Chapter 10

Validation Examples

In this chapter some applications developed using Kratos are presented to test the desired flexibility and extensibility of the framework.

10.1 Incompressible Fluid Solver

In this example an incompressible fluid application implemented in Kratos is described. It shows the ability of Kratos to handle a standard finite element formulation efficiently while achieving performance comparable to single purpose codes.

10.1.1 Methodology Used

An Arbitrary Lagrangian Eulerian (ALE) formulation is used to solve the fluid problem [32]. The solver is based on fractional step method [26] using equal order pressure-velocity elements stabilized by Orthogonal subscales (OSS) [27]. This application uses an element based approach optimized for simplex elements.

10.1.2 Implementation in Kratos

The fractional step method chosen consists of four solution steps, of which the first one involves a nonlinear loop for solving the nonlinearity in the convection term, while the rest are linear and the third one involves the explicit calculation of projection terms.

This method can be implemented effectively by creating a new `SolvingStrategy` combining existing ones for different steps. The strategy is hard coded in C++, however the implementation is such that all the different steps can be solved separately. In this way a flexible Python interface allowing direct interaction with the solver can be defined. With this interface, different parts of the algorithm can be changed reusing existing strategies with minimal performance loss. This flexibility provides major advantages in defining new fluid structure interaction coupling strategies based in existing ones.

For the present application a combined `Strategy` is created to handle the solution process. The following code shows the `Solve` method of this `Strategy` which calls other methods for implementing different steps:

```
double Solve()
{
  // assign the correct fractional step coefficients
  InitializeFractionalStep(this->m_step, this->mtime_order);
  double Dp_norm;

  //predicting the velocity
  PredictVelocity(this->m_step,this->mprediction_order);

  //initialize projections at the first steps
  InitializeProjections(this->m_step);

  //Assign Velocity To Fract Step Velocity and Node Area to Zero
  AssignInitialStepValues();

  if(this->m_step <= this->mtime_order)
      Dp_norm = IterativeSolve();
  else
  {
      if(this->mpredictor_corrector == false) //standard fractional step
          Dp_norm = FracStepSolution();
      else  //iterative solution
          Dp_norm = IterativeSolve();
  }

  if(this->mReformDofAtEachIteration == true )
      this->Clear();

  this->m_step += 1;
  this->mOldDt =  BaseType::GetModelPart().GetProcessInfo()[DELTA_TIME];

  return Dp_norm;
}


double FracStepSolution()
{
  //setting the fractional velocity to the value of the velocity
  AssignInitialStepValues();

  //solve first step for fractional step velocities
  this->SolveStep1(this->mvelocity_toll, this->mMaxVelIterations);

  //solve for pressures (and recalculate the nodal area)
  double Dp_norm = this->SolveStep2();

  this->ActOnLonelyNodes();

  //calculate projection terms
  this->SolveStep3();

  //correct velocities
  this->SolveStep4();
```

```
   return Dp_norm;
 }
```

This strategy can be exported to Python without loss of performance, by providing access to the methods implementing each step in above strategy class. We can write in Python an equivalent solution step as follows:

```
def SolutionStep1(self):
    normDx = Array3(); normDx[0] = 0.00; normDx[1] = 0.00; normDx[2] = 0.00;
    is_converged = False
    iteration = 0

    while(  is_converged == False and iteration < self.max_vel_its  ):
    (self.solver).FractionalVelocityIteration(normDx);
        is_converged = (self.solver).ConvergenceCheck(normDx,self.vel_toll);
    print iteration,normDx
        iteration = iteration + 1


def Solve(self):
    if(self.ReformDofAtEachIteration == True):
        (self.neighbour_search).Execute()

    (self.solver).InitializeFractionalStep(self.step, self.time_order);
    (self.solver).InitializeProjections(self.step);
    (self.solver).AssignInitialStepValues();


    self.SolutionStep1()

    (self.solver).SolveStep2();
    (self.solver).ActOnLonelyNodes();
    (self.solver).SolveStep3();
    (self.solver).SolveStep4();

    self.step = self.step + 1

    if( self.ReformDofAtEachIteration == True):
        (self.solver).Clear()
```

As can be seen the Python code is self explanatory and simple. Providing this interface also has the great advantage of allowing users to customize the global algorithm without accessing the internal implementation in Kratos.

In order to implement the elemental formulation a new `Element` has to be created. The `Element` should provide different contributions for each solution step. This is achieved by passing the current fractional step number as a variable of the `ProcessInfo` to the calculation method of the `Element`. Interestingly, this can be done without any modification of the standard elemental interface. This is one of the cases where the generality of the interface helps to integrate new types of formulations. The following code shows the structure of the calculation method for this `Element`:

```
void Fluid3D::CalculateLocalSystem(MatrixType& rLeftHandSideMatrix,
                                   VectorType& rRightHandSideVector,
                                   ProcessInfo& rCurrentProcessInfo)
```

```
{
  KRATOS_TRY

  int FractionalStepNumber = rCurrentProcessInfo[FRACTIONAL_STEP];

  if(FractionalStepNumber <= 3)
  {
      int ComponentIndex = FractionalStepNumber - 1;
      Stage1(rLeftHandSideMatrix,
             rRightHandSideVector,
             rCurrentProcessInfo,
             ComponentIndex);
  }
  else if (FractionalStepNumber == 4)
  {
      Stage2(rLeftHandSideMatrix,
             rRightHandSideVector,
             rCurrentProcessInfo);
  }

  KRATOS_CATCH("")
}
```

Where `Stage1` and `Stage2` are private methods.

10.1.3 Benchmark

Kratos is a general purpose code. Therefore, it is expected to show a lower performance than codes optimized for a single purpose. A well optimized implementation can reduce the performance overhead to the amount introduced by Kratos. An effort was made to optimize the implementation mentioned above, so it is interesting to compare its performance against existing fluid solvers in order to estimate the order of performance overhead introduced by Kratos.

As usual it is not trivial to perform a good benchmark as each program implements a slightly different formulation. Nevertheless comparison was possible with the code Zephyr, an in house program in UPC, and with FEFLO a highly optimized fluid solver developed at the Laboratory for Computational Physics and Fluid Dynamics (LCPFD) in George Washington University at Washington, DC [56]. For the first case the formulation is exactly the same with only minor differences in the implementation. The second solver is an edge based formulation and the only possible comparison was with a predictor corrector scheme.

The benchmark represents the analysis of a three dimensional cylinder at Reynolds number $Re = 190$. Figure 10.1 shows the model used. The no slip boundary condition is used at the walls of the cylinder, while slip conditions are used everywhere else. The inflow velocity is set to $1m/s$. The mesh provided by Prof. R. Löhner with resolved boundary layer and contains 30000 nodes and $108k$ elements. Figure 10.2 shows the mesh used for this test. The values computed were the lift and drag history for the cylinder.

Figures 10.3 and 10.4 show the pressure and velocity calculated by Kratos. The results showed an excellent agreement with the values calculated by FEFLO both in term of peak values and of shedding frequency, as can be seen in figures 10.5 and 10.6.

The timing results are interesting. FEFLO appeared to be 50% faster than Kratos. This is considered a good result taking in account that FEFLO features a highly optimized edge based data structure while Kratos is purely element based.

