Tutorial

M. Schick, C. Song

Solving the Poisson equation with uncertain parameters using the Spectral-Stochastic-Finite-Element-Method

April 16th, 2015

Version 1.5

http://www.hiflow3.org

Contents

1 Introduction 2
1.1 How to use the Tutorial . . . 2
1.1.1 Using HiFlow^3 as a Developer . . . 2

2 Mathematical Setup 3
2.1 Problem . . . 3
2.2 Using Polynomial Chaos as a basis for stochastic part . . . 4
2.3 Discretization by stochastic Galerkin projection . . . 5
2.4 Numerical methods for the solution of the linear systems . . . 6

3 The Commented Program 7
3.1 Preliminaries . . . 7
3.2 Parameter File . . . 7
3.3 Main Function . . . 8
3.4 Member Functions . . . 9
3.4.1 run() . . . 10
3.4.2 setup_mesh() . . . 10
3.4.3 setup_application() . . . 11
3.4.4 setup_space() . . . 12
3.4.5 setup_system() . . . 12
3.4.6 setup_la_structure() . . . 12
3.4.7 setup_linear_solver() . . . 14
3.4.8 setup_bc() . . . 14
3.4.9 solve_linear_system() . . . 16
3.4.10 compute_matrix() . . . 16
3.4.11 compute_residual() . . . 17
3.4.12 visualization_solution() . . . 19

4 Program Output 20
4.1 Sequential Mode . . . 20
4.2 Parallel Mode . . . 20
4.3 Visualization of the Solution . . . 21

5 Numerical Example 22
5.1 Two-Dimensional Example . . . 22

Bibliography 23

Using HiFlow^3 for solving the Poisson equation with uncertain parameters

1 Introduction

HiFlow^3 is a multi-purpose finite element software providing powerful tools for efficient and accurate solution of a wide range of problems modeled by partial differential equations (PDEs). Based on object-oriented concepts and the full capabilities of C++ the HiFlow^3 project follows a modular and generic approach for building efficient parallel numerical solvers. It provides highly capable modules dealing with the mesh setup, finite element spaces, degrees of freedom, linear algebra routines, numerical solvers, and output data for visualization. Parallelism - as the basis for high performance simulations on modern computing systems - is introduced on two levels: coarse-grained parallelism by means of distributed grids and distributed data structures, and fine-grained parallelism by means of platform-optimized linear algebra back-ends.

1.1 How to use the Tutorial

You find the example code (`poisson_uncertainty.cc`, `poisson_uncertainty.h`), a parameter file (`poisson_uncertainty/poisson_uncertainty.xml`). The geometry data (`*.inp`) is stored in the folder `/hiflow/examples/data/`.

1.1.1 Using HiFlow^3 as a Developer

First build and compile HiFlow^3. Go to the directory `/build/examples/poisson_uncertainty`, where the binary poisson uncertainty tutorial is stored. Type **./poisson_uncertainty poisson_uncertainty.xml** to execute the program in sequential mode, To execute in parallel mode with four processes for spatial decomposition and two processes for stochastic modes, type **mpirun -np 8 ./poisson_uncertainty poisson_uncertainty.xml 4** . In both cases,you need to make sure that the default parameter file `poisson_uncertainty.xml` is stored in the same directory as the binary, and that the geometry data specified in the parameter file is stored in `/hiflow/examples/data`. Alternatively, you can specify the path of your own xml-file with the name of your xml-file in the comment line, i.e. **mpirun -np 8 ./poisson_uncertainty "path_to_parameterfile/name_of_parameterfile".xml 4**.

2 Mathematical Setup

2.1 Problem

We consider the Poisson equation with uncertain, spatially dependent diffusivity parameter ν. The equations are imposed on the unit-square, i.e., $\Omega = [0,1]^2$. The boundary conditions and the forcing term are assumed to be deterministic.

The governing equations can be expressed by :

$$
\begin{aligned}
-\nabla \cdot (\nu(x,\theta)\nabla u(x,\theta)) &= 1, && x \in \Omega, \theta \in \Theta && (1)\\
u(x,\theta) &= 1, && x \in \Gamma, \theta \in \Theta && (2)\\
\frac{\partial u}{\partial x}(x,\theta) &= 0, && x \in \partial\Omega \backslash \Gamma, \theta \in \Theta && (3)
\end{aligned}
$$

Here, the Dirichlet boundary Γ is defined as $\Gamma = \{x = (x_1, 0) \in \Omega, x_1 \in [0,1]\}$. Θ denotes the abstract sampling space of the uncertain event $\theta \in \Theta$. We introduce an uncertainty model for $\nu(x,\theta)$ by:

$$
\nu(x,\theta) = \nu_0 + \nu_0 \sigma \sum_{i=1}^{M} q^{i-1} \sin(2\pi i x_1) \sin(2\pi i x_2) \xi_i(\theta), \tag{4}
$$

where $\boldsymbol{\xi(\theta)} := (\xi_1(\theta), \ldots, \xi_M(\theta))$ denotes a random vector, components of which are independent and uniformly distributed in the interval $[-1,1]$, i.e., $\xi_i \sim \mathcal{U}(-1,1)$. The mean of $\nu(x,\theta)$ is then $\mathbb{E}(\nu) = \nu_0$ by definition. The stochastic parameter σ denotes the magnitude of fluctuation of the diffusivity around its mean. The parameter q denotes the decay rate of the fluctuations. In order to obtain a well-posed problem, we need to enforce that :

