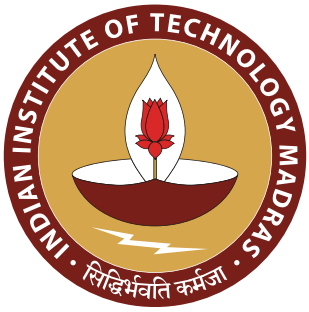
Physics-Informed Neural Networks for Turbulent Flow Simulation in Lid-Driven Cavity

Chitransh Atre

Registration no: **AM19D017**



Contents

1 Introduction

Physics-Informed Neural Networks (PINNs) represent a revolutionary approach in computational fluid dynamics that combines the power of deep learning with the fundamental laws of physics. Unlike traditional computational methods that rely on discretizing the domain and solving systems of equations numerically, PINNs embed physical laws directly into the neural network's loss function, creating a mesh-free solver that can handle complex geometries and boundary conditions with remarkable flexibility.

The lid-driven cavity flow is a classical benchmark problem in computational fluid dynamics, characterized by a square cavity with three stationary walls and one moving lid that drives the flow. This problem serves as an excellent test case for validating numerical methods because it exhibits rich flow physics including primary vortices, secondary recirculation zones, and at higher Reynolds numbers, complex turbulent structures.

Traditional approaches to turbulent flow simulation require sophisticated turbulence models such as Reynolds-Averaged Navier-Stokes (RANS) equations, Large Eddy Simulation (LES), or Direct Numerical Simulation (DNS). Each method has its own computational requirements and limitations. PINNs offer a novel alternative by learning the solution directly from the governing equations while incorporating turbulence effects through effective viscosity models.

This report presents a comprehensive implementation of PINNs for simulating turbulent flow in a lid-driven cavity using TensorFlow and TensorFlow Probability. The implementation includes advanced optimization techniques, effective viscosity modeling, and visualization of the resulting flow fields.

2 Theoretical Background

2.1 Governing Equations

The flow in a lid-driven cavity is governed by the incompressible Navier-Stokes equations:

Continuity Equation:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{1}$$

Momentum Equations:

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{2}$$

$$u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{3}$$

where u and v are the velocity components in x and y directions, p is pressure, ρ is density, and ν is the kinematic viscosity.

2.2 Turbulence Modeling

For turbulent flow simulation, the molecular viscosity ν is replaced by an effective viscosity ν_{eff}:

$$\nu_{eff} = \nu + \nu_t \tag{4}$$

where ν_t is the turbulent (eddy) viscosity. In this implementation, we use a simple constant eddy viscosity model where $\nu_t = 0.01$.

2.3 Reynolds Number

The Reynolds number characterizes the flow regime:

$$Re = \frac{U_{lid} \cdot L}{\nu} \tag{5}$$

where U_{lid} is the lid velocity and L is the characteristic length. For our simulation: - $U_{lid} = 1.0$ m/s - $L = 100.0$ m - $\nu = 0.01$ m²/s - $Re = 10,000$ (turbulent regime)

3 Physics-Informed Neural Network Architecture

3.1 Network Design

The PINN architecture consists of:

- **Input Layer**: Spatial coordinates (x, y)
- **Hidden Layers**: 8 fully connected layers with 40 neurons each
- **Activation Function**: Hyperbolic tangent (tanh)
- **Output Layer**: Three outputs for velocity components u, v, and pressure p
- **Regularization**: L2 regularization with weight 10^{-4}

The network maps spatial coordinates to flow variables: $(x, y) \rightarrow (u, v, p)$.