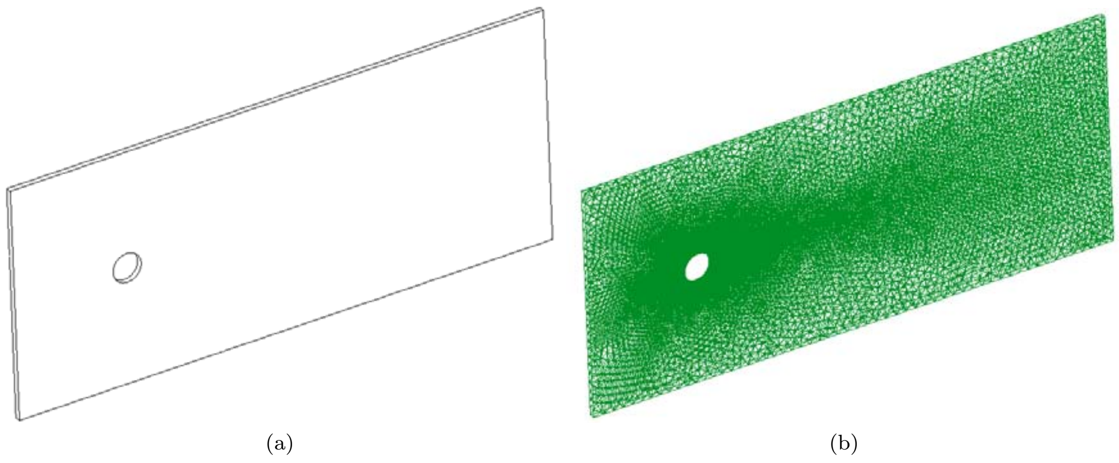


Figure 10.1: a) Geometry of cylinder example, domain dimension $19.0 \times 8.0 \times 0.2$ and $R_c = 0.5$. b) the mesh used.

On the other hand, Zephyr features a element based formulation and implements the same fractional step. The main difference was the treatment of the projection terms and the use of four integration points for the calculation of the element contributions in Zephyr. The results showed that Kratos is about 100% faster with a solution time around one half of the solution time of Zephyr.

10.2 Fluid-Structure Interaction

Coupled problems can be naturally implemented inside Kratos taking advantage of the Python interface. The fluid solver and the structural solver can be implemented separately and coupled using this interface without any problem. The first action required to solve a fluid structure interaction (FSI) problem is to load the different applications involved. The following code shows

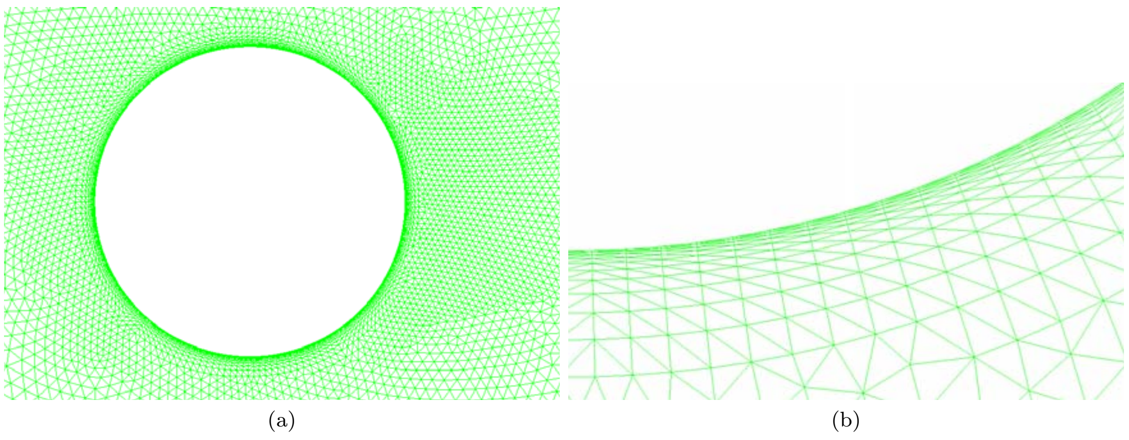


Figure 10.2: Detail of the mesh used for the cylinder example.

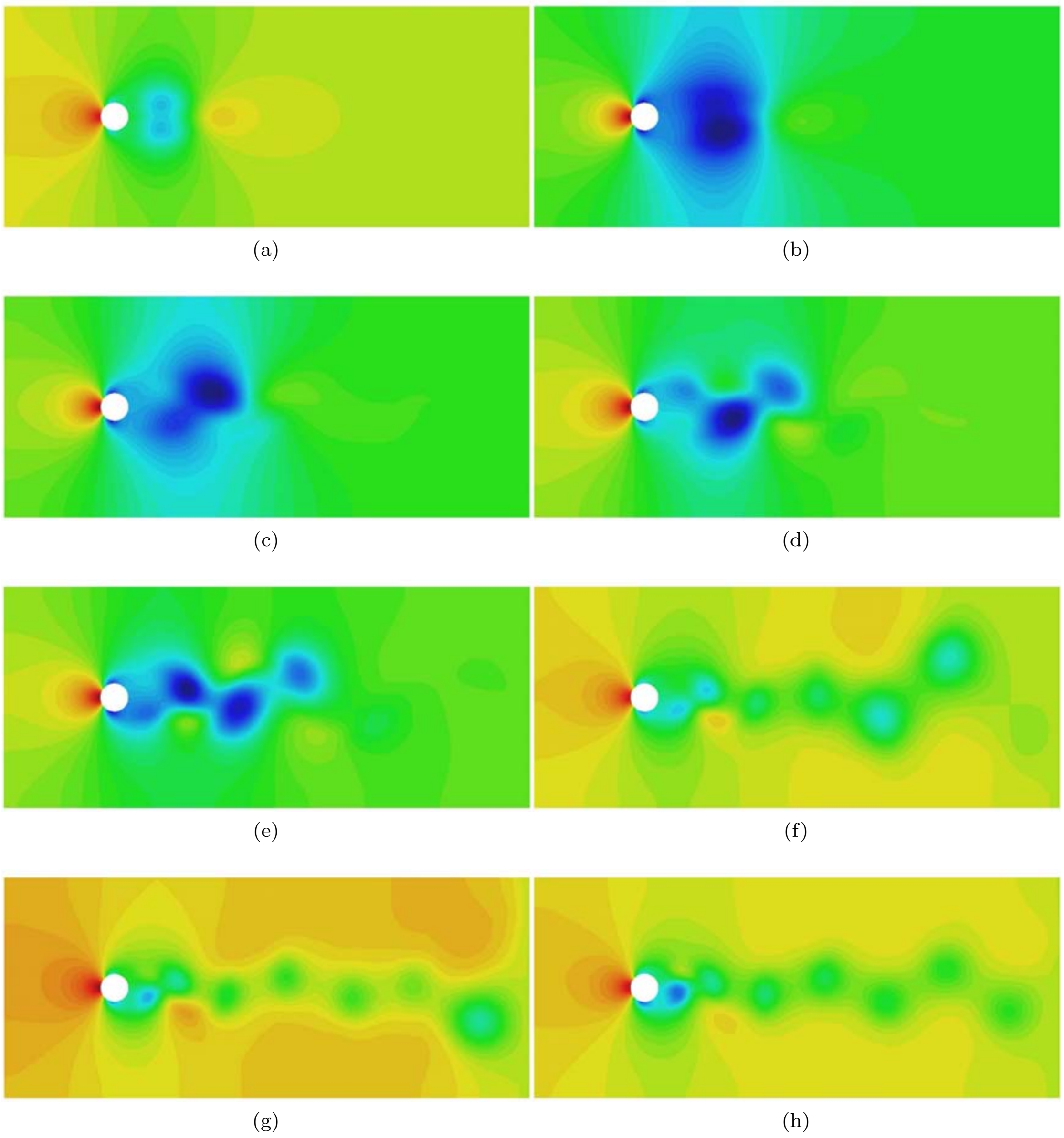


Figure 10.3: Pressure at different time steps.