$$
0 < q < 1, \quad \text{and } 0 < \frac{\sigma}{1-q} < 1.
$$

The notation of θ will be dropped in the following equations, due to the fact that we consider only the uncertain parameter vector $\boldsymbol{\xi}$ in our mathematical set-up. In general, the governing PDE could be carried out from (1) - (3) with ξ:

$$
\begin{aligned}
-\nabla \cdot (\nu(x,\boldsymbol{\xi})\nabla u(x,\boldsymbol{\xi})) &= 1, && x \in \Omega, && (5)\\
u(x,\boldsymbol{\xi}) &= 1, && x \in \Gamma, && (6)\\
\frac{\partial u}{\partial x}(x,\boldsymbol{\xi}) &= 0, && x \in \partial\Omega \backslash \Gamma && (7)
\end{aligned}
$$

where the equations are required to hold almost surely in Θ.

2.2 Using Polynomial Chaos as a basis for stochastic part

We assume a square-integrable stochastic solution, i.e., $u(x,\cdot) \in L^2(\Xi)$ for all $x \in [0,1]^2$ with Ξ denoting the range of $\boldsymbol{\xi}$. Then we can express the stochastic solution using a spectral expansion:

$$u(x,\boldsymbol{\xi}) = \sum_{i=0}^{+\infty} u_i(x)\psi_i(\boldsymbol{\xi}), \tag{8}$$

where, $u_i(x)$ denotes the **deterministic part**, and $\psi_i(\boldsymbol{\xi})$ denotes the **stochastic part**. We use the Polynomial Chaos (PC) expansion for choosing the basis functionals $\psi_i(\boldsymbol{\xi})$. In standard formulation, these correspond to multivariate Hermite-Polynomials in terms of Gaussian random variables [4]. However, since we assume a Uniform distribution for $\boldsymbol{\xi}$ we use multivariate Legendre Polynomials instead, since these satisfy an orthogonality condition with respect to the constant probability density function $1/2^M$ of $\boldsymbol{\xi}$:

$$\begin{aligned}
\langle \psi_n, \psi_m \rangle &:= \frac{1}{2^M}\int_{[-1,1]^M} \psi_n(\boldsymbol{\xi})\psi_m(\boldsymbol{\xi})d\boldsymbol{\xi} = \|\psi_n\|^2_{L^2(\Xi)}\delta_{n,m} \\
\mathbb{E}[\psi_0] &= \frac{1}{2^M}\int_{[-1,1]^M} \psi_0(\boldsymbol{\xi})d\boldsymbol{\xi} = 1 \\
\mathbb{E}[\psi_n] &= \frac{1}{2^M}\int_{[-1,1]^M} \psi_n(\boldsymbol{\xi})d\boldsymbol{\xi} = 0, n > 0
\end{aligned}$$

The multivariate Chaos Polynomials can be computed by a tensor product of their one-dimensional counterpart, see for example [5].

The expectation $\mathbb{E}$ and the variance σ^2 of the **stochastic solution** $u(x,\xi)$ can be computed by:

$$\mathbb{E}[u](x) = \frac{1}{2^M}\int_{[-1,1]^M} \sum_{i=0}^{+\infty} u_i(x)\psi_i d\xi = u_0(x) \tag{9}$$

$$\sigma^2[u](x) = \mathbb{E}[u^2](x) - (\mathbb{E}[u](x))^2 = \sum_{i=1}^{+\infty} u_i^2(x)\langle \psi_i, \psi_i \rangle. \tag{10}$$

In order to achieve a finite-dimensional approximation for numerical computation, we need to truncate the stochastic spectral expansion (8). We do so by prescribing a maximum total polynomial degree p for the polynomials $\{\psi_i\}$, such that

$$u(x,\boldsymbol{\xi}) = \sum_{i=0}^{P} u_i(x)\psi_i(\boldsymbol{\xi}), \tag{11}$$

where the number of terms $P+1$ are given by [2]

$$P + 1 = \frac{(p+M)!}{p!M!}. \tag{12}$$

Currently, two distributions are implemented in HiFlow^3, which are stated in Table 1.

Distribution	**pdf**	**Polynomials**	**Support**
Uniform	$\frac{1}{2}$	Legendre	$[-1, 1]$
Gaussian	$\frac{e^{-x^2/2}}{\sqrt{2\pi}}$	Hermite	$[-\infty, +\infty]$

pdf: probability density functions

Table 1: Chaos Polynomials implemented in HiFlow^3.

2.3 Discretization by stochastic Galerkin projection

We introduce a short notation for the stochastic diffusivity parameter $\nu = \nu(x, \boldsymbol{\xi})$ for notational convenience:

$$\nu(x, \boldsymbol{\xi}) = \sum_{i=0}^{M} \nu_i(x) \psi_i(\boldsymbol{\xi}), \tag{13}$$

where

$$\begin{aligned} &\nu_0(x) := \nu_0, \\ &\nu_i(x) := \nu_0 \sigma q^{i-1} \sin(2\pi i x_1) \sin(2\pi i x_2), \quad i = 1, \ldots, M. \end{aligned}$$

Note, that by definition of the Chaos Polynomials, we have

$$\psi_i(\boldsymbol{\xi}) = \xi_i, \quad i = 1, \ldots, M. \tag{14}$$

We illustrate the stochastic Galerkin approach on (5). The boundary conditions can be discretized in a similar way. The stochastic Galerkin projection is based on a variational setting of (5) with respect to the stochastic space $L^2(\Xi)$. By $\mathcal{S}_p \subset L^2(\Xi)$ we denote the space spanned by the Chaos Polynomials up to total degree p, i.e., $\mathcal{S}_p = span\{\psi_0, \psi_1, \ldots, \psi_P\}$.