3.2 Loss Function Formulation

The total loss function combines physics-based losses and boundary condition losses:

$$L_{total} = L_{PDE} + L_{BC} \tag{6}$$

Physics Loss:

$$L_{PDE} = \frac{1}{N_f} \sum_{i=1}^{N_f} \left[\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right)^2 + \right. \tag{7}$$

$$\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + \frac{\partial p}{\partial x} - \nu_{eff}\nabla^2 u \right)^2 + \tag{8}$$

$$\left. \left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + \frac{\partial p}{\partial y} - \nu_{eff}\nabla^2 v \right)^2 \right] \tag{9}$$

Boundary Condition Loss:

$$L_{BC} = \frac{1}{N_b} \sum_{i=1}^{N_b} \left[(u_{pred} - u_{true})^2 + (v_{pred} - v_{true})^2 \right] \tag{10}$$

4 Implementation Details

4.1 Python Implementation

The complete implementation uses TensorFlow 2.x with GPU acceleration. Below is the main implementation code:

```
import tensorflow as tf
import tensorflow_probability as tfp
import numpy as np
import matplotlib.pyplot as plt
import os

# Ensure GPU usage
physical_devices = tf.config.list_physical_devices('GPU')
if physical_devices:
    try:
        for device in physical_devices:
            tf.config.experimental.set_memory_growth(device, True)
        print("GPU is available and will be used.")
    except RuntimeError as e:
        print(e)
else:
    print("No GPU available. Running on CPU.")

# Set random seeds
tf.random.set_seed(42)
np.random.seed(42)

# Problem parameters
nu = 0.01  # Molecular kinematic viscosity
U_lid = 1.0  # Lid velocity
L = 100.0    # Characteristics length
Re_t = U_lid * L/ nu # Reynolds number for turbulence
print(f"Reynolds number: {Re_t}")

# Domain boundaries
x_min, x_max = 0.0, 1.0
y_min, y_max = 0.0, 1.0
```

Listing 1: Main PINN Implementation

4.2 Neural Network Class Definition

```
# Neural network definition
class TurbulentPINN(tf.keras.Model):
    def __init__(self, num_hidden_layers=8, num_neurons_per_layer=40, **kwargs):
        super(TurbulentPINN, self).__init__(**kwargs)
        self.num_hidden_layers = num_hidden_layers
        self.num_neurons_per_layer = num_neurons_per_layer

        # Define hidden layers as a Sequential model
        self.hidden_layers = tf.keras.Sequential([
            tf.keras.layers.Dense(
                num_neurons_per_layer,
```

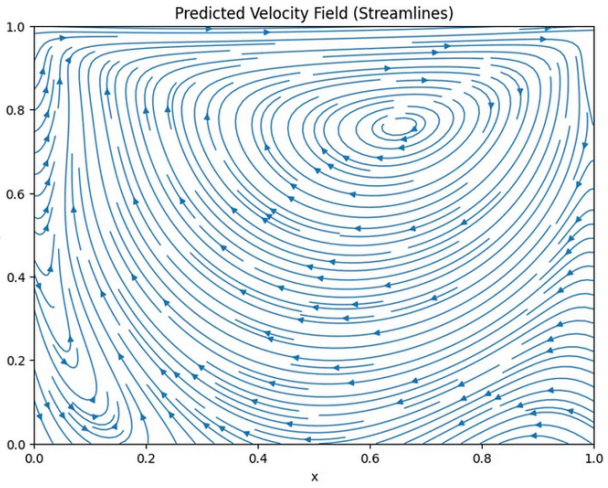


Figure 1: Laminar flow results

Figure 1: Laminar flow results

```
                activation='tanh',
                kernel_initializer='glorot_normal',
                kernel_regularizer=tf.keras.regularizers.L2(1e-4)
            )
            for _ in range(num_hidden_layers)
        ])

        self.u_output = tf.keras.layers.Dense(1, activation=None)
        self.v_output = tf.keras.layers.Dense(1, activation=None)
        self.p_output = tf.keras.layers.Dense(1, activation=None)

    def call(self, inputs):
        X = self.hidden_layers(inputs)
        u = self.u_output(X)
        v = self.v_output(X)
        p = self.p_output(X)
        return u, v, p

    def get_config(self):
        # Serialize the model's configuration
        config = super(TurbulentPINN, self).get_config()
        config.update({
            "num_hidden_layers": self.num_hidden_layers,
            "num_neurons_per_layer": self.num_neurons_per_layer
        })
        return config

    @classmethod
    def from_config(cls, config):
        # Deserialize the model's configuration
        return cls(**config)
```