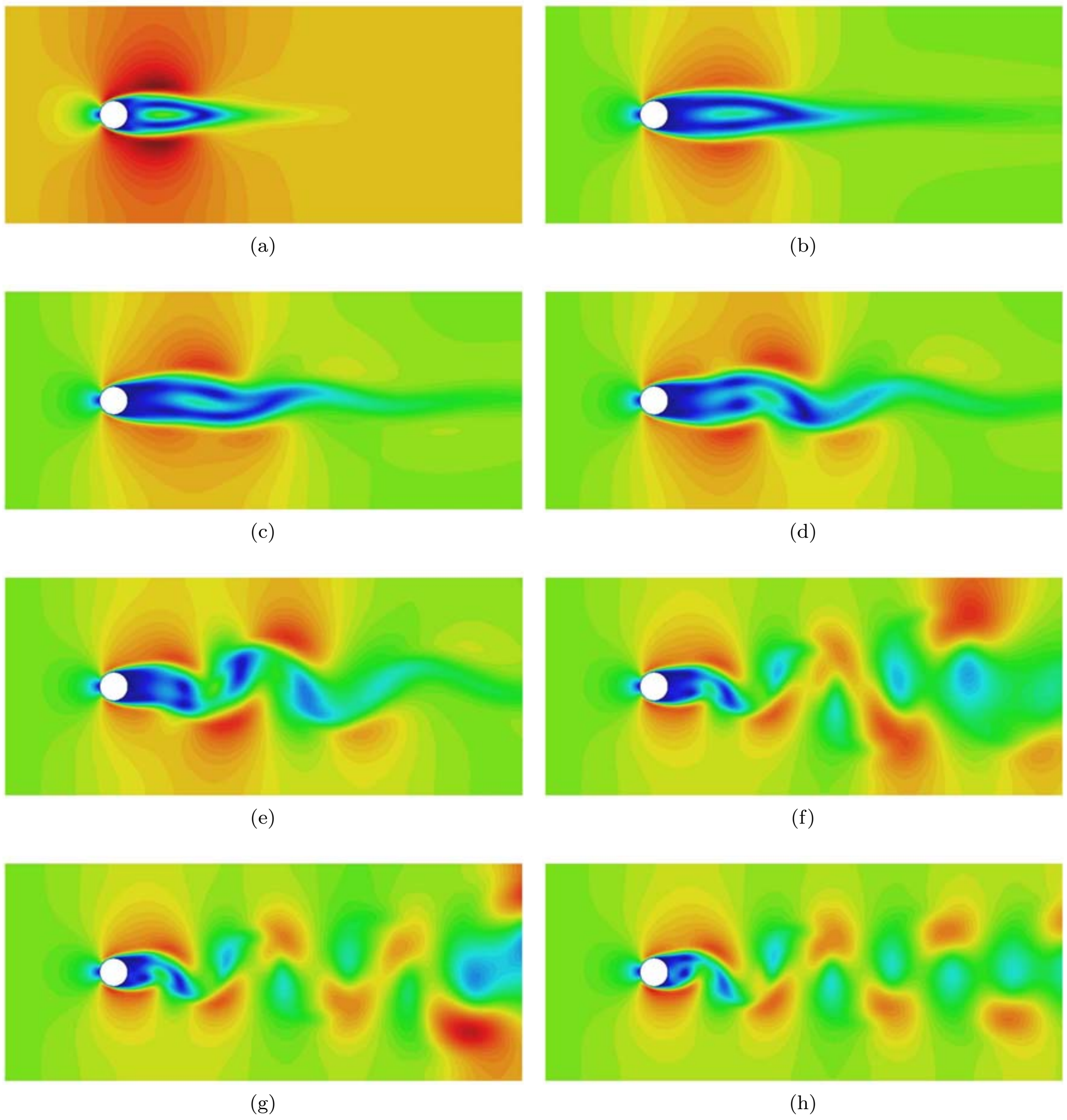


Figure 10.4: Velocity at different time steps.

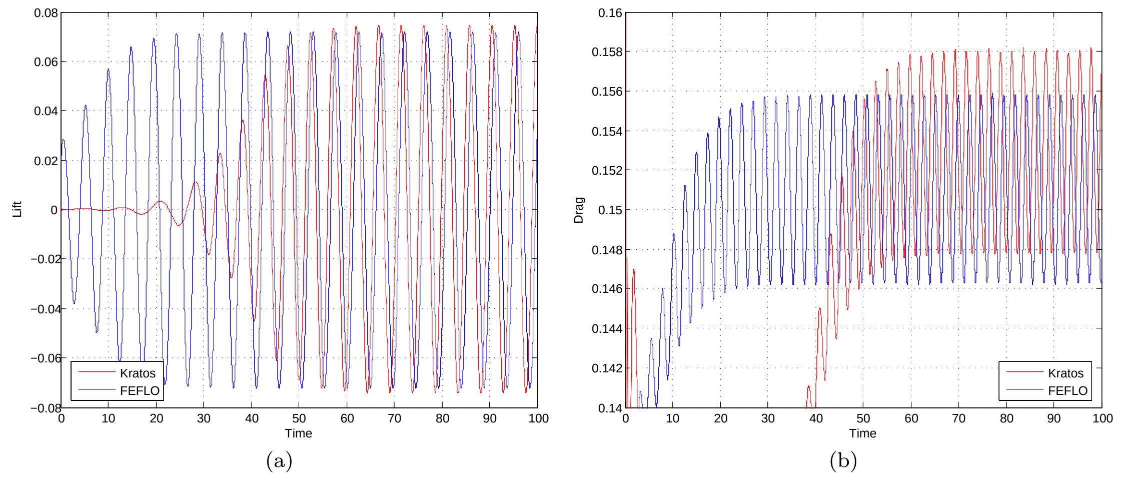


Figure 10.5: The comparison of results obtained by Kratos (red line) using a fractional step algorithm and FEFLO (blue line). a) Lift calculated for the cylinder b) Drag calculated for the cylinder.