The corresponding (discrete) variational setting with respect to the stochastic space reads: find $u(x, \cdot) \in \mathcal{S}_p$ such that

$$-\langle \nabla \cdot (\nu(x, \boldsymbol{\xi}) \nabla u(x, \boldsymbol{\xi})), \psi \rangle = \langle 1, \psi \rangle, \quad \forall \psi \in \mathcal{S}_p. \tag{15}$$

for all $x \in [0, 1]^2$. Next we insert (13) and (11) into (15) and get by orthogonality of the Chaos Polynomials

$$-\sum_{i=0}^{M} \sum_{j=0}^{P} \nabla \cdot (\nu_i(x) \nabla u_j(x)) c_{ijk} = \delta_{0,k}, \quad \forall k = 0, \ldots, P, \tag{16}$$

where we define the third order stochastic Galerkin Tensor $\mathbb{T} = (c_{ijk})$ by [2]

$$c_{ijk} := \frac{\langle \psi_i \psi_j, \psi_k \rangle}{\langle \psi_k, \psi_k \rangle}. \tag{17}$$

Finally, we obtained a deterministic set of coupled partial differential equations for the PC modes $u_i(x)$, $i = 0, \dots, P$, which is solved using the Finite-Element-Method.

2.4 Numerical methods for the solution of the linear systems

The user of this tutorial can choose between two types of solution methods: CG (conjugate gradient) with or without mean-based preconditioning, and a stochastic multilevel solver, which is based on the hierarchical multilevel structure of the Chaos Polynomials

$$\mathcal{S}_0 \subset \mathcal{S}_1 \subset \cdots \subset \mathcal{S}_p.$$

The mean-based preconditioner for CG is using an approximation of (16) by means of a block-diagonal approach. Therefore, all terms $\nu_i(x)$ are set to zero for all $i > 0$, such that only the mean of ν is used in the preconditioning step. Therefore, due to orthogonality of the Chaos Polynomials ψ_i the system simplifies to a single deterministic system (the so-called mean system). This is used for a preconditioning step for all $u_i(x)$ in order to speed up the convergence of CG.

In the corresponding xml-file (see section 3.2) with `<LinearSolver>`, this can be done by choosing either `CG` or `ML` for linear solver, and `Preconditioner` could also be modified with according preconditioner respectively.

3 The Commented Program

3.1 Preliminaries

HiFLow^3 is designed for high performance computing on massively parallel machines. So it applies the Message Passing Interface (MPI) library specification for message-passing, see sections 3.3 and 3.4, also [1]. The poisson uncertainty tutorial needs the following two input files:

- `poisson_uncertainty.xml` A xml-file containing all information needed to execute the program. It is read in by the program at runtime and thus does not require the recompilation of the program when parameters are changed. Parameters for example defining the termination condition of the linear solver and preconditioner are listed as well as parameters of polynomial degree and number of uncertain parameter. xml-file needs to be provided explicitly when poisson uncertainty executes, as it is described in section **??**, the default xml-file is referred in section 3.2, which contains the parameters of the two-dimensional numerical example, see section 5. This file is stored in `/hiflow/examples/poisson_uncertainty/`.

- Geometry data: The file containing the geometry is specified in the parameter file. In the example in section 5, we used `unit_square.inp`

HiFlow^3 does not generate meshes for the domain Ω. Meshes in `*.inp` and `*.vtu` format can be read in. It is possible to extend the reader for other formats. Some geometry data is provided in the test/data folder. Furthermore it is possible to generate other geometries by using external programs (Mesh generators) or by hand. Both formats provide the possibility to mark cell or facets by material numbers.

3.2 Parameter File

The needed parameters are initialized in the parameter file `poisson_uncertainty.xml`.

```
<Param>
  <Mesh>
    <Filename>unit_square.inp</Filename>
    <RefinementLevel>6</RefinementLevel>
<!-- <Filename>unit_cube.inp</Filename>-->
<!-- <RefinementLevel>4</RefinementLevel>-->
  </Mesh>
  <GalerkinLinearSolver>
    <LinearSolver>CG</LinearSolver>
<!-- <LinearSolver>Multilevel</LinearSolver>-->
<!-- <Preconditioner>None</Preconditioner>-->
<!-- <Preconditioner>Mean</Preconditioner>-->
    <Preconditioner>ML</Preconditioner>
  </GalerkinLinearSolver>
  <Multilevel>
    <MLType>Matrix</MLType>
    <Nu1>1</Nu1>
    <Nu2>1</Nu2>
    <Mu>1</Mu>
```

```
<!-- <Smoothing>Umfpack</Smoothing>-->
<!-- <Smoothing>InexactPCG</Smoothing>-->
    <Smoothing>InexactCG</Smoothing>
  </Multilevel>
  <Output>
    <InfoFilename>uq_poisson_mpi.info</InfoFilename>
    <MeshFilename>uq_poisson_mesh.vtu</MeshFilename>
    <VisuFolder>./</VisuFolder>
  </Output>
  <PolynomialChaos>
    <No>3</No>
    <N>6</N>
    <q>1</q>
  </PolynomialChaos>
  <Application>
    <MeanViscosity>0.01</MeanViscosity>
    <Variability>0.2</Variability>
    <Decay>0.5</Decay>
  </Application>
</Param>
```