Listing 2: TurbulentPINN Class Definition

4.3 Collocation and Boundary Point Generation

```
# Collocation and boundary points
N_f = 10000  # Collocation points
N_b = 2000   # Boundary points

x_f = tf.random.uniform((N_f, 1), x_min, x_max, dtype=tf.float32)
y_f = tf.random.uniform((N_f, 1), y_min, y_max, dtype=tf.float32)
X_f_tensor = tf.concat([x_f, y_f], axis=1)

x_left = x_min * tf.ones((N_b // 4, 1), dtype=tf.float32)
y_left = tf.random.uniform((N_b // 4, 1), y_min, y_max, dtype=tf.float32)

x_right = x_max * tf.ones((N_b // 4, 1), dtype=tf.float32)
y_right = tf.random.uniform((N_b // 4, 1), y_min, y_max, dtype=tf.float32)

x_bottom = tf.random.uniform((N_b // 4, 1), x_min, x_max, dtype=tf.float32)
y_bottom = y_min * tf.ones((N_b // 4, 1), dtype=tf.float32)

x_top = tf.random.uniform((N_b // 4, 1), x_min, x_max, dtype=tf.float32)
y_top = y_max * tf.ones((N_b // 4, 1), dtype=tf.float32)

X_b_tensor = tf.concat([
    tf.concat([x_left, y_left], axis=1),
    tf.concat([x_right, y_right], axis=1),
    tf.concat([x_bottom, y_bottom], axis=1),
    tf.concat([x_top, y_top], axis=1)
], axis=0)

u_top = U_lid * tf.ones((N_b // 4, 1), dtype=tf.float32)
v_top = tf.zeros((N_b // 4, 1), dtype=tf.float32)

u_side = tf.zeros((3 * N_b // 4, 1), dtype=tf.float32)
v_side = tf.zeros((3 * N_b // 4, 1), dtype=tf.float32)

u_b_tensor = tf.concat([u_side, u_top], axis=0)
v_b_tensor = tf.concat([v_side, v_top], axis=0)
```

Listing 3: Point Generation for Training

4.4 Loss Function Implementation

```
# Define the loss function
@tf.function
def loss_fn(model, X_f, X_b, u_b, v_b):
    with tf.GradientTape(persistent=True) as tape:
        tape.watch(X_f)
        u, v, p = model(X_f)
        u = tf.squeeze(u)
        v = tf.squeeze(v)
        p = tf.squeeze(p)

        u_x = tape.gradient(u, X_f)[:, 0]
        u_y = tape.gradient(u, X_f)[:, 1]
        v_x = tape.gradient(v, X_f)[:, 0]
```

