions (horizontal and vertical) and time. The space is divided into small intervals, represented by discrete steps in the horizontal and vertical directions, as well as time. By breaking up the space and time into smaller steps, the heat equation can be solved iteratively over time, updating the temperature at each grid point based on the temperatures of its neighboring points.

The update rule describes how the temperature at each point on the grid is updated over time using a method called finite difference. This method takes an average of the temperatures at neighboring grid points in both the horizontal and vertical directions, along with the current point itself. A small parameter is included to help control how heat diffuses across the grid. Matrix multiplication is central to the computation, where the grid of temperature values is treated as a matrix, and the update step is represented as a matrix-vector multiplication. Boundary conditions are also taken into account during the update, ensuring that the edges of the grid either stay constant or allow heat to escape, depending on the specific situation.

Methods

We will look at four methods to simulate 2-dimensional heat diffusion: matrix multiplication, sparse matrix in JAX, direct operation in numpy, and with JAX itself. First, let's look at matrix multiplication.

Matrix Multiplication

As in the linear algebra lecture, let’s use matrix-vector multiplication to simulate the heat diffusion in the 2D space. The vector here is created by flattening the current solution. Each iteration of the update is given by the following code:

```
# Function to advance the heat equation by one timestep
def advance_time_matvecmul(A, u, epsilon):
    """
    Advances the simulation by one timestep using matrix-vector
multiplication.
    """
    N = u.shape[0]
    u = u + epsilon * (A @ u.flatten()).reshape((N, N))
    return u
```

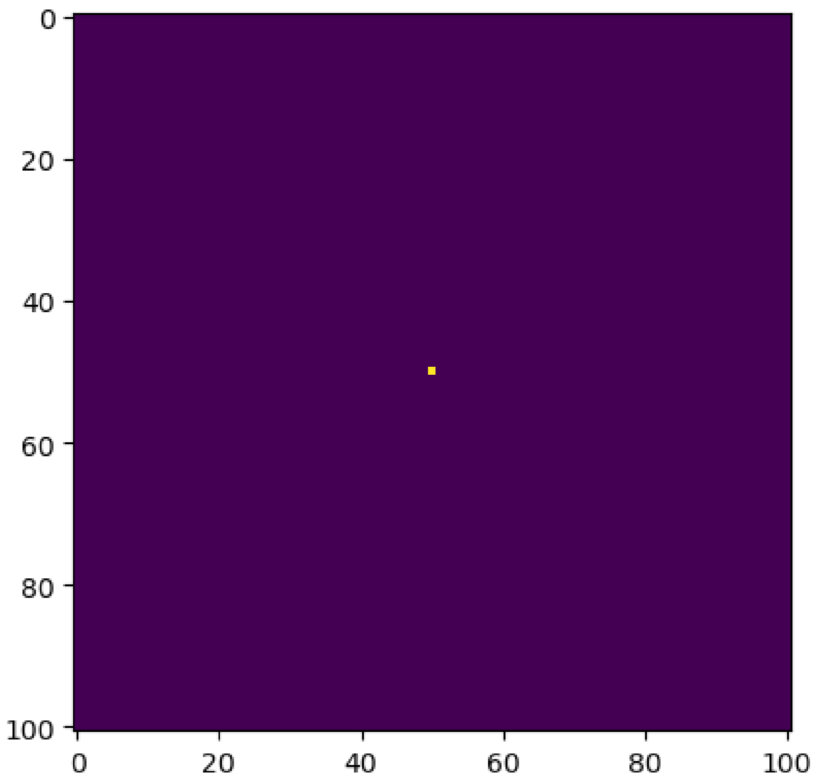
Below is the source code for the matrix multiplication method.

```
import numpy as np
import inspect
from heat_equation import get_A

source_code = inspect.getsource(get_A)
print(source_code)
```

```
def get_A(N):
    """
    This function constructs the 2D finite difference matrix for heat
diffusion simulation.

    Argumentss:
        N (int): Grid size (N x N)

    Returns:
        A (np.ndarray): The finite difference matrix of size (N^2 x
N^2)
    """
    n = N * N

    # Create diagonals
    diagonals = [
        -4 * np.ones(n),       # Main diagonal
        np.ones(n-1),          # First superdiagonal
        np.ones(n-1),          # First subdiagonal
        np.ones(n-N),          # Nth superdiagonal
        np.ones(n-N)           # Nth subdiagonal
    ]

    # Fix boundary conditions by setting specific elements to zero
    # For the first superdiagonal: set elements at the right boundary
to zero
    diagonals[1][(N-1)::N] = 0

    # For the first subdiagonal: set elements at the left boundary to
zero
    diagonals[2][(N-1)::N] = 0

    # Construct the sparse matrix A using numpy's diag function
    A = np.diag(diagonals[0]) + \
        np.diag(diagonals[1], 1) + \
        np.diag(diagonals[2], -1) + \
        np.diag(diagonals[3], N) + \
        np.diag(diagonals[4], -N)

    return A
```

Below is a simple example where we set up the parameters and construct the initial condition. In this scenario, there is only oen unit of heat at the midpoint. The graph below also visualizes this.