3.3 Main Function

The main function starts the simulation of the stochastic Poisson problem (`poisson_uncertainty.cc`).

```
.
int main ( int argc, char** argv )
{
    MPI_Init ( &argc, &argv );
    std::string fName;
    if ( argc > 1 )
    {
        fName = argv[1];
    }
    else
    {
        std::cout << "Pass XMl parameter file as argument!" << std::↩
            endl;
        std::cout << "Usage: ./poisson_uncertainty <xml-filename>" << ↩
            std::endl;
        exit ( -1 );
    }

    // Setup timer.
    Timer main_timer;

    int num_partitions_space = 1;
    if ( argc > 2 )
        num_partitions_space = std::atoi ( argv[2] );
    // Read parameters
    PropertyTree config ( fName, MASTER_RANK, MPI_COMM_WORLD );

    // Start Logging
    std::ofstream info_file ( config["Output"]["InfoFilename"].get<std↩
        ::string>( ).c_str ( ) );
    LogKeeper::get_log ( "info" ).set_target ( &info_file );
    // LogKeeper::get_log("info").set_target(&std::cout);
    int size;
    MPI_Comm_size ( MPI_COMM_WORLD, &size );
    LOG_INFO ( "MPI Processes", size );
    // Compute MPI subgroup for sequential domain treatment
    MPI_Comm comm_space;
    MPI_Comm comm_uq;

    MPI_Group orig_group, new_group_space, new_group_uq;
    MPI_Comm_group ( MPI_COMM_WORLD, &orig_group );

    int num_processors = size;
    int num_partitions_uq = num_processors / num_partitions_space;

    int my_rank;
    MPI_Comm_rank ( MPI_COMM_WORLD, &my_rank );

    int* ranks = new int[num_partitions_space];
    for ( int i = 0; i < num_partitions_space; ++i )
        ranks[i] = my_rank % num_partitions_uq + i * num_partitions_uq↩
            ;

    MPI_Group_incl ( orig_group, num_partitions_space, ranks, &↩
        new_group_space );
    MPI_Comm_create ( MPI_COMM_WORLD, new_group_space, &comm_space );

    delete[] ranks;
    ranks = new int[num_partitions_uq];
    for ( int i = 0; i < num_partitions_uq; ++i )
        ranks[i] = my_rank - my_rank % num_partitions_uq + i;

    MPI_Group_incl ( orig_group, num_partitions_uq, ranks, &↩
        new_group_uq );
    MPI_Comm_create ( MPI_COMM_WORLD, new_group_uq, &comm_uq );
    delete[] ranks;

    // Run application
    PoissonMPI application ( config, comm_space, comm_uq );
    application.run ( );

    // flush log here to avoid problems
    LogKeeper::get_log ( "info" ).flush ( );

    // Stop timer.
    main_timer.stop ( );
    LOG_INFO ( "Total program run time [without MPI Init and Finalize]↩
        ",
               main_timer );

    MPI_Finalize ( );
    return 0;
}
```

3.4 Member Functions

Following member functions are components of the poisson uncertainty tutorial:

- run()
- setup_mesh()
- setup_application()
- setup_space()
- setup_system()

- setup_la_structure()
- setup_linear_solver()
- setup_bc()
- setup_linear_system()
- compute_matrix()
- compute_residual()
- visualize_solution()

3.4.1 run()

The member function `run()` calls the functions `compute_matrix()`, `compute_residual()`, `setup_linear_solver()`, `solve_linear_system()` and `visualize_solution()` to solve the stochastic Poisson problem which is described in section 2.

```
void PoissonMPI::run ( )
{
    compute_matrix ( );
    compute_residual ( );

    setup_linear_solver ( );

    Timer timer;
    timer.start ( );
    solve_linear_system ( );
    timer.stop ( );
    if ( pctensor_.MyRank ( ) == 0 )
        std::cout << "Solving linear system = " << timer.get_duration ↩
            ( ) << "\n";

    std::string visu_folder = ( *config_ )["Output"]["VisuFolder"].get↩
        <std::string>( );

    for ( int mode = 0; mode < pctensor_.SizeLocal ( ); ++mode )
    {
        std::stringstream input;
        input << visu_folder;
        int glo_mode = pctensor_.L2G ( mode );
        input << "uq_poisson_mode_" << glo_mode;
        visualize_solution ( *galerkin_sol_->Mode ( mode ), input.str ↩
            ( ) );
    }
}
```

3.4.2 setup_mesh()