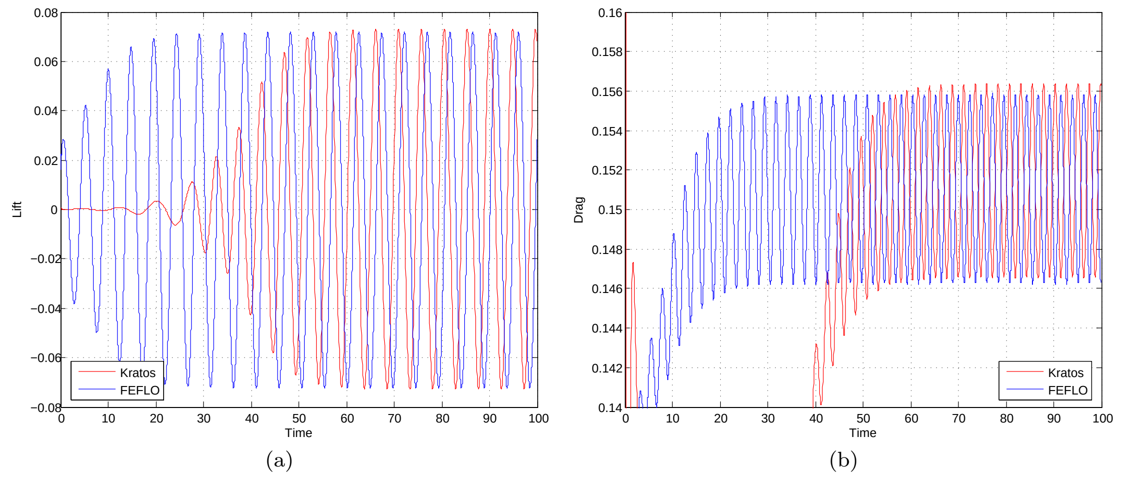


Figure 10.6: The comparison of results obtained by Kratos (red line) using a predictor corrector scheme and FEFLO (blue line). a) Lift calculated for the cylinder b) Drag calculated for the cylinder.

this step in Python:

```
#including kratos path
kratos_libs_path = 'kratos/libs/'
kratos_applications_path = 'kratos/applications/'
import sys
sys.path.append(kratos_libs_path)
sys.path.append(kratos_applications_path)

#importing Kratos main library
from Kratos import *
kernel = Kernel()   #defining kernel

#importing applications
import applications_interface
applications_interface.Import_ALEApplication = True
applications_interface.Import_IncompressibleFluidApplication = True
applications_interface.Import_StructuralApplication = True
applications_interface.Import_FSIApplication = True
applications_interface.ImportApplications(kernel, kratos_applications_path)
```

Then a very simple explicit coupling procedure can be expressed as:

```
class ExplicitCoupling:

  def Solve(self):

    # solve the structure (prediction)
    (self.structural_solver).Solve()

    ## map displacements to the structure
    (self.mapper).StructureToFluid_VectorMap(DISPLACEMENT,DISPLACEMENT)

    ## move the mesh
    (self.mesh_solver).Solve()

    ## set the fluid velocity at the interface to
    ## be equal to the corresponding mesh velocity
    self.CopyVectorVar(MESH_VELOCITY,VELOCITY,self.interface_fluid_nodes);

    ## solve the fluid
    (self.fluid_solver).Solve()

    ## map displacements to the structure
    (self.mapper).FluidToStructure_ScalarMap(PRESSURE,POSITIVE_FACE_PRESSURE)

    # solve the structure (correction)
    (self.structural_solver).Solve()
```

Of course, many different coupling schemes can be implemented without making changes to the single field solvers. An example of fluid-structure interaction is given in figure 10.7.

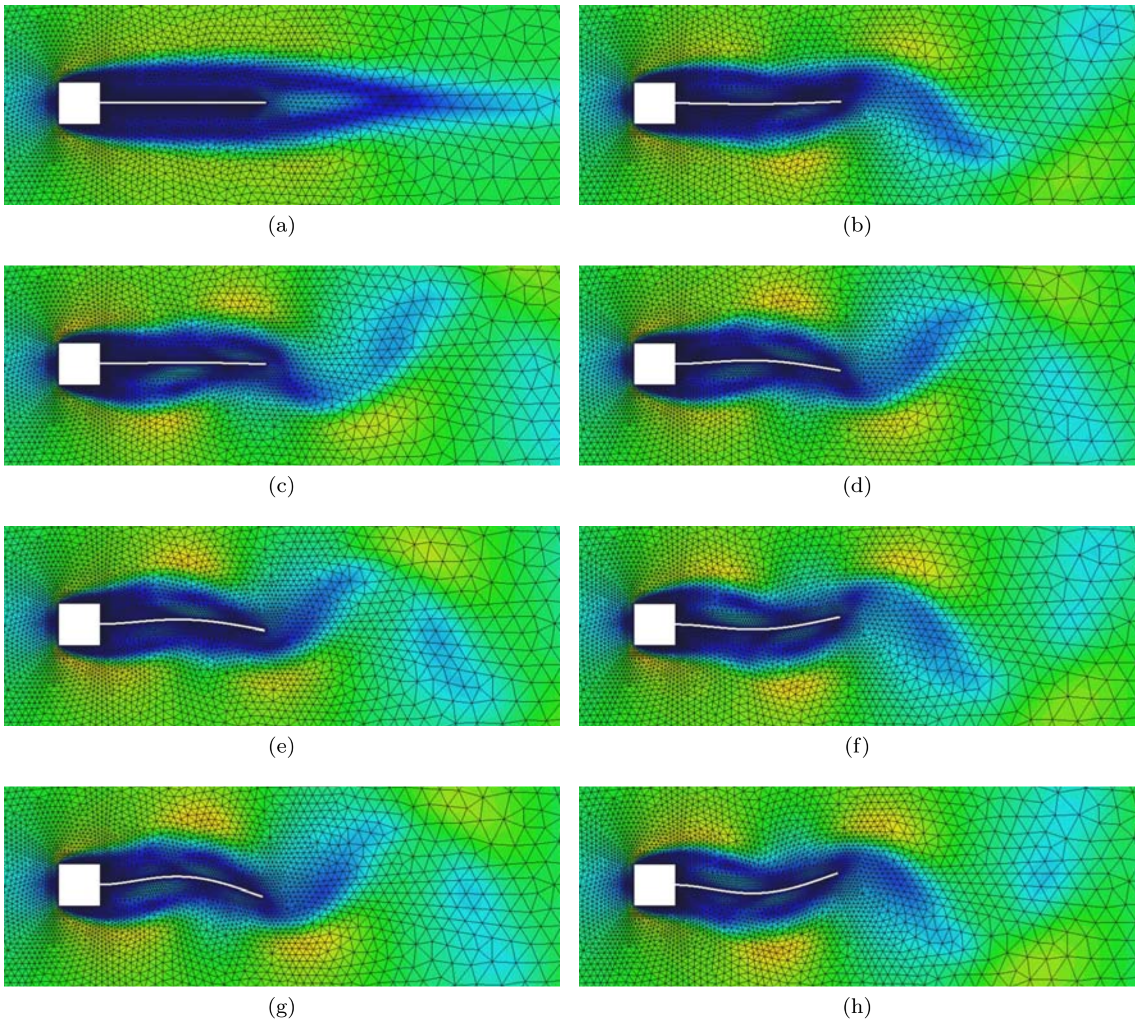


Figure 10.7: Flag flatter simulation using fluid structure interaction with mesh movement.

