Revolutionizing Boutique Hotel Development

DL for LBM Fluid Dynamics Simulation

Team Fluid

Heidi Ongkowijaya
Cornell Master in
Hospitality ’25

Eirwyn Zhang
Carnegie Mellon SCS
GAI & LLM ’25

James Lee
MIT
MBA ‘25

Hamza Naeem
Northeastern
Master in AI ‘26

Khubaib Khan
Northeastern
Master in Robotics ‘26

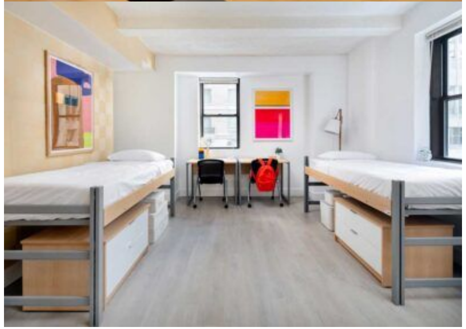
1. Problem faced by Large Hotel Chains
2. Why Deep Learning as the solution?
3. How have it been done before?
4. Model Implementation (Preliminary)

Large Hotel Chains Adaptive Reuse of Historical Buildings

Temple Court Building 1883

The Beekman Hotel, by Hyatt

525 Lexington Avenue

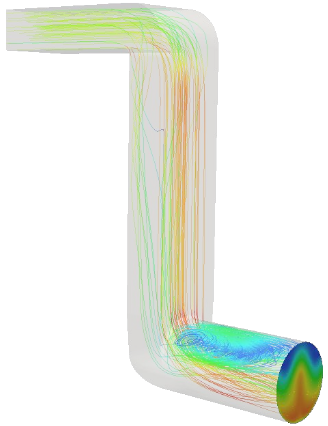


Marriott East Manhattan

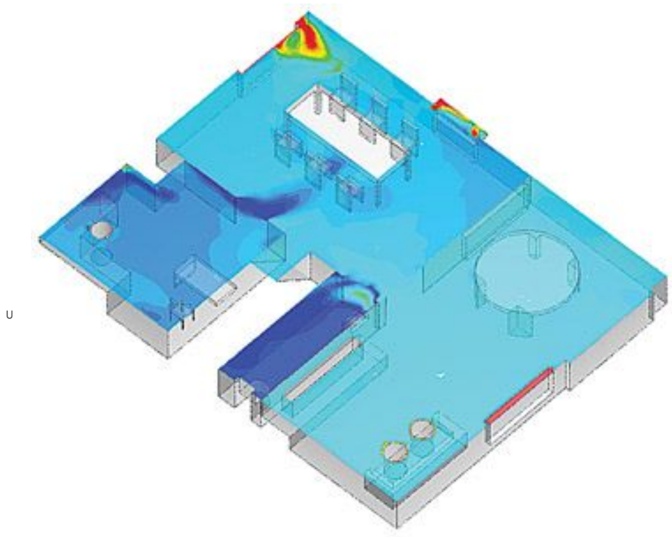
For $10 million boutique hotel project...

Operational & Maintenance (Annual)	$500,000 - $1,000,000	Inefficient systems **increase energy** and **maintenance costs**
Regulatory Compliance & Permitting	$100,000 - $300,000	
Risk Management & Scheduling	$500,000 - $1,000,000	Delays extend construction, **delaying revenue**.
Material	$4,000,000 - $5,000,000	Wrong materials **raise operational costs**.
Prototyping and Testing	$200,000 - $500,000	
Engineering & System Optimization (HVAC, Plumbing, Electrical, etc.)	$600,000 - $1,500,000	Poor optimization **adds 5-15% to engineering costs**.
Architectural & Engineering	$1,000,000 - $2,500,000	Design **iterations add 10-15% to fees**.

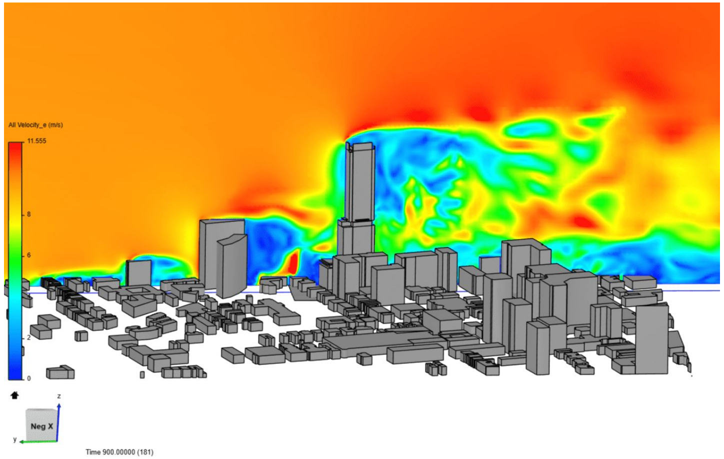
How does this relate to Fluid Dynamics?