The member function `setup_mesh()` reads in the mesh (`unit_square.int`)

```
void PoissonMPI::setup_mesh ( )
{
    assert ( DIM == 2 || DIM == 3 );
    LOG_INFO ( "Problem dimension", DIM );
    ( *config_ )["Mesh"]["RefinementLevel"].read<int>( ↩
        refinement_level_ );
    LOG_INFO ( "Refinement level", refinement_level_ );

    if ( rank_ == MASTER_RANK )
    {
        // read mesh (sequential, dimension DIM
        std::string mesh_filename = std::string ( DATADIR ) + ( *↩
            config_ )["Mesh"]["Filename"].get<std::string>( );
        master_mesh_ = read_mesh_from_file ( mesh_filename, DIM, DIM, ↩
            0 );

        for ( int r = 0; r < refinement_level_; ++r )
        {
            master_mesh_ = master_mesh_->refine ( );
        }
    }

    MeshPtr local_mesh = partition_and_distribute ( master_mesh_, ↩
        MASTER_RANK, comm_space_ );
    assert ( local_mesh != 0 );
    SharedVertexTable shared_verts;
    mesh_ = compute_ghost_cells ( *local_mesh, comm_space_, ↩
        shared_verts );

    std::ostringstream rank_str;
    rank_str << rank_;
    PVtkWriter writer ( comm_space_ );
    std::string output_file = ( *config_ )["Output"]["MeshFilename"].↩
        get<std::string>( );
    writer.write ( output_file.c_str ( ), *mesh_ );
}
```

3.4.3 setup_application()

The member function `setup_application()` sets the polynomial chaos information and computes the tensor structure.

```
void PoissonMPI::setup_application ( )
{
    // setup stochastic parameters
    ( *config_ )["PolynomialChaos"]["No"].read<int>( No_ );
    LOG_INFO ( "Polynomial Chaos order", No_ );
    ( *config_ )["PolynomialChaos"]["N"].read<int>( N_ );
    LOG_INFO ( "Dimension of random space", N_ );
    // Polynomial degree of random input -> here: linear random input
    No_input_ = 1;
    LOG_INFO ( "Polynomial degree of random input", No_input_ );
    std::vector<PCBasis::Distribution> dist ( N_, PCBasis::UNIFORM );
    LOG_INFO ( "Probability distribution", "UNIFORM" );

    pcbasis_.Init ( N_, No_, dist );
    pctensor_.SetMPIComm ( comm_uq_ );
    pctensor_.ComputeTensor ( No_input_, pcbasis_ );

    double nu0, sigma, q;
    ( *config_ )["Application"]["MeanViscosity"].read<double>( nu0 );
    ( *config_ )["Application"]["Variability"].read<double>( sigma );
    ( *config_ )["Application"]["Decay"].read<double>( q );

    nu_.resize ( N_ + 1, 0.0 );
    nu_[0] = nu0;
    for ( int i = 1; i < N_ + 1; ++i )
```

```
        nu_[i] = sigma * nu0 * pow ( q, i - 1.0 );
}
```

3.4.4 setup_space()

The member function `setup_space()` sets vector space.

```
void PoissonMPI::setup_space ( )
{
    // setup space Q1 elements
    space_ = new VectorSpace<double>( comm_space_ );
    int p_degree = 0;
    ( *config_ )["PolynomialChaos"]["q"].read<int>( p_degree );
    std::vector<int> degrees ( 1, p_degree );
    space_->Init ( degrees, *mesh_ );

    std::cout << "Rank: " << pctensor_.MyRank ( ) << " using NDofs = "↩
         <<
            space_->dof ( ).get_nb_dofs ( )*( pctensor_.SizeLocal ( ) ↩
                + pctensor_.SizeOffModes ( ) ) << "\n";

    if ( pctensor_.MyRank ( ) == 0 )
        std::cout << "Number of Total Dofs: " << space_->dof ( ).↩
            get_nb_dofs ( ) << "x"
        << pctensor_.Size ( ) << " = " << ( long ) space_->dof ( ).↩
            get_nb_dofs ( ) * pctensor_.Size ( ) << "\n";
}
```

3.4.5 setup_system()

The member function `setup_system()` sets the computing platform.

```
void PoissonMPI::setup_system ( )
{
    la_sys_.Platform = CPU;
    la_sys_.rank = 0;
    init_platform ( la_sys_ );
    matrix_impl_ = NAIVE;
    vector_impl_ = NAIVE;
    la_matrix_format_ = CSR;
    matrix_precond_ = NOPRECOND;
    la_sys_.GPU_CUBLAS = false;
}
```

3.4.6 setup_la_structure()