10.3 Particle Finite Element Method

The Particle Finite Element Method (PFEM) [76, 51, 50, 49] is a method for the solution of fluid problems on arbitrarily varying domains. The basic concept is that each particle is followed in a lagrangian way and the mesh is regenerated at each time step.

The main computational challenges faced to are the efficient regeneration of the mesh and the optimized recalculation of all elemental contributions. Good performance is achieved by linking with an external mesh generation library and by using the optimized Kratos fluid solver. The solution sequence is controlled by the Python interface. Here a part of the Python script is given:

```
def Solve(self,time,gid_io):

    self.PredictionStep(time)
    self.FluidSolver.Solve()

def PredictionStep(self,time):
    domain_size = self.domain_size

    # performing a first order prediction of the fluid displacement
    (self.PfemUtils).Predict(self.model_part)
    self.LagrangianCorrection()
    (self.MeshMover).Execute();

    (self.PfemUtils).MoveLonelyNodes(self.model_part)
    (self.MeshMover).Execute();

    ## ensure that no node gets too close to the walls
    (self.ActOnWalls).Execute();

    ## move the mesh
    (self.MeshMover).Execute();

    ## smooth the final position of the nodes to
    ## homogenize the mesh distribution
    (self.CoordinateSmoother).Execute();

    ## move the mesh
    (self.MeshMover).Execute();

    # regenerate the mesh
    self.Remesh()
```

This example shows how a previously implemented fluid solver can be reused when implementing a new algorithm. This reusability allows the fast development of new formulations which can be tested by solving large scale real-life problems. Figure 10.8 shows a dam break simulation done with the PFEM application implemented in Kratos [57].

10.4 Thermal Inverse Problem

Inverse problems are found in many areas of science and engineering. They can be described as being opposite to direct problems. In a direct problem the cause is given, and the effect is determined. In an inverse problem the effect is given, and the cause must be estimated [52]. There are two main types of inverse problems: input estimation problems, in which the system properties and output are known and the input is to be estimated; and properties estimation problems, in which the the system input and output are known and the properties are to be estimated [52].

Mathematically, inverse problems fall into the more general class of variational problems. The aim of a variational problem is to find a function which minimizes the value of a specified functional. By a functional, we mean a correspondence which assigns a number to each function belonging to some class. Also, inverse problems might be *ill-posed* in which case the solution might not meet existence, uniqueness or stability requirements.

While some simple inverse problems can be solved analytically, the only practical technique for general problems is to approximate the solution using direct methods. The fundamental idea underlying the so called direct methods is to consider the variational problem as a limit problem

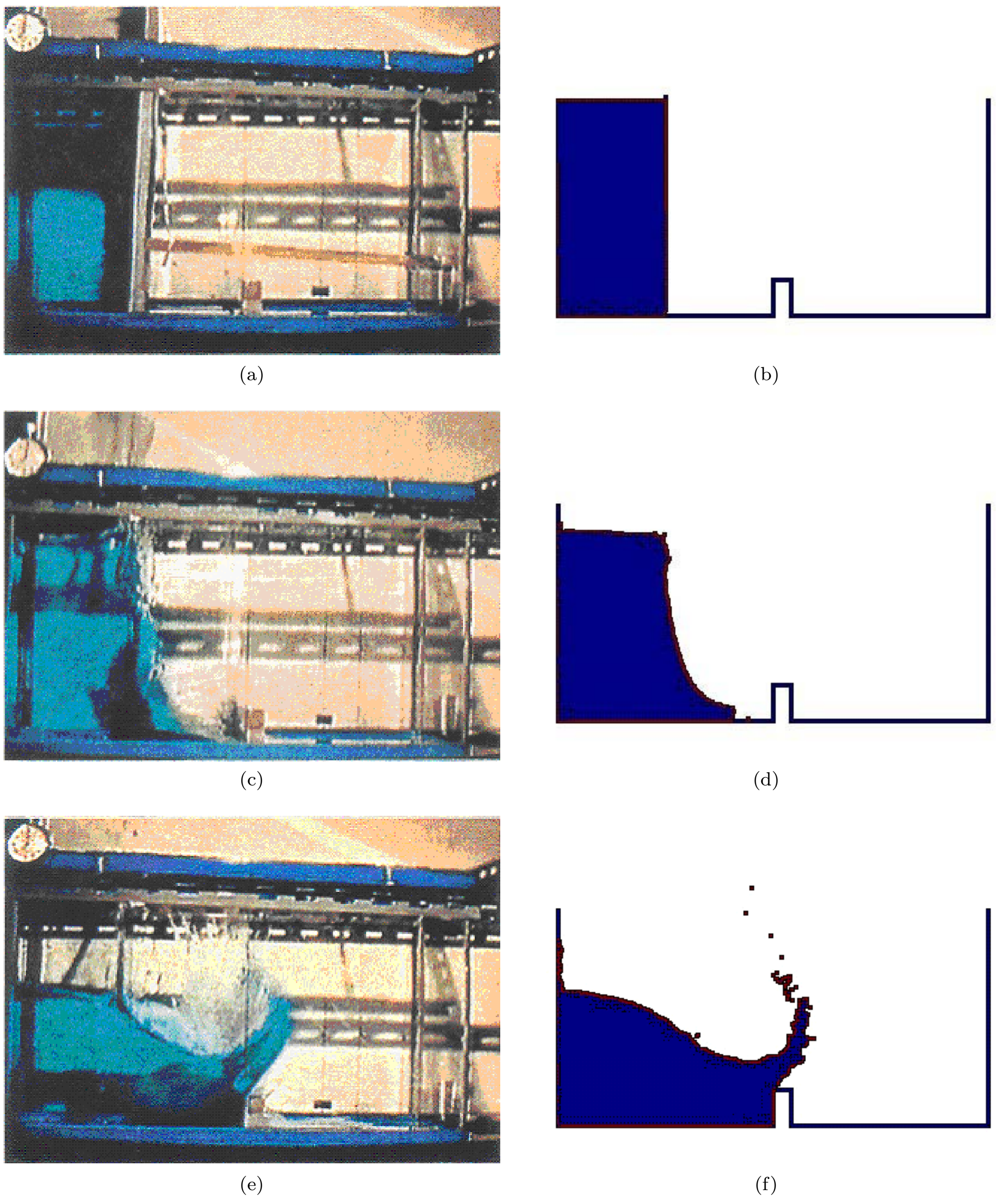
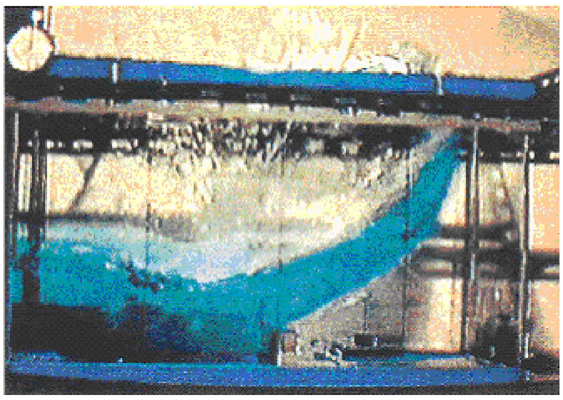
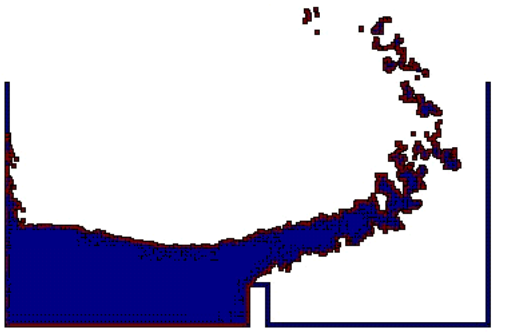
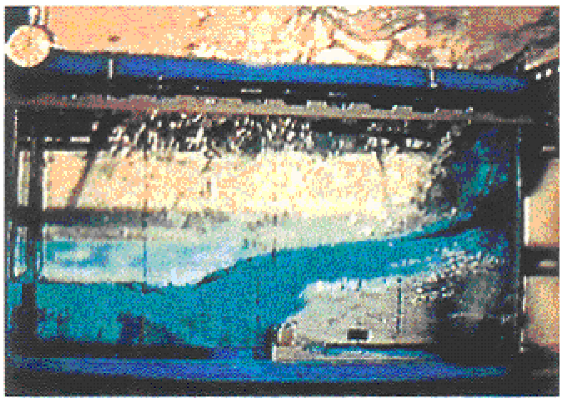
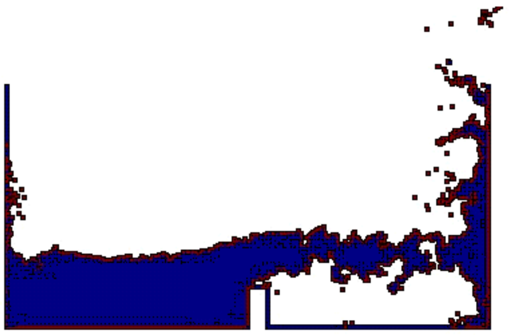