Duct Design



Thermal Comfort

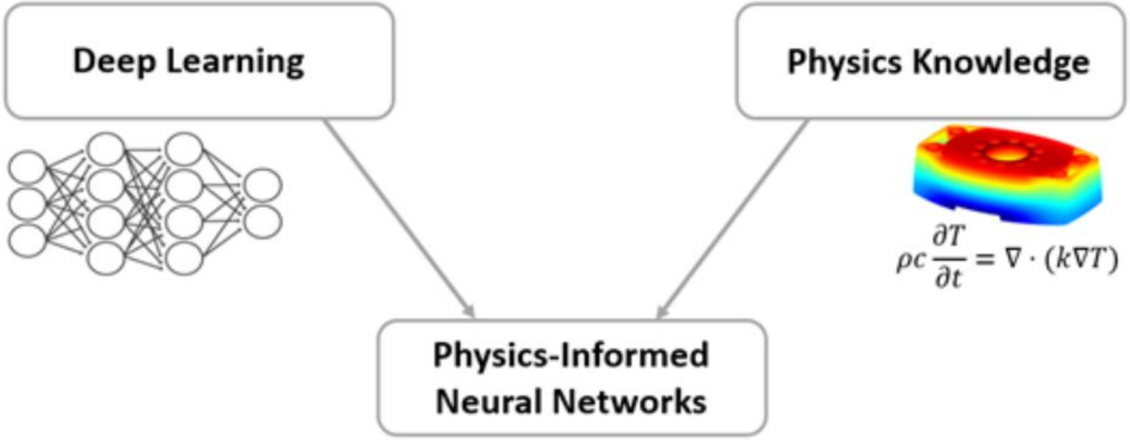


Air flow around
Buildings

Comparison with Traditional Techniques

Feature	CFD (Computational Fluid Dynamics)	LBM (Lattice Boltzmann Method)	Deep Learning (RNN)
Computational Speed	Slow	Faster than CFD	10-100x faster
Memory Requirements	High	Moderate	Low (via representations in low-dim latent space)
Real-Time Predictions	No	No	Yes
Physical Constraints	Built-in	Built-in	Physics-informed loss function

Previous Work: PINNs in Fluid Dynamics

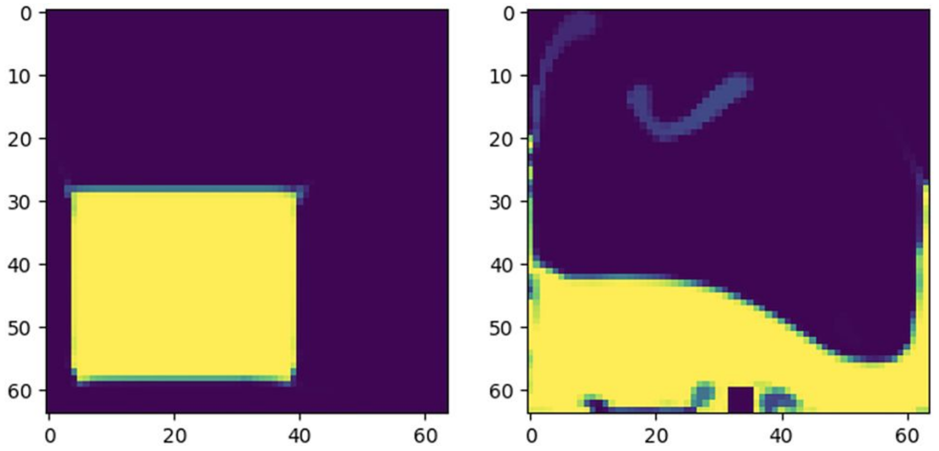


Physics-Informed Neural Networks (PINNs) (Raissi 2019)

Limitations

- Relies on knowledge of equations
- Struggle with 3D Simulations
- Computationally Expensive
- Slow for real-time

Previous Work: GANs & DRL in Fluid Dynamics

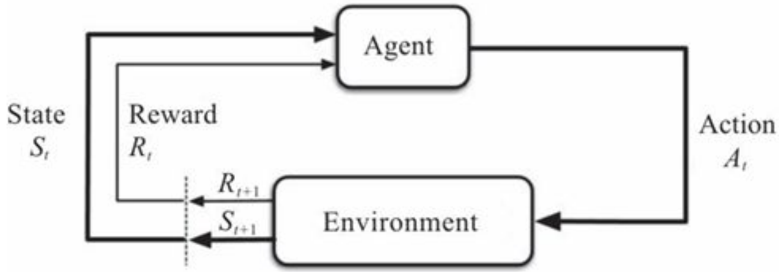


Classic Dam Break problem using Physics Informed GANs
([Li 2022])

Computationally Expensive

Large Dataset

Lack of Temporal Modeling



Deep Reinforcement Learning
([Rabault 2019])

Computationally Expensive

Temporal Modeling

Model Implementation (Preliminary)

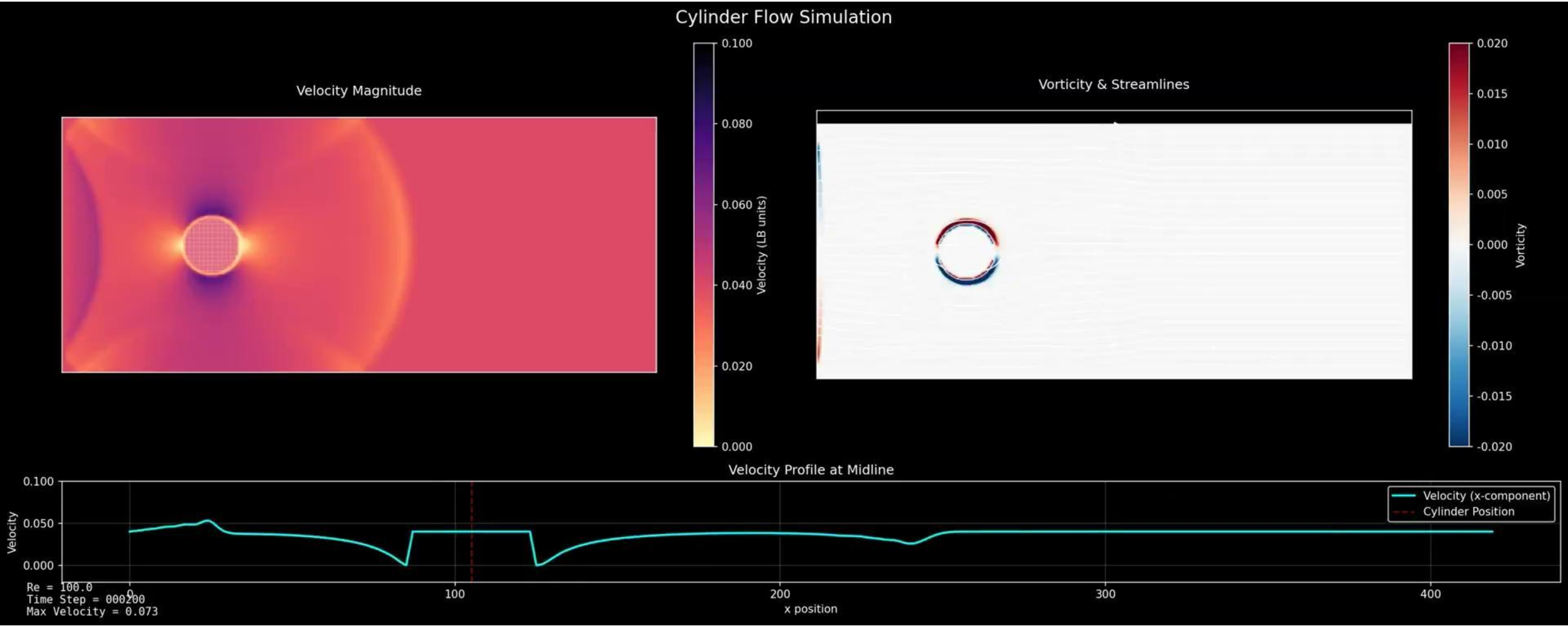
I. Data Generation Phase

- LBM Simulation of cylinder flow
- Reynolds number = 100 (von Kármán vortex shedding)
- Save velocity fields every 10 timesteps
- Data format: 2D velocity components (u, v)

```
class LBM:
    …
    nx, ny = 420, 180     # Domain dimensions
    Re = 100              # Reynolds Number
    uLB = 0.04            # Inlet Velocity
    save_interval = 10    # Timestep save frequency
    …

def save_velocity_field(step, velocity):
    if step % LBMSetup.save_interval == 0:
        np.savez(f'data/field_{step:04d}.npz',
        velocity=velocity)
```

Model Implementation (Preliminary)



Carnegie Mellon University

Model Implementation (Preliminary)

II. Data Processing & Preparation

- Create training sequences from saved fields
- Input: 10 consecutive timesteps (t-9 to t)
- Output: Next timestep prediction (t+1)
- Preserve spatial structure (no flattening)

```
class LBMDataset(Dataset):
    def __getitem__(self, idx):
        # Input: 10 consecutive timesteps
        input_sequence = load_velocity_sequence(idx, seq_len=10)
        # Output: Next timestep
        target = load_next_velocity(idx + 10)

        # Preserve spatial structure
        inputs = torch.FloatTensor(input_sequence)   # (10, 2, H, W)
        target = torch.FloatTensor(target)           # (2, H, W)

        return inputs, target
```

Model Implementation (Preliminary)

III. Neural Network Architecture -- **ConvLSTM**: Hybrid architecture incorporating convolution operations for spatial features (**CNN**-like) and memory cells for temporal sequences (**RNN**-like)

- Input Layer: Velocity field sequences
- Three ConvLSTM Layers:
    - Each layer processes both spatial and temporal features
    - Hidden channels: 64
    - Kernel size: 3x3
- Output Layer: 1x1 Convolution
    - Maps features back to velocity field
    - Predicts next timestep

Model Implementation (Preliminary)

```python
class ConvLSTMCell(nn.Module):
    def __init__(self, input_channels, hidden_channels=64):
        # Spatial feature extraction
        self.conv = nn.Conv2d(
            in_channels=input_channels + hidden_channels,
            out_channels=4 * hidden_channels,
            kernel_size=3,  # 3x3 kernel
            padding=1
        )

    def forward(self, x, h_prev, c_prev):
        # Combine spatial and temporal features
        combined = torch.cat([x, h_prev], dim=1)

        # Process through convolutional LSTM cell
        conv_output = self.conv(combined)
        # Gate computation & state update
        gates = torch.split(conv_output, self.hidden_channels, dim=1)
        # LSTM update rules applied...
```

Model Implementation (Preliminary)

IV. Training Configuration

- Batch size: 8
- Learning rate: 0.001
- Optimizer: Adam
- Loss function: MSE
- Learning rate scheduling
- Gradient clipping

```
def train_model():
    # Hyperparameters
    config = {
        'batch_size': 8,
        'learning_rate': 0.001,
        'optimizer': 'Adam',
        'loss': 'MSE',
        'epochs': 50
    }

    # Gradient clipping & learning rate scheduling
    optimizer = torch.optim.Adam(model.parameters(),
                lr=config['learning_rate'])
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='min', factor=0.5, patience=5
    )
```

Model Implementation (Preliminary)

```python
for epoch in range(num_epochs):
    model.train()
    total_loss = 0

    for batch_inputs, batch_targets in tqdm(dataloader, desc=f'Epoch {epoch+1}/{num_epochs}'):
        batch_inputs = batch_inputs.to(device)  # (batch, seq, channels, height, width)
        batch_targets = batch_targets.to(device)  # (batch, channels, height, width)

        optimizer.zero_grad()
        outputs, _ = model(batch_inputs)
        loss = criterion(outputs, batch_targets)

        loss.backward()
        # Gradient clipping to prevent exploding gradients
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        total_loss += loss.item()

    avg_loss = total_loss / len(dataloader)
    print(f'Epoch {epoch+1}, Average Loss: {avg_loss:.6f}')

    # Learning rate scheduling
    scheduler.step(avg_loss)
```

```
Epoch 1/50:  12%|█          | 310/2499 [5:49:39<37:02:38, 60.92s/it]
```

Model Implementation (Preliminary)

V. Expected Output

- Predicted velocity field at t+1
- Same spatial dimensions as input
- Two channels (u, v components)

```
def predict_next_timestep(model, input_sequence):
    """
    Predicts next velocity field
    Maintains input spatial dimensions
    """
    predicted_field = model(input_sequence)
    return predicted_field  # Shape: (2, H, W)
```

```
Velocity Field Prediction:
V(t+1) = ConvLSTM(V(t-9), ..., V(t))

Where:
- V(t): Velocity field at timestep t
- ConvLSTM: Learns spatiotemporal mapping
```