The member function `setup_la_structure()` sets the linear algebra structure.

```
void PoissonMPI::setup_la_structures ( )
{
    // Initialize linear algebra structures
    couplings_.Clear ( );
    couplings_.Init ( comm_space_, space_->dof ( ) );

    // compute matrix graph to build matrix structure
    std::vector<int> diagonal_rows, diagonal_cols, off_diagonal_rows, ↩
        off_diagonal_cols;
    std::vector < std::vector<bool> > var_coupling ( 1 );
    var_coupling[0].resize ( 1, true );
    InitStructure ( *space_, &diagonal_rows, &diagonal_cols,
                    &off_diagonal_rows, &off_diagonal_cols, &↩
                        var_coupling );

    couplings_.InitializeCouplings ( off_diagonal_rows, ↩
        off_diagonal_cols );

    // Initialize matrices and vectors
    matrix_.Init ( comm_space_, couplings_, la_sys_.Platform,
                   matrix_impl_, la_matrix_format_ );

    matrix_.InitStructure ( vec2ptr ( diagonal_rows ),
                            vec2ptr ( diagonal_cols ),
                            diagonal_rows.size ( ),
                            vec2ptr ( off_diagonal_rows ),
                            vec2ptr ( off_diagonal_cols ),
                            off_diagonal_rows.size ( ) );

    sol_.Init ( comm_space_, couplings_, la_sys_.Platform, ↩
        vector_impl_ );
    sol_.InitStructure ( );
    sol_.Zeros ( );

    galerkin_matrix_.resize ( pcbasis_.N ( ) + 1 );
    for ( int i = 0; i < pcbasis_.N ( ) + 1; ++i )
        galerkin_matrix_[i] = new CoupledMatrix<double>;

    galerkin_sol_tmp_.resize ( pctensor_.SizeLocal ( ) + pctensor_.↩
        SizeOffModes ( ) );

    for ( int i = 0; i<static_cast < int > ( galerkin_sol_tmp_.size ( ↩
        ) ); ++i )
    {
        galerkin_sol_tmp_[i] = new CoupledVector<double>;
        galerkin_sol_tmp_[i]->CloneFrom ( sol_ );
    }

    galerkin_sol_ = new PCGalerkinVector<double>;
    galerkin_res_ = new PCGalerkinVector<double>;
    galerkin_cor_ = new PCGalerkinVector<double>;

    galerkin_sol_->SetMPIComm ( comm_uq_ );
    galerkin_res_->SetMPIComm ( comm_uq_ );
    galerkin_cor_->SetMPIComm ( comm_uq_ );

    galerkin_sol_->SetModes ( galerkin_sol_tmp_, pctensor_.SizeLocal (↩
         ), pctensor_.SizeOffModes ( ) );
    galerkin_sol_->Zeros ( );
    galerkin_res_->CloneFrom ( *galerkin_sol_ );
    galerkin_cor_->CloneFrom ( *galerkin_sol_ );

    for ( int mode = 0; mode<static_cast < int > ( galerkin_sol_tmp_.↩
        size ( ) ); ++mode )
    {
        delete galerkin_sol_tmp_[mode];
    }

    system_matrix_ = new PCGalerkinMatrix<double>( sol_ );
}
```

3.4.7 setup_linear_solver()

The member function `setup_linear_solver()` sets the information about linear solver and preconditioner, the different solvers and preconditioners could be chosen via xml-file, see section 3.1.

```
void PoissonMPI::setup_linear_solver ( )
{
    std::string solver = ( *config_ )["GalerkinLinearSolver"]["↩
        LinearSolver"].get<std::string>( );
    if ( solver == "CG" )
    {
        cg_solver_.InitControl ( 1000, 1.e-12, 1.e-12, 1.e6 );
        cg_solver_.SetupOperator ( *system_matrix_ );
        linear_solver_ = &cg_solver_;
    }
    else if ( solver == "Multilevel" )
    {
        linear_solver_ = new PCMultilevelSolver<↩
            LADescriptorPolynomialChaosD>( *config_ );
        linear_solver_->InitControl ( 100, 1.e-12, 1.e-12, 1.e6 );
        linear_solver_->SetupOperator ( *system_matrix_ );
        linear_solver_->Build ( );
    }
    else
    {
        std::cout << "ERROR: No solver defined!\n";
        interminable_assert ( 0 );
    }

    std::string preconditioner = ( *config_ )["GalerkinLinearSolver"][↩
        "Preconditioner"].get<std::string>( );
    if ( solver == "CG" && preconditioner == "Mean" )
    {
#ifdef WITH_UMFPACK
        mean_precond_ = new MeanbasedPreconditioner<↩
            LADescriptorPolynomialChaosD>;
        mean_precond_->SetupOperator ( *system_matrix_ );
        cg_solver_.InitParameter ( "Preconditioning" );
        cg_solver_.SetupPreconditioner ( *mean_precond_ );
#else
        interminable_assert ( 0 );
#endif
    }
    else if ( solver == "CG" && preconditioner == "ML" )
    {
        ml_precond_ = new PCMultilevelSolver<↩
            LADescriptorPolynomialChaosD>( *config_ );
        ml_precond_->SetupOperator ( *system_matrix_ );
        ml_precond_->Build ( );

        cg_solver_.InitParameter ( "Preconditioning" );
        cg_solver_.SetupPreconditioner ( *ml_precond_ );
    }
}
```

3.4.8 setup_bc()

The member function `setup_bc()` sets up the Dirichlet boundary values.

```
void PoissonMPI::setup_bc ( )
```

```
{
    dirichlet_dofs_.clear ( );
    dirichlet_values_.clear ( );

    if ( DIM == 2 )
    {
        PoissonMPIDirichletBC2d bc;
        compute_dirichlet_dofs_and_values ( bc, *space_, 0,
                                            dirichlet_dofs_, ↩
                                                dirichlet_values_ );

    }
    else
    {
        PoissonMPIDirichletBC3d bc;
        compute_dirichlet_dofs_and_values ( bc, *space_, 0,
                                            dirichlet_dofs_, ↩
                                                dirichlet_values_ );
    }

    if ( !dirichlet_dofs_.empty ( ) )
    {
        // correct solution with dirichlet BC
        galerkin_sol_->Mode ( 0 )->SetValues ( vec2ptr ( ↩
            dirichlet_dofs_ ), dirichlet_dofs_.size ( ), vec2ptr ( ↩
            dirichlet_values_ ) );
    }
}
```