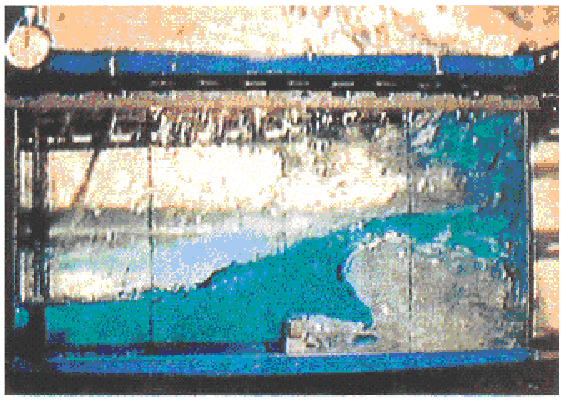
Figure 10.8: A Dam break test and its simulation by the PFEM implementation in Kratos. [57]


(a)


(b)


(c)


(d)


(e)

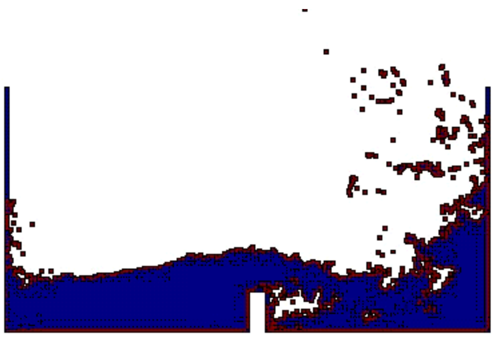

(f)

Figure 10.9: Dam break, continued. [57]

for some function optimization problem in many dimensions. Unfortunately, due to both their variational and ill-posed nature, inverse problems are difficult to solve.

Neural networks is one of the main fields of artificial intelligence [47]. There are many different types of neural networks, of which the *multilayer perceptron* is an important one [94]. Neural networks provide a direct method for the solution of general variational problems and consequently inverse problems [30].

In this example neural networks are used to solve thermal inverse problems. In order to solve this problem we need to solve the heat transfer equation. The *Flood* library [60] developed at CIMNE is an open source neural networks library written in C++ and used to create the neural network necessary for solving this problem. While this library does not include utilities for solving partial differential equations, it uses Kratos and its thermal application to solve the thermal problem. This example validates the integrability of Kratos as a library in another project. It also demonstrates its robustness due to the fact that this algorithm runs Kratos to analyze the same model several times. In this situation any small problem (for example, in memory management) might cause an execution error.

10.4.1 Methodology

The general solution of variational problems using Neural networks consists of three steps [**?**]:

- Definition of the functional space. The solution here is to be represented by a multilayer perceptron.
- Formulation of the variational problem. For their effect a performance functional $F(y(x,a))$ must be defined. In order to evaluate that functional we need to solve a partial differential equation which is done using the FEM within Kratos.
- Solution of the reduced function optimization problem f(a). This is achieved by the training algorithm. The training algorithm will evaluate the performance function $f(a)$ many times.

Figure 10.10 shows these three steps.

This algorithm provides an example of how Kratos can be embedded into an optimization application in which different steps of finite element analysis are necessary to achieve the solution.

Kratos has been embedded inside the Flood library as its solving engine in order to calculate the solution of partial differential equations.

Here the above methodology is applied to solve two different thermal inverse problems.

10.4.2 Implementation

The Flood library uses Kratos to solve a thermal problem several times with different properties and boundary conditions. In order to do this it has to access to internal data of Kratos and change the boundary conditions assigned to the different `Nodes`. This is done without any file interface which would dramatically reduce performance. The first part is the interface for the direct solution using Kratos. The following code shows the main part of this interface:

```
// Initializing Kratos kernel
Kratos::Kernel kernel; kernel.Initialize();

// Initializing Kratos thermal application
Kratos::KratosThermalApplication kratosThermalApplication;
kernel.AddApplication(kratosThermalApplication);
```