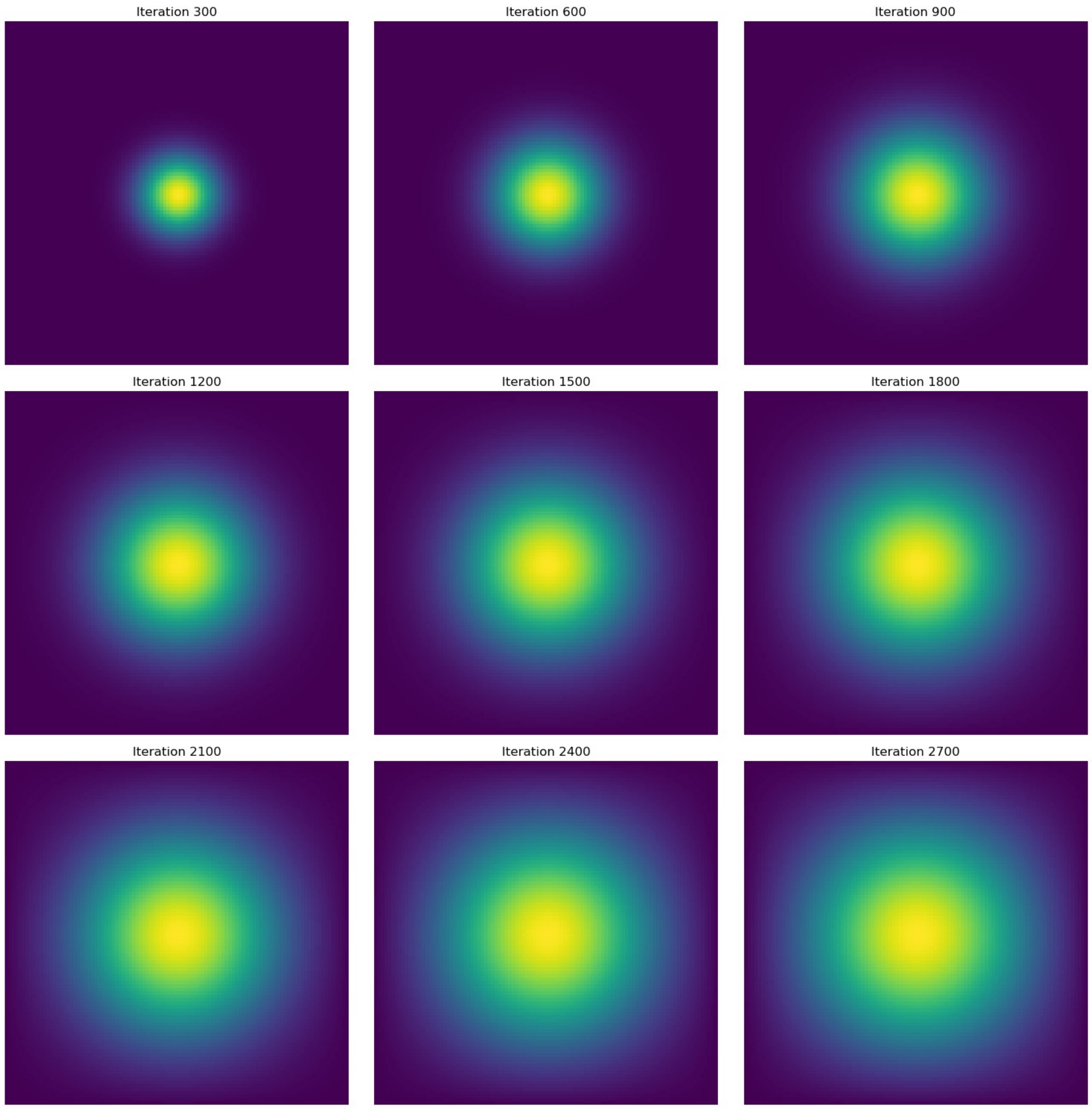
```
import matplotlib.pyplot as plt

# Set the Parameters
N = 101
epsilon = 0.2
# construct initial condition: 1 unit of heat at midpoint.
u0 = np.zeros((N, N))
u0[int(N/2), int(N/2)] = 1.0
plt.imshow(u0)
n = N * N
diagonals = [-4 * np.ones(n), np.ones(n-1), np.ones(n-1), np.ones(n-
N), np.ones(n-N)]
diagonals[1][(N-1)::N] = 0
diagonals[2][(N-1)::N] = 0
A = np.diag(diagonals[0]) + np.diag(diagonals[1], 1) +
np.diag(diagonals[2], -1) + np.diag(diagonals[3], N) +
np.diag(diagonals[4], -N)
```