The boundary conditions are defined in the structure PoissonMPIDirichlet.

```
struct PoissonMPIDirichletBC2d
{

    PoissonMPIDirichletBC2d ( )
    {
    }

    std::vector<double> evaluate ( const Entity& face, const std::↩
        vector<Coord>& coords_on_face ) const
    {
        std::vector<double> values;

        const int material_num = face.get_material_number ( );
        const bool inflow = ( material_num == 11 );
        if ( inflow )
        {
            values.resize ( coords_on_face.size ( ), 1.0 );
        }
        return values;
    }
};

struct PoissonMPIDirichletBC3d
{

    PoissonMPIDirichletBC3d ( )
    {
    }

    std::vector<double> evaluate ( const Entity& face, const std::↩
        vector<Coord>& coords_on_face ) const
    {
        std::vector<double> values;

        const int material_num = face.get_material_number ( );
        const bool bdy = ( material_num == 10 );
        if ( bdy )
        {
            values.resize ( coords_on_face.size ( ), 0.0 );
        }
        return values;
    }
};
```

3.4.9 solve_linear_system()

The member function `solve_linear_system()` applies the linear solver in order to obtain the solution.

```
void PoissonMPI::solve_linear_system ( )
{
    linear_solver_->Solve ( *galerkin_res_, galerkin_cor_ );
    galerkin_sol_->Axpy ( *galerkin_cor_, 1.0 );
}
```

3.4.10 compute_matrix()

The member function `compute_matrix()` computes the matrix for stochastic Poisson equation.

```
void PoissonMPI::compute_matrix ( )
{
    StandardGlobalAssembler<double> global_asm;

    for ( int mode = 0; mode < pcbasis_.N ( ) + 1; ++mode )
    {
        local_mode_asm_ = new PoissonMPIModeAssembler ( mode, nu_[mode↩
            ] );
        global_asm.assemble_matrix ( *space_, *local_mode_asm_, ↩
            matrix_ );

        // correct BC -- set Dirichlet rows
        if ( !dirichlet_dofs_.empty ( ) )
        {
            if ( mode == 0 )
            {
                matrix_.diagonalize_rows ( vec2ptr ( dirichlet_dofs_ )↩
                    , dirichlet_dofs_.size ( ), 1. );
            }
            else
            {
                matrix_.diagonalize_rows ( vec2ptr ( dirichlet_dofs_ )↩
                    , dirichlet_dofs_.size ( ), 0. );
            }
        }

        galerkin_matrix_[mode]->CloneFrom ( matrix_ );
        delete local_mode_asm_;
    }

    system_matrix_->SetNumThreads ( 1 );
    system_matrix_->SetMatrix ( galerkin_matrix_ );
    system_matrix_->SetTensor ( &pctensor_ );
}
```

The operator for assembling of the matrix is implemented in the class PoissonMPIModeAssembler.

```
void PoissonMPIModeAssembler::operator() ( const Element<double>& ↩
     element, const Quadrature<double>& quadrature, LocalMatrix& lm )
{
     AssemblyAssistant<DIM, double>::initialize_for_element ( element, ↩
          quadrature );
     const int num_q = num_quadrature_points ( );
     const int total_dofs = num_dofs_total ( );
     lm.Resize ( total_dofs, total_dofs );
     lm.Zeros ( );

     if ( mode_ == 0 )
     {
          for ( int q = 0; q < num_q; ++q )
          {
               const double wq = w ( q );
               const double dJ = std::fabs ( detJ ( q ) );
               // assemble a1(u,v) = \int \alpha_1 * {\grad(u) : \grad(v)↩
                    }
               for ( int i = 0; i < num_dofs ( 0 ); ++i )
               {
                    for ( int j = 0; j < num_dofs ( 0 ); ++j )
                    {
                         lm ( dof_index ( i, 0 ), dof_index ( j, 0 ) ) +=
                                   wq * nu_ * dJ * dot ( grad_phi ( j, q, 0 )↩
                                        , grad_phi ( i, q, 0 ) );
                    }
               }
          }
     }
     else
     {
          double pi = acos ( 0.0 )*2.0;
          for ( int q = 0; q < num_q; ++q )
          {
               const double wq = w ( q );
               const double dJ = std::fabs ( detJ ( q ) );
               // assemble a1(u,v) = \int \alpha_1 * {\grad(u) : \grad(v)↩
                    }
               for ( int i = 0; i < num_dofs ( 0 ); ++i )
               {
                    for ( int j = 0; j < num_dofs ( 0 ); ++j )
                    {
                         lm ( dof_index ( i, 0 ), dof_index ( j, 0 ) ) +=
                                   wq * nu_ * sin ( 2.0 * pi * x ( q )[0] * ↩
                                        mode_ ) * sin ( 2.0 * pi * x ( q )[1] ↩
                                        * mode_ ) * dJ * dot ( grad_phi ( j, q↩
                                        , 0 ), grad_phi ( i, q, 0 ) );
                    }
               }
          }
     }
}
```

3.4.11 compute_residual()