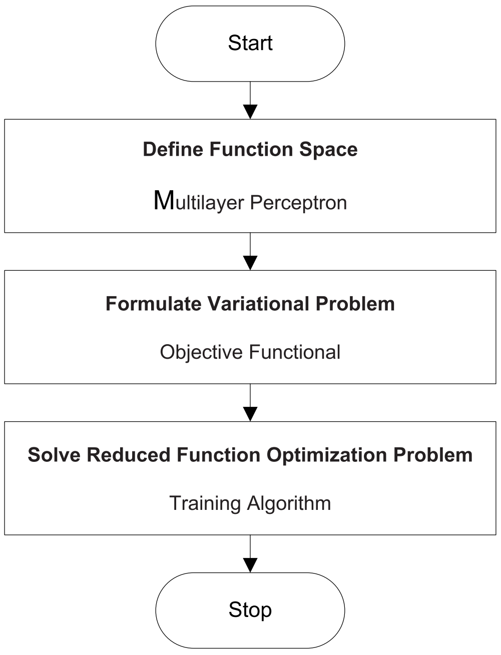


Figure 10.10: General solution of variational problems using neural networks consists of three main steps.

```
// Read mesh
Kratos::GidIO gidIO("thermal_problem"); gidIO >> mesh;

// Set properties
mesh.GetProperties(1)[DENSITY] = density;
mesh.GetProperties(1)[SPECIFIC_HEAT_RATIO] = specificHeat;
mesh.GetProperties(1)[THERMAL_CONDUCTIVITY] = thermalConductivity;

// Assign initial temperature
for(MeshType::NodeIterator i_node = mesh.NodesBegin();
    i_node != mesh.NodesEnd(); i_node++)
  if(!(i_node->IsFixed(TEMPERATURE)))
    i_node->GetSolutionStepValue(TEMPERATURE) = initialTemperature;

// Creating solver
// ...

// Main loop
for(int i = 1; i < numberOfTimeSteps; i++) {
   // Obtain time
   time[i] = time[i-1] + deltaTime;

   // Obtain boundary temperature
   // Gaussian function
```

```
    double mu = 0.5;
    double sigma = 0.05;

    double numerator = exp(-pow(time[i]-mu,2)/(2.0*pow(sigma,2)));
    double denominator = 8*sigma*sqrt(2.0*pi);

    boundaryTemperature[i] = numerator/denominator;

    // Assign boundary temperature
    for(MeshType::NodeIterator i_node = mesh.NodesBegin() ;
        i_node != mesh.NodesEnd() ; i_node++)
      if(i_node->IsFixed(TEMPERATURE))
        i_node->GetSolutionStepValue(TEMPERATURE) =
                                        boundaryTemperature[i];

    // Solving using thermal solver
    Solve();

    // Now updating the nodal temperature values
    // by result of solved equation system.
    Update();

    // Obtain node temperature
    nodeTemperature[i] =
      mesh.GetNode(nodeIndex).GetSolutionStepValue(TEMPERATURE);

    equation_system.ClearData();
}
```

The part initializing the solver has been removed to make the sample code shorter and only the parts that Flood uses to interact with Kratos are kept. This code shows the flexible but clear and intuitive interface that Kratos provides for other applications to communicate with it. First, application changes the `Element` properties to its prescribed values. Then, it changes the temperature value for all `Node`s to some initial value. Afterwards it tries to solve using different boundary conditions assigning fixed values of temperature to `Node`s. Finally it takes the temperature at a specific `Node`. It can be seen that some steps are directly inside the time loop. This restrict us from solving the problem using usual time processes.

The inverse problem also needs similar steps but in the form of performance functions of the Flood library. In this case Kratos is adapted to the working methodology of Flood without problems.

10.4.3 The Boundary Temperature Estimation Problem

For the boundary temperature estimation problem, consider the heat equation in the square domain $\Omega = \{(x,y) : |x| \leq 0.5, |y| \leq 0.5\}$ with boundary $\Gamma = \{(x,y) : |x| = 0.5, |y| = 0.5\}$,

$$\nabla^2 u(x,y;t) = \frac{\partial u(x,y;t)}{\partial t} \quad in \quad \Omega, \tag{10.1}$$

for $t \in [0,1]$, with the initial condition $u(x,y;0) = 0$ in Ω. The problem is to estimate the boundary temperature $y(t)$ on Γ and for $t \in [0,1]$, from measurements of the temperature at the center of the square $u(0,0;t)$ for $t \in [0,1]$,

$$\begin{array}{cc} t_1 & u_1(0,0;t_1) \\ t_2 & u_2(0,0;t_2) \\ \vdots & \vdots \\ t_P & u_P(0,0;t_P) \end{array}$$

where P is the number of time steps considered. For this problem we use 101 time steps.

The first stage in solving this problem is to choose a network architecture to represent the boundary temperature $y(t)$ for $t \in [0,1]$. Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used [47]. This neural network is a class of universal approximator [48]. The network must have one input and one output neuron. We guess a good number of neurons in the hidden layer to be six. This neural network spans a family V of parameterized functions $y(t;\underline{\alpha})$ of dimension $s = 19$, which is the number of free parameters in the network.

The second stage is to derive a performance functional in order to formulate the variational problem. This is to be the mean squared error between the computed temperature at the center of the square for a given boundary temperature and the measured temperature at the center of the square,

$$F[y(t;\underline{\alpha})] \quad = \quad \frac{1}{P}\sum_{i=1}^{P}\left(\hat{u}_{y(t;\underline{\alpha})}(0,0;t_i) - u_i(0,0;t_i)\right)^2. \qquad (10.2)$$

Please note that evaluation of the performance functional (10.2) requires a numerical method for solving partial differential equations. Kratos is used here to solve this problem.

The boundary temperature estimation problem for the multilayer perceptron can then be formulated as follows:

> *Let V be the space consisting of all functions $y(t;\underline{\alpha})$ spanned by a multilayer perceptron with 1 input, 6 sigmoid neurons in the hidden layer and 1 linear output neuron. The dimension of V is 19. Find a vector of free parameters $\underline{\alpha}^* \in \mathbf{R}^{19}$ that addresses a function $y^*(t;\underline{\alpha}^*) \in V$ for which the functional (10.2), defined on V, takes on a minimum value.*

The third stage in solving this problem is to choose a suitable training algorithm. Here we use a conjugate gradient with Polak-Ribiere train direction and Brent optimal train rate [24]. The tolerance in the Brent's method is set to 10^{-6}. Training of the neural network with the conjugate gradient algorithm requires the evaluation of the performance function gradient vector $\nabla f(\underline{\alpha})$. This is carried out by means of numerical differentiation [24]. In particular, we use the symmetrical central differences method [24] with an ϵ value of 10^{-6}.

In this example, we set the training algorithm to stop when the norm of the performance function gradient $\nabla f(\underline{\alpha})$ falls below 10^{-6}. That means that the necessary condition for a local minimum has been satisfied. The neural network is initialized with a vector of free parameters chosen at random in the interval $[-1,1]$. During the training process the performance function decreases until the stopping criterium is satisfied. Table 10.1 shows the training results for this problem. Here N denotes the number of epochs, M the number of performance evaluations, $F[y^*(t;\underline{\alpha}^*)]$ the final performance, and $\nabla f(\underline{\alpha}^*)$ the final performance function gradient norm.

Figure 10.11 shows the actual boundary temperature, the boundary temperature estimated by the neural network, and the measured temperature at the center of the square for this problem.

N	M	$F[y^*(t;\underline{\alpha}^*)]$	$\nabla f(\underline{\alpha}^*)$
421	25352	$2.456 \cdot 10^{-4}$	$8.251 \cdot 10^{-7}$

Table 10.1: Training results for the boundary temperature estimation problem.

The solution here is good, since the estimated boundary temperature matches the actual boundary temperature.