Now, let's see approximately how long it takes for the following parameters: N = 100, epsilon = 0.1, and the number of iterations being 2700.

```python
# Main function to run the simulation
def run_simulation(N, epsilon, num_iterations):
    """
    Runs the heat diffusion simulation and visualizes snapshots in a 
3x3 grid.
    """
    A = get_A(N)  # Get the finite difference matrix A
    u = np.zeros((N, N))  # Initial heat distribution (start with 0 
heat everywhere)
    u[N//2, N//2] = 1  # Initial heat source at the center

    # Create a figure with a 3x3 grid of subplots
    fig, axes = plt.subplots(3, 3, figsize=(15, 15))

    # Select snapshots at every 300 iterations
    snapshot_indices = [i for i in range(300, num_iterations + 1, 
300)][:9]  # Ensure exactly 9 snapshots

    snapshot_count = 0  # To track which subplot to update
    for t in range(num_iterations + 1):
        u = advance_time_matvecmul(A, u, epsilon)  # Update the grid

        # If current iteration matches a snapshot index, plot it in 
the grid
        if t in snapshot_indices:
            row, col = divmod(snapshot_count, 3)  # Get subplot 
position
            axes[row, col].imshow(u, cmap='viridis', 
interpolation='nearest')
            axes[row, col].set_title(f'Iteration {t}')
            axes[row, col].axis('off')  # Hide axes for clarity
            snapshot_count += 1

    plt.tight_layout()
    plt.show()

# Run the simulation with specified parameters
N = 100  # Grid size (100x100 grid)
epsilon = 0.1  # Stability constant
num_iterations = 2700  # Number of iterations

import time

# Record the start time
start_time = time.time()
run_simulation(N, epsilon, num_iterations)
```

```python
# Record the end time
end_time = time.time()
```



```python
# Calculate the time difference
elapsed_time = end_time - start_time
print(f"Time taken for execution: {elapsed_time} seconds")
```

```
Time taken for execution: 95.26415205001831 seconds
```

As we can see by the code chunk above, the time taken to execute the matrix multiplication method is approximately 95.2642 seconds.