The member function `compute_residual()` computes the residual for stochastic Poisson equation.

```
void PoissonMPI::compute_residual ( )
```

```
{
    system_matrix_->VectorMult ( *galerkin_sol_, galerkin_res_ );
    galerkin_res_->Scale ( -1.0 );

    if ( pctensor_.MyRank ( ) == 0 )
    {
        StandardGlobalAssembler<double> global_asm;

        CoupledVector<double> mean_rhs;
        mean_rhs.CloneFrom ( sol_ );
        mean_rhs.Zeros ( );

        local_mode_asm_ = new PoissonMPIModeAssembler ( 0, nu_[0] );
        // Assemble mean rhs
        global_asm.assemble_vector ( *space_, *local_mode_asm_, ↩
            mean_rhs );
        delete local_mode_asm_;

        galerkin_res_->ModeAxpy ( 0, mean_rhs, 1.0 );
    }

    if ( !dirichlet_dofs_.empty ( ) )
    {
        std::vector<LAD::DataType> zeros ( dirichlet_dofs_.size ( ), ↩
            0. );
        for ( int mode = 0; mode < pctensor_.SizeLocal ( ); ++mode )
        {
            galerkin_res_->Mode ( mode )->SetValues ( vec2ptr ( ↩
                dirichlet_dofs_ ), dirichlet_dofs_.size ( ), vec2ptr (↩
                 zeros ) );
        }
    }
}
```

The operator for assembling of the residual is implemented in the class PoissonMPIModeAssembler.

```
void PoissonMPIModeAssembler::operator() ( const Element<double>& ↩
    element, const Quadrature<double>& quadrature, LocalVector& lv )
{
    AssemblyAssistant<DIM, double >::initialize_for_element ( element, ↩
        quadrature );
    const int num_q = num_quadrature_points ( );
    const int total_dofs = num_dofs_total ( );
    lv.clear ( );
    lv.resize ( total_dofs, 0. );

    for ( int q = 0; q < num_q; ++q )
    {
        const double wq = w ( q );
        const double dJ = std::fabs ( detJ ( q ) );
        // assemble a1(u,v) = \int  * {\grad(u) : \grad(v)}
        for ( int i = 0; i < num_dofs ( 0 ); ++i )
        {
            lv[dof_index ( i, 0 )] +=
                    wq * dJ * phi ( i, q, 0 );
        }
    }
}
```

3.4.12 visualization_solution()

The member function `visualization_solution()` writes data for visualization of the solution.

```
void PoissonMPI::visualize_solution ( LAD::VectorType& u, std::string ↩
    const& filename ) const
{
    u.UpdateCouplings ( );
    int num_intervals = 2;
    ParallelCellVisualization<double> visu ( *( space_ ), ↩
        num_intervals, comm_space_, 0 );

    std::vector<double> remote_index ( mesh_->num_entities ( mesh_->↩
        tdim ( ) ), 0 );
    std::vector<double> sub_domain ( mesh_->num_entities ( mesh_->tdim↩
         ( ) ), 0 );

    for ( mesh::EntityIterator it = mesh_->begin ( mesh_->tdim ( ) );
          it != mesh_->end ( mesh_->tdim ( ) );
          ++it )
    {
        int temp1, temp2;
        mesh_->get_attribute_value ( "_remote_index_", mesh_->tdim ( )↩
            ,
                                     it->index ( ),
                                     &temp1 );
        mesh_->get_attribute_value ( "_sub_domain_", mesh_->tdim ( ),
                                     it->index ( ),
                                     &temp2 );
        remote_index.at ( it->index ( ) ) = temp1;
        sub_domain.at ( it->index ( ) ) = temp2;
    }

    std::stringstream input;
    input << filename;

    if ( num_partitions_ > 1 )
        input << ".pvtu";
    else
        input << ".vtu";

    visu.visualize ( EvalFeFunction<LAD>( *( space_ ), u ), "val" );

    visu.visualize_cell_data ( remote_index, "_remote_index_" );
    visu.visualize_cell_data ( sub_domain, "_sub_domain_" );
    visu.write ( input.str ( ) );
}
```

4 Program Output

HiFlow^3 can be executed in a parallel or sequential mode which influence the generated output data. Note that the log files can be viewed by any editor.

4.1 Sequential Mode

Executing the program sequentially by typing **./poisson_uncertainty poisson_uncertainty.xml** generates following output data:

- Mesh/Geometry data:
  - **uq_poisson_mesh.pvtu** Global mesh
  - **uq_poisson_mesh_0.vtu** Global mesh owned by process 0 containing the mesh information

- Solution data:

  Since it is only possible to visualize data of polynomial degree 1 (Q1 on quads in 2 dimensions), only the data corresponding to the degrees of freedom of a Q1-element are written out. It means the information of the degrees of freedom of higher order are lost due to the fact that this information cannot be visualized using the vtk-format.

  - **uq_poisson_mode_X_0.vtu** Solution of Poisson problem (vtk-format) owned by PC mode X for X = 0, 1,..., 19.

- Log files:
  - **uq_poisson_mpi.info** is a list of parameters and some helpful informations to control the program, for example information about the residual of the linear and non-linear solver.

4.2 Parallel Mode

Executing the program in parallel, for example : by typing **mpirun -n 8 ./poisson_uncertainty poisson_uncertainty.xml 4**, it executes using 8 processors for computation in total with domain decomposition for 4 (the total number of processor is divisible by the number of subdomain), and it generates following output data:

- Mesh/Geometry data:
  - **uq_poisson_mesh.pvtu** Global mesh
  - **uq_poisson_mesh_X.vtu** Local mesh owned by process X for X=0, 1, 2 and 3 (vtk-format).

- Solution data:

  Since it is only possible to visualize data of polynomial degree 1 (Q1 on quads in 2 dimensions), only the data corresponding to the degrees of freedom of a Q1-element are written out. It means the information of the degrees of freedom of higher order are lost due to the fact that this information cannot be visualized using the vtk-format.

  - **uq_