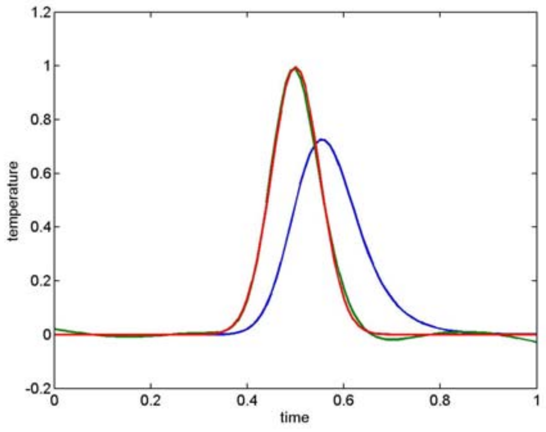


Figure 10.11: Actual boundary temperature (red), estimated boundary temperature (green) and measured temperature at the center of the square (blue) for the boundary temperature estimation problem.

10.4.4 The Diffusion Coefficient Estimation Problem

For the diffusion coefficient estimation problem, consider the equation of diffusion in an inhomogeneous medium in the square domain $\Omega = \{(x,y) : |x| \leq 0.5, |y| \leq 0.5\}$ with boundary $\Gamma = \{(x,y) : |x| = 0.5, |y| = 0.5\}$,

$$\nabla\left(\kappa(x,y)\nabla u(x,y;t)\right) = \frac{\partial u(x,y;t)}{\partial t} \quad in \quad \Omega, \tag{10.3}$$

for $t \in [0,1]$ and where $\kappa(x,y)$ is called the diffusion coefficient, with boundary condition $u(x,y;t) = 0$ on Γ and for $t \in [0,1]$, and initial condition $u(x,y;0) = 1$ in Ω. The problem is to estimate the diffusion coefficient $\kappa(x,y)$ in Ω, from measurements of the temperature at different points on the square $u(x,y;t)$ in Ω and for $t \in [0,1]$,

$$\begin{array}{cccccc}
t_1 & u_{11}(x_1,y_1;t_1) & u_{12}(x_2,y_2;t_1) & \ldots & u_{1Q}(x_Q,y_Q;t_1) \\
t_2 & u_{21}(x_1,y_1;t_2) & u_{22}(x_2,y_2;t_2) & \ldots & u_{2Q}(x_Q,y_Q;t_2) \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
t_P & u_{P1}(x_1,y_1;t_P) & u_{P2}(x_2,y_2;t_P) & \ldots & u_{PQ}(x_Q,y_Q;t_P)
\end{array}$$

where P and Q are the number of points and time steps considered, respectively. For this problem we use 485 points and 11 time steps.

The first stage in solving this problem is to choose a network architecture to represent the diffusion coefficient $\kappa(x, y)$ in Ω. Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used. This neural network is a class of universal approximator [48]. The neural network must have two inputs and one output neuron. We guess a good number of neurons in the hidden layer to be six. This neural network is denoted as a $2 : 6 : 1$ multilayer perceptron. It spans a family V of parameterized functions $\kappa(x, y; \underline{\alpha})$ of dimension $s = 25$, which is the number of free parameters in the network.

The second stage is to derive a performance functional for the diffusion coefficient estimation problem. The performance functional for this problem is to be the mean squared error between the computed temperature for a given diffusion coefficient and the measured temperature,

$$E[\kappa(x, y; \underline{\alpha})] \quad = \quad \frac{1}{PQ}\sum_{i=1}^{P}\left(\sum_{j=1}^{Q}\left(\hat{u}_{\kappa(x,y;\underline{\alpha})}(x_j, y_j; t_i) - u_{ij}(x_j, y_j; t_i)\right)^2\right), \qquad (10.4)$$

The diffusion coefficient estimation problem for the multilayer perceptron can then be formulated as follows:

Let V be the space consisting of all functions $\kappa(x, y; \underline{\alpha})$ spanned by a multilayer perceptron with 2 inputs, 6 sigmoid neurons in the hidden layer and 1 linear output neuron. The dimension of V is 25. Find a vector of free parameters $\underline{\alpha}^* \in \mathbf{R}^{25}$ that addresses a function $\kappa^*(x, y; \underline{\alpha}^*) \in V$ for which the functional (10.4), defined on V, takes on a minimum value.

Evaluation of the performance functional (10.4) requires a numerical method for integration of partial differential equations. Here we choose the Finite Element Method [104]. For this problem we use a triangular mesh with 888 elements and 485 nodes.

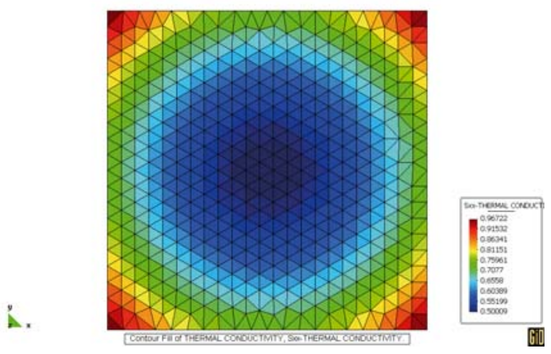
The third stage is to choose a suitable algorithm for training. Here we use a conjugate gradient with Polak-Ribiere train direction and Brent optimal train rate methods for training [24]. The tolerance in the Brent's method is set to 10^{-6}. Training of the neural network with the conjugate gradient algorithm requires the evaluation of the performance function gradient vector $\nabla f(\underline{\alpha})$ [24]. This is carried out by means of numerical differentiation. In particular, we use the symmetrical central differences method [24] with $\epsilon = 10^{-6}$.

In this example, we set the training algorithm to stop when the norm of the performance function gradient falls below 10^{-6}. That means that the necessary condition for a local minimum has been satisfied. The neural network is initialized with a vector of free parameters chosen at random in the interval $[-1, 1]$. Table 10.1 shows the training results for this problem. Here N denotes the number of epochs, M the number of performance evaluations, $F[\kappa^*(x, y; \underline{\alpha}^*)]$ the final performance, and $\nabla f(\underline{\alpha}^*)$ the final performance function gradient norm.

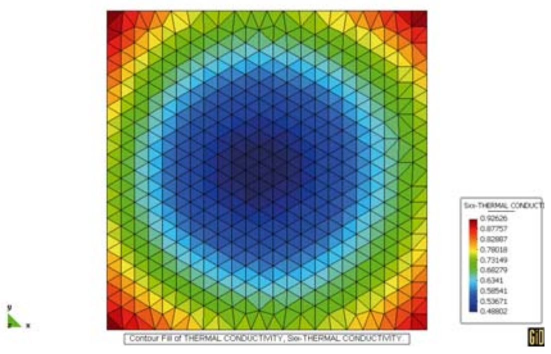
N	M	$F[\kappa^*(x, y; \underline{\alpha}^*)]$	$\nabla f(\underline{\alpha}^*)$
312	22652	$1.562 \cdot 10^{-5}$	$7.156 \cdot 10^{-7}$

Table 10.2: Training results for the diffusion coefficient estimation problem.

Figure 10.12 shows the actual diffusion coefficient and the diffusion coefficient estimated by the neural network for this problem. The solution here is good, since the estimated diffusion coefficient matches the actual diffusion coefficient.



(a)



(b)

Figure 10.12: Actual diffusion coefficient a) and estimated diffusion coefficient b) for the diffusion coefficient estimation problem.

Further applications and other class of problems can be found in [73, 85, 86, 88, 87, 29].