Sparse Matrix in JAX

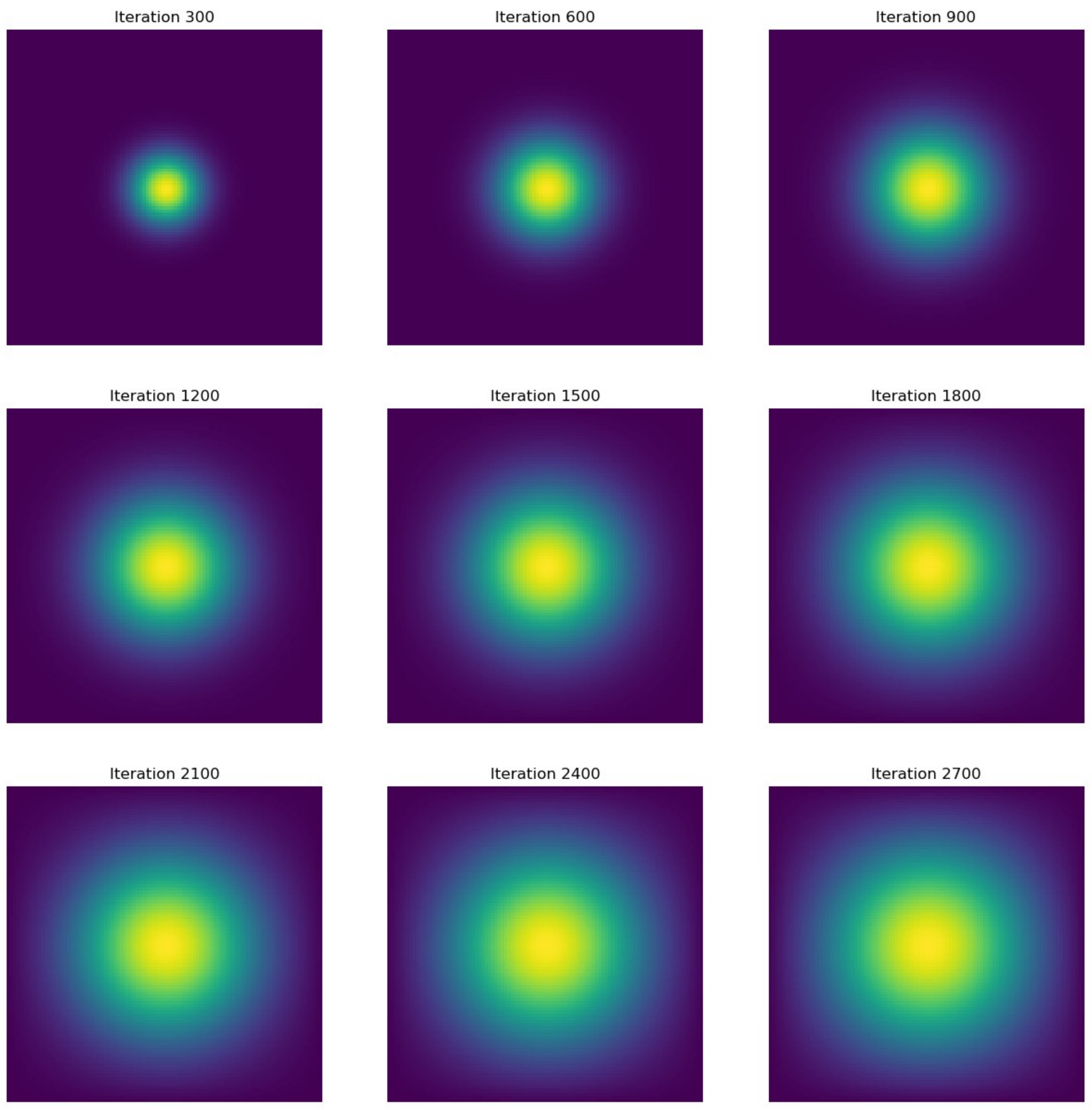
For this next method, we look at sparse matrices which can be produced from the JAX package of Python. In regular matrix multiplication, Most of operations are spent too much on computing zeros. Let’s use the data structure that exploits a lot of zeros in the matrix A: sparse matrix data structures. The JAX package helps us utilize the batched coordinate (BCOO) format to simulate the heat diffusion at a faster rate. First, let's look at the code source of the get_sparse_A() function below.

```
import jax
import jax.numpy as jnp
from jax.experimental.sparse import BCOO
from heat_equation import get_sparse_A

source_code2 = inspect.getsource(get_sparse_A)
print(source_code2)
```

```
def get_sparse_A(N):
    """
    This function constructs the 2D finite difference matrix for heat
diffusion simulation in JAX sparse BCOO format.

    Arguments:
        N (int): Grid size (N x N)

    Returns:
        A_sp_matrix (jax.experimental.sparse.BCOO): The finite
difference matrix in JAX BCOO sparse format
    """

    n_points = N * N  # Total number of grid points

    # Calculate number of elements
    total_indices = n_points + 2*(n_points-N) + 2*(n_points-1)
    rows = np.zeros(total_indices, dtype=np.int32)
    cols = np.zeros(total_indices, dtype=np.int32)
    values = np.zeros(total_indices, dtype=np.float32)

    # Set original index to 0
    index = 0

    # Set main diagonal
    rows[index:index+n_points] = np.arange(n_points)
    cols[index:index+n_points] = np.arange(n_points)
    values[index:index+n_points] = -4.0
    index += n_points

    # Horizontal Connections
    for i in range(N):
        for j in range(N-1):
            current_point = i*N + j
            # Connection on the Right Side
            rows[index] = current_point
            cols[index] = current_point + 1
            values[index] = 1.0
            index += 1
            # Connection on the Left Side
            rows[index] = current_point + 1
            cols[index] = current_point
            values[index] = 1.0
            index += 1

    # Vertical Connection
    for i in range(N-1):
        for j in range(N):
            current_point = i*N + j
            # Connection Going to the Bottom
            rows[index] = current_point
            cols[index] = current_point + N
            values[index] = 1.0
            index += 1
            # Connection Going to the Top
            rows[index] = current_point + N
            cols[index] = current_point
            values[index] = 1.0
            index += 1

    # Convert lists to arrays using jax.numpy (instead of np)
    final_array_positions = jnp.column_stack((jnp.array(rows[:index]),
jnp.array(cols[:index])))
    values = jnp.array(values[:index])

    # Create the sparse matrix
    A_sp_matrix = BCOO((values, final_array_positions),
shape=(n_points, n_points))

    return A_sp_matrix
```

Now, let's see approximately how long the sparse matrix in JAX method takes for same parameters as before: N = 100, epsilon = 0.1, and the number of iterations being 2700.

```
# Main function to run the simulation
def run_simulation_sparse(N, epsilon, num_iterations):
    """
    Runs the heat diffusion simulation for the specified number of
iterations
    and visualizes snapshots in a 3x3 grid.
    """
    A = get_sparse_A(N)  # Get the finite difference matrix A
    u = np.zeros((N, N))  # Initial heat distribution (start with 0
heat everywhere)
    u[N//2, N//2] = 1  # Initial heat source at the center

    # Create a figure with a 3x3 grid of subplots
    fig, axes = plt.subplots(3, 3, figsize=(15, 15))

    # Select snapshots at every 300 iterations
    snapshot_indices = [i for i in range(300, num_iterations + 1,
300)][:9]  # Ensure exactly 9 snapshots

    snapshot_count = 0  # Track which subplot to update
    for t in range(num_iterations + 1):
        u = advance_time_matvecmul(A, u, epsilon)  # Update the grid

        # If current iteration matches a snapshot index, plot it in
the grid
        if t in snapshot_indices:
            row, col = divmod(snapshot_count, 3)  # Get subplot
position
            axes[row, col].imshow(u, cmap='viridis',
interpolation='nearest')
            axes[row, col].set_title(f'Iteration {t}')
            axes[row, col].axis('off')  # Hide axes for clarity
            snapshot_count += 1

# Run the simulation with specified parameters
N = 100  # Grid size (100x100 grid)
epsilon = 0.1  # Stability constant
num_iterations = 2700  # Number of iterations

# Record the start time
start_time2 = time.time()
run_simulation_sparse(N, epsilon, num_iterations)
# Record the end time
end_time2 = time.time()
```



```python
# Calculate the time difference
elapsed_time2 = end_time2 - start_time2
print(f"Time taken for execution: {elapsed_time2} seconds")
```

```
Time taken for execution: 7.658028841018677 seconds
```

As we can see, the sparse matrix multiplication method takes only 7.6580 seconds! This is over 10 times faster (approximately 12.44 times faster) than the regular matrix multiplication method.

Numpy

Now, let's look at the Numpy method to simulate heat diffusion. To simulate heat diffusion using NumPy, you can use a vectorized approach with array operations, such as np.roll(), to shift the values of an input array u by one timestep. The function advance_time_numpy(u, epsilon) should update the heat distribution by applying this shift, along with any necessary boundary conditions, and return the updated array. You can pad the input array with zeros to handle boundaries, ensuring the returned solution still has dimensions of N×N. Below is the source code for the Numpy method function.

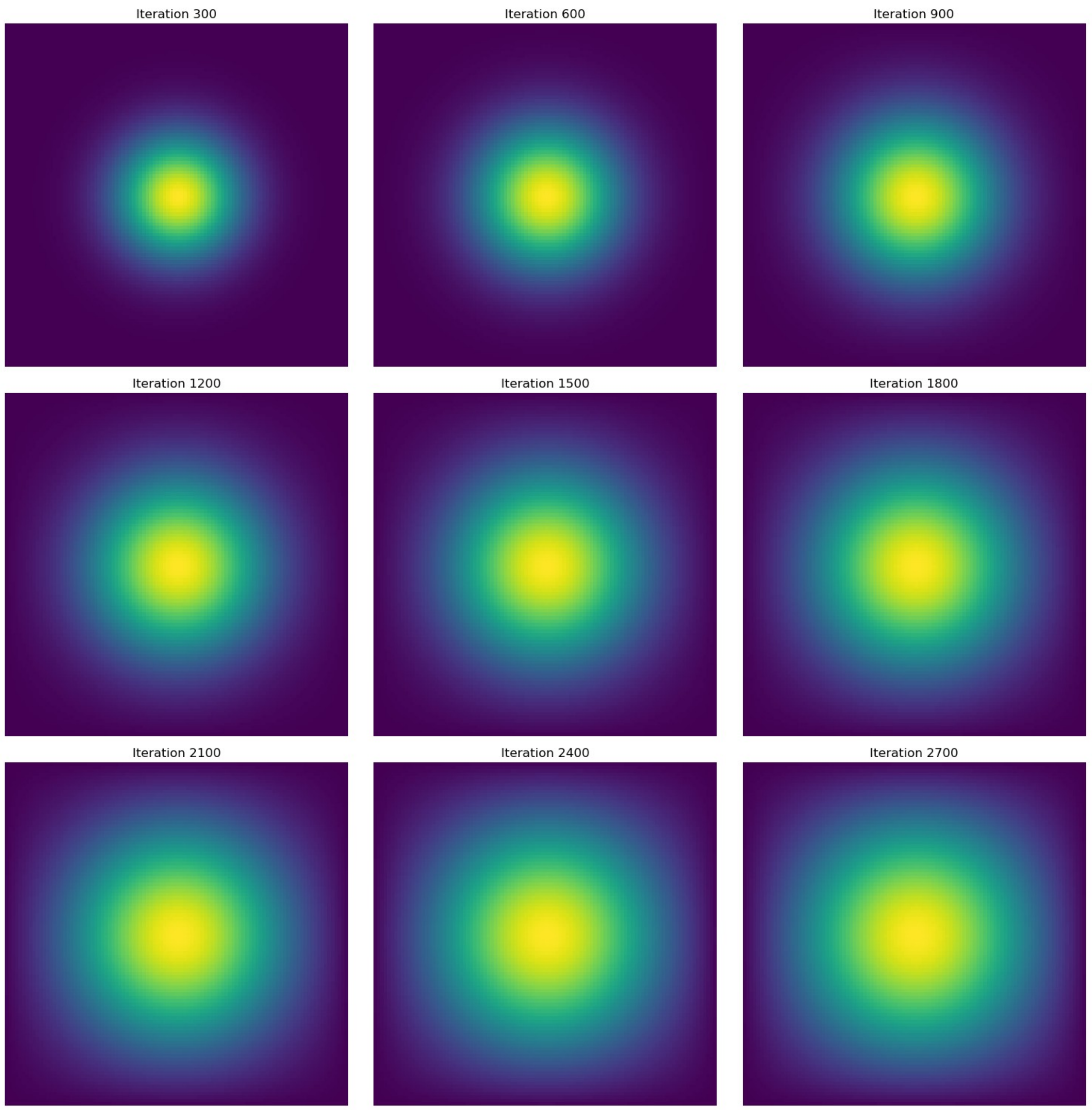
```
from heat_equation import advance_time_numpy

source_code3 = inspect.getsource(advance_time_numpy)
print(source_code3)
```

```
def advance_time_numpy(u, epsilon):
    """
    This function advances the heat equation simulation by one
timestep using NumPy operations.

    Arguments:
        u (np.ndarray): N x N grid state at timestep k
        epsilon (float): stability constant

    Returns:
        u_new (np.ndarray): N x N grid state at timestep k+1
    """
    N = u.shape[0]

    # Create a padded array with zeros around the boundary
    u_padded = np.zeros((N + 2, N + 2))
    u_padded[1:-1, 1:-1] = u

    # Apply the stencil operation using NumPy's array operations
    u_new = u + epsilon * (
        u_padded[0:-2, 1:-1] +  # Top
        u_padded[1:-1, 0:-2] +  # Left
        u_padded[2:, 1:-1] +    # Bottom
        u_padded[1:-1, 2:] -    # Right
        4 * u                   # Center (multiplied by -4)
    )

    return u_new
```

Now, let's see approximately how long the Numpy method takes for same parameters as before: N = 100, epsilon = 0.1, and the number of iterations being 2700.

```
def run_simulation_numpy(N, epsilon, num_iterations):
    """
    Runs the heat diffusion simulation using NumPy for the specified
```

```
number of iterations
    and visualizes snapshots in a 3x3 grid.

    Arguments:
        N (int): Grid size (N x N)
        epsilon (float): Stability constant
        num_iterations (int): Number of iterations to run

    Returns:
        u (np.ndarray): Final state of the grid
        states (list of np.ndarray): List of states at intervals of
300 iterations
    """
    # Initial heat distribution (start with 0 heat everywhere)
    u = np.zeros((N, N))

    # Create a hot spot in the center (Gaussian bump)
    center = N // 2
    radius = N // 10
    x = np.arange(0, N)
    y = np.arange(0, N)
    X, Y = np.meshgrid(x, y)
    u = np.exp(-((X - center)**2 + (Y - center)**2) / (2 * radius**2))

    # Normalize to range [0, 1]
    u = u / np.max(u)

    # Create a figure with a 3x3 grid of subplots
    fig, axes = plt.subplots(3, 3, figsize=(15, 15))

    # Select snapshots at every 300 iterations
    snapshot_indices = [i for i in range(300, num_iterations + 1,
300)][:9]  # Ensure exactly 9 snapshots

    states = [u.copy()]  # Store the initial state
    snapshot_count = 0  # Track which subplot to update

    # Running the simulation
    for t in range(1, num_iterations + 1):
        u = advance_time_numpy(u, epsilon)  # Update the grid using
NumPy method

        # If current iteration matches a snapshot index, store and
plot it
        if t in snapshot_indices:
            states.append(u.copy())
            row, col = divmod(snapshot_count, 3)  # Get subplot
position
            axes[row, col].imshow(u, cmap='viridis',
interpolation='nearest')
            axes[row, col].set_title(f'Iteration {t}')
            axes[row, col].axis('off')  # Hide axes for clarity
            snapshot_count += 1

    plt.tight_layout()
    plt.show()

    return u, states

# Run the simulation with specified parameters
N = 100  # Grid size (100x100 grid)
epsilon = 0.1  # Stability constant
num_iterations = 2700  # Number of iterations

# Record the start time
start_time3 = time.time()
run_simulation_numpy(N, epsilon, num_iterations)
# Record the end time
end_time3 = time.time()
```



```
# Calculate the time difference
elapsed_time3 = end_time3 - start_time3
print(f"Time taken for execution: {elapsed_time3} seconds")

Time taken for execution: 0.7361979484558105 seconds
```

As we can see, the sparse matrix multiplication method takes only about 0.7362 seconds! This is over 10 times faster (approximately 10.40 times faster) than the sparse matrix multiplication method.

JAX Method

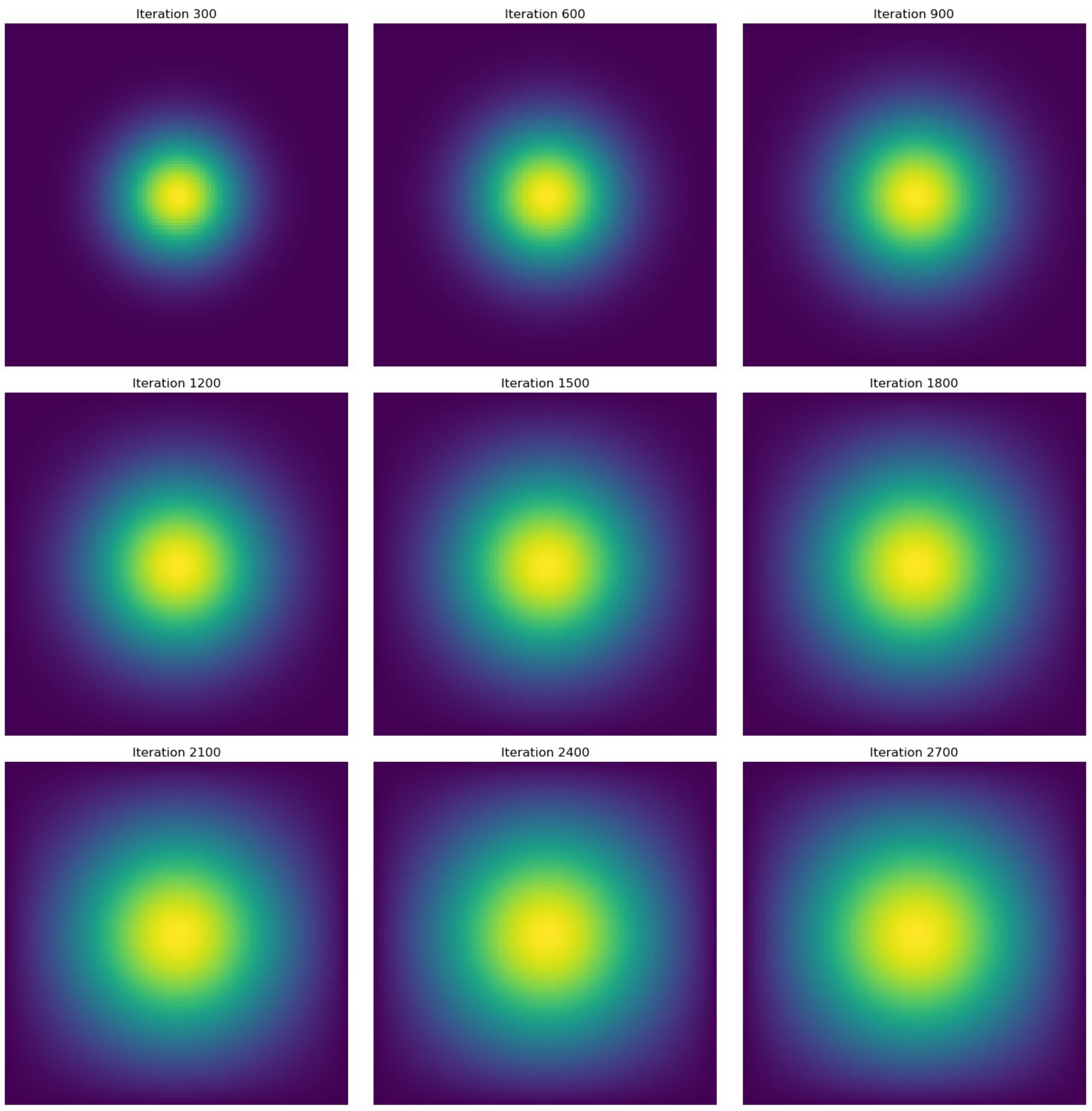
To simulate heat diffusion using JAX, you can define a function advance_time_jax(u, epsilon) based on the previous NumPy implementation. However, JAX requires just-in-time compilation with the jit decorator for improved performance, as it compiles the function for faster execution on subsequent runs. The function should be designed similarly to the NumPy approach but must avoid using index assignments due to JAX's constraints. Below is the source code of function of the JAX method.

```
from heat_equation import advance_time_jax

source_code4 = inspect.getsource(advance_time_jax)
print(source_code4)

@jax.jit
def advance_time_jax(u, epsilon):
    """
    This function advances the heat equation simulation by one
timestep using JAX operations.
    JIT-compiled for faster execution.

    Arguments:
        u (jax.numpy.ndarray): N x N grid state at timestep k
        epsilon (float): stability constant

    Returns:
        u_new (jax.numpy.ndarray): N x N grid state at timestep k+1
    """
    # Create a padded array with zeros around the boundary
    u_padded = jnp.pad(u, 1, mode='constant', constant_values=0)

    # Apply the stencil operation using JAX's array operations
    # This performs the same operation as in the numpy version but
using JAX
    u_new = u + epsilon * (
        u_padded[:-2, 1:-1] +  # Top
        u_padded[1:-1, :-2] +  # Left
        u_padded[2:, 1:-1] +   # Bottom
        u_padded[1:-1, 2:] -   # Right
        4 * u                  # Center (multiplied by -4)
    )

    return u_new
```

Now, let's see approximately how long the JAX method takes for same parameters as before: N = 100, epsilon = 0.1, and the number of iterations being 2700.

```
# Main function to run the simulation
def run_simulation_jax(N, epsilon, num_iterations):
    """
    Runs the heat diffusion simulation using JAX for the specified
number of iterations
    and visualizes snapshots in a 3x3 grid.

    Arguments:
        N (int): Grid size (N x N)
        epsilon (float): Stability constant
        num_iterations (int): Number of iterations to run

    Returns:
        u (np.ndarray): Final state of the grid
        states (list of np.ndarray): List of states at intervals of
300 iterations
    """
    # Initial heat distribution with a Gaussian hot spot in the center
    center = N // 2
    radius = N // 10
    x = np.arange(0, N)
    y = np.arange(0, N)
    X, Y = np.meshgrid(x, y)
    u_init = np.exp(-((X - center)**2 + (Y - center)**2) / (2 *
radius**2))
    u_init = u_init / np.max(u_init)  # Normalize to range [0, 1]

    # Convert to JAX array
    u = jnp.array(u_init)

    # Create a figure with a 3x3 grid of subplots
    fig, axes = plt.subplots(3, 3, figsize=(15, 15))

    # Select snapshots at every 300 iterations
    snapshot_indices = [i for i in range(300, num_iterations + 1,
300)][:9]  # Ensure exactly 9 snapshots

    states = [np.array(u)]  # Store initial state as numpy array for
visualization
    snapshot_count = 0  # Track which subplot to update

    # Running the simulation
    for t in range(1, num_iterations + 1):
        u = advance_time_jax(u, epsilon)  # Update the grid using JAX

        # If current iteration matches a snapshot index, store and
plot it
        if t in snapshot_indices:
            u_np = np.array(u)  # Convert JAX array to NumPy for
visualization
            states.append(u_np)
            row, col = divmod(snapshot_count, 3)  # Get subplot
position
```

```
            axes[row, col].imshow(u_np, cmap='viridis',
interpolation='nearest')
            axes[row, col].set_title(f'Iteration {t}')
            axes[row, col].axis('off')  # Hide axes for clarity
            snapshot_count += 1

    plt.tight_layout()
    plt.show()

    return np.array(u), states

# Run the simulation with specified parameters
N = 100  # Grid size (100x100 grid)
epsilon = 0.1  # Stability constant
num_iterations = 2700  # Number of iterations

# Record the start time
start_time4 = time.time()
run_simulation_jax(N, epsilon, num_iterations)
# Record the end time
end_time4 = time.time()
```



```python
# Calculate the time difference
elapsed_time4 = end_time4 - start_time4
print(f"Time taken for execution: {elapsed_time4} seconds")
```

```
Time taken for execution: 0.4389030933380127 seconds
```

As we can see, the sparse matrix multiplication method takes only about 0.4389 seconds! This is about twice as fast (approximately 1.68 times faster) when compared to the Numpy method!

Conclusion

In conclusion, simulating heat diffusion in Python can be achieved through various methods, with the most common being NumPy and JAX. The NumPy method is straightforward, using vectorized operations such as np.roll() to update the heat distribution over time. However, while this approach is efficient, it may not fully utilize the computational power of modern hardware for large-scale simulations. JAX, on the other hand, leverages just-in-time (JIT) compilation, which allows for faster execution by compiling functions before running them, thus improving performance, especially for large iterations. The JAX method's ability to precompile code and optimize execution makes it significantly faster than NumPy for tasks like simulating heat diffusion, as it minimizes overhead during computation. Consequently, JAX is the optimal choice for simulations that require repeated, large-scale computations, offering a more efficient approach without compromising on accuracy.