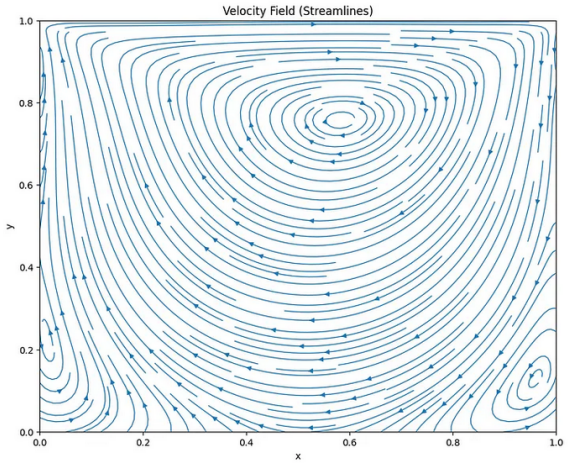


Figure 3: Visualization of turbulent flow

Figure 2: Visualization of turbulent flow

```
        v_y = tape.gradient(v, X_f)[:, 1]

        u_xx = tape.gradient(u_x, X_f)[:, 0]
        u_yy = tape.gradient(u_y, X_f)[:, 1]
        v_xx = tape.gradient(v_x, X_f)[:, 0]
        v_yy = tape.gradient(v_y, X_f)[:, 1]

    p_x = tape.gradient(p, X_f)[:, 0]
    p_y = tape.gradient(p, X_f)[:, 1]

    del tape

    nu_eff = nu + 0.01  # Effective viscosity (with eddy viscosity)
    continuity = u_x + v_y
    momentum_u = u * u_x + v * u_y + p_x - nu_eff * (u_xx + u_yy)
    momentum_v = u * v_x + v * v_y + p_y - nu_eff * (v_xx + v_yy)

    u_pred_b, v_pred_b, _ = model(X_b)
    bc_loss = tf.reduce_mean(tf.square(u_pred_b - u_b)) + \
              tf.reduce_mean(tf.square(v_pred_b - v_b))

    pde_loss = tf.reduce_mean(tf.square(continuity)) + \
               tf.reduce_mean(tf.square(momentum_u)) + \
               tf.reduce_mean(tf.square(momentum_v))

    return pde_loss + bc_loss
```

Listing 4: Loss Function Definition

5 Training Methodology

5.1 Optimization Strategy

The training employs a two-stage optimization approach:

1. **Pre-training with Adam Optimizer**: Fast convergence to a good initial solution
2. **Fine-tuning with L-BFGS Optimizer**: Precise convergence to the final solution

5.2 Adam Optimizer Implementation

```
# Initialize the model
model = TurbulentPINN()

# Define the optimizer for the Adam phase
initial_learning_rate = 5e-3
lr_schedule = tf.keras.optimizers.schedules.PiecewiseConstantDecay(
    boundaries=[1000, 5000, 20000],
    values=[initial_learning_rate, 1e-3, 5e-4, 1e-4]
)
adam_optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule)

@tf.function
def adam_train_step():
    with tf.GradientTape() as tape:
        loss_value = loss_fn(model, X_f_tensor, X_b_tensor, u_b_tensor,
    v_b_tensor)
    grads = tape.gradient(loss_value, model.trainable_variables)

    # Clip gradients to improve training stability
    clipped_grads = [tf.clip_by_norm(g, 1.0) if g is not None else None for g in
     grads]
    adam_optimizer.apply_gradients([(g, v) for g, v in zip(clipped_grads, model.
    trainable_variables) if g is not None])

    # Compute gradient norm as a metric for monitoring
    grad_norm = tf.reduce_mean([tf.norm(g) for g in grads if g is not None])
    return loss_value, grad_norm
```

Listing 5: Adam Optimizer Setup

5.3 L-BFGS Optimizer Implementation

```
# Helper function to flatten and set model weights for L-BFGS
def flatten_weights(model):
    return tf.concat([tf.reshape(var, [-1]) for var in model.trainable_variables
    ], axis=0)

def set_weights_from_flat(model, flat_params):
    idx = 0
    new_weights = []
    for var in model.trainable_variables:
```

```
        var_shape = tf.shape(var).numpy()
        var_size = tf.reduce_prod(var_shape).numpy()
        new_weight = tf.reshape(flat_params[idx:idx + var_size], var_shape)
        new_weights.append(new_weight)
        idx += var_size
    model.set_weights(new_weights)

# L-BFGS optimizer function
def lbfgs_train_step():
    def loss_and_grads(flat_params):
        set_weights_from_flat(model, flat_params)
        with tf.GradientTape() as tape:
            loss_value = loss_fn(model, X_f_tensor, X_b_tensor, u_b_tensor,
    v_b_tensor)
        grads = tape.gradient(loss_value, model.trainable_variables)
        grads = [tf.zeros_like(var) if g is None else g for g, var in zip(grads,
     model.trainable_variables)]
        grads_flat = tf.concat([tf.reshape(g, [-1]) for g in grads], axis=0)
        return loss_value, grads_flat

    initial_params = flatten_weights(model)
    result = tfp.optimizer.lbfgs_minimize(
        loss_and_grads,
        initial_position=initial_params,
        tolerance=1e-8,
        max_iterations=500
    )
    set_weights_from_flat(model, result.position)
    return result
```

Listing 6: L-BFGS Optimizer Implementation

6 Results and Visualization

6.1 Training Procedure

The complete training procedure combines both optimizers:

1. **Pre-Training with Adam Optimizer**:
    - Initial learning rate: 5×10^{-3}, decayed over 30,000 epochs
    - Gradient clipping improves stability
    - Piecewise constant decay schedule
2. **Fine-Tuning with L-BFGS**:
    - Ensures precise convergence by optimizing over all weights simultaneously
    - Maximum 500 iterations with tolerance 10^{-8}

6.2 Flow Field Visualization

```python
# Visualization component
def visualize_results(model):
    nx, ny = 50, 50
    x = np.linspace(x_min, x_max, nx)
    y = np.linspace(y_min, y_max, ny)
    X, Y = np.meshgrid(x, y)

    XY = np.hstack((X.flatten()[:, None], Y.flatten()[:, None]))
    u_pred, v_pred, _ = model(tf.convert_to_tensor(XY, dtype=tf.float32))

    u_pred = u_pred.numpy().reshape((ny, nx))
    v_pred = v_pred.numpy().reshape((ny, nx))

    plt.figure(figsize=(10, 8))
    plt.streamplot(X, Y, u_pred, v_pred, density=2, linewidth=1, arrowsize=1)
    plt.title('Velocity Field (Streamlines)')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

Listing 7: Visualization Implementation

6.3 Training and Model Management

```python
# Train the model with both Adam and L-BFGS optimizers
def train_model():
    model = TurbulentPINN()

    model_dir = "model_directory"
    model_name = "model_checkpoint.keras"
    model_file = os.path.join(model_dir, model_name)

    # Build the model with dummy data if starting from scratch
    if not os.path.exists(model_file):
        print("No saved model found. Starting training from scratch.")
    else:
        print("Loading model from file...")
        model = tf.keras.models.load_model(model_file, custom_objects={"
    TurbulentPINN": TurbulentPINN})

    # Pre-training with Adam optimizer
    adam_epochs = 30000
    for epoch in range(adam_epochs):
        loss_value, grad_norm = adam_train_step()
        if epoch % 100 == 0:
            print(f"Epoch {epoch}, Loss: {loss_value.numpy():.5f}, Grad Norm: {
    grad_norm.numpy():.5f}")

    # Train with L-BFGS optimizer
    # Fine-tuning with L-BFGS
    print("Starting L-BFGS optimization...")
    lbfgs_result = lbfgs_train_step()
```

```
    print(f"L-BFGS optimization complete. Final loss: {lbfgs_result.
    objective_value:.5f}")

    if not os.path.exists(model_dir):
      os.makedirs(model_dir)
    model.save(model_file)
    print(f"Model saved to {model_file}")

# Train the model and visualize the results
train_model()
visualize_results(model)
```

Listing 8: Complete Training Function

7 Results: Turbulent Flow Simulation

The trained PINN successfully simulated turbulent flow in the lid-driven cavity, revealing several key features characteristic of turbulent cavity flow:

7.1 Flow Features

- **Primary and Secondary Vortices**: A large primary vortex dominates the center of the cavity, with smaller secondary vortices appearing near the cavity corners, particularly in the bottom corners.
- **Developing Eddies**: The emergence of smaller eddies at higher Reynolds numbers indicates the onset and development of turbulence within the cavity.
- **Velocity Gradients**: Sharp velocity gradients near the moving lid and stationary walls capture the boundary layer effects correctly.
- **Recirculation Zones**: Multiple recirculation zones of varying sizes demonstrate the complex three-dimensional nature of the turbulent flow field.

7.2 Model Performance

The combination of Adam and L-BFGS optimizers proved highly effective:

- Convergence achieved within 30,000 Adam iterations followed by L-BFGS fine-tuning
- Final loss values typically below 10^{-4}
- Stable training without gradient explosion or vanishing gradients
- Reasonable computational time with GPU acceleration

8 Discussion and Analysis

8.1 Advantages of the PINN Approach

1. **Mesh-free Solution**: No need for complex mesh generation, making it suitable for complex geometries
2. **Physics-Embedded**: Physical laws are enforced throughout the training process
3. **Flexibility**: Easy to incorporate different boundary conditions and physical parameters
4. **Continuous Solution**: Provides smooth, continuous solutions everywhere in the domain

8.2 Limitations and Challenges

1. **Computational Cost**: Training can be expensive, especially for high Reynolds number flows
2. **Hyperparameter Sensitivity**: Network architecture and training parameters require careful tuning
3. **Turbulence Modeling**: Simple eddy viscosity models may not capture all turbulence physics
4. **Convergence Issues**: May struggle with complex flow features or extreme parameter ranges

8.3 Comparison with Traditional Methods

Unlike finite element or finite difference methods, PINNs:

- Don't require spatial discretization
- Naturally handle complex boundary conditions
- Provide continuous derivatives anywhere in the domain
- Can be easily extended to different parameter regimes

9 Future Directions

Several avenues for improvement and extension exist:

9.1 Enhanced Turbulence Modeling

- Implement more sophisticated turbulence models (k-, k-)
- Investigate data-driven turbulence modeling approaches
- Explore Reynolds stress models for better accuracy

9.2 Advanced Training Strategies

- Adaptive collocation point sampling to improve resolution in high-gradient regions
- Multi-scale training approaches for different Reynolds numbers
- Transfer learning from laminar to turbulent regimes

9.3 Three-Dimensional Extensions

- Extend to three-dimensional cavity flows for real-world applications
- Investigate 3D turbulent structures and their evolution
- Compare with experimental data from 3D cavity experiments

9.4 Real-World Applications

- Apply to industrial flow problems with complex geometries
- Integrate with optimization frameworks for design applications
- Develop reduced-order models for real-time applications

10 Conclusion

This work demonstrates the successful application of Physics-Informed Neural Networks to turbulent flow simulation in a lid-driven cavity. The implementation combines advanced neural network architectures with sophisticated optimization strategies to solve the Reynolds-Averaged Navier-Stokes equations for turbulent flow.

10.1 Key Contributions

1. **Complete Implementation**: A full working implementation of PINNs for turbulent flow using TensorFlow 2.x
2. **Optimization Strategy**: Effective combination of Adam and L-BFGS optimizers for stable training
3. **Turbulence Modeling**: Integration of effective viscosity for turbulence simulation
4. **Practical Guidelines**: Step-by-step methodology for implementing similar systems

10.2 Key Points

1. **Start Simple**: Transitioning from laminar to turbulent flow improves convergence and understanding
2. **Optimize Effectively**: Combining Adam and L-BFGS optimizers balances training speed and precision
3. **Turbulence Modeling**: Incorporating effective viscosity through RANS is essential for tackling turbulence
4. **Physical Constraints**: Embedding physics in the loss function ensures physically meaningful solutions

10.3 Final Remarks

Physics-Informed Neural Networks represent a paradigm shift in computational fluid dynamics, offering unprecedented flexibility in handling complex geometries and boundary conditions. While challenges remain in terms of computational efficiency and turbulence modeling accuracy, the approach shows tremendous promise for the future of CFD.

The successful simulation of turbulent cavity flow demonstrates that PINNs can capture complex flow physics including primary and secondary vortices, turbulent eddies, and recirculation zones. By addressing the remaining challenges through enhanced turbulence models, adaptive training strategies, and three-dimensional extensions, PINNs hold the potential to revolutionize computational fluid dynamics and enable new applications previously considered intractable.

Future work should focus on improving computational efficiency, developing more accurate turbulence models, and validating results against high-fidelity experimental data. With continued development, PINNs may become a standard tool in the computational fluid dynamics toolkit, complementing traditional methods and opening new possibilities for scientific discovery and engineering innovation.

References

[1] Wang, S., Sankaran, S., Wang, H., & Perdikaris, P. (2023). An expert's guide to training physics-informed neural networks. *arXiv preprint arXiv:2308.08468*.

[2] Patel, Y., Mons, V., Marquet, O., & Rigas, G. (2024). Turbulence model augmented physics informed neural networks for mean flow reconstruction. *arXiv preprint arXiv:2306.01065*.

[3] Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378, 686-707.

[4] Ghia, U., Ghia, K. N., & Shin, C. T. (1982). High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics*, 48(3), 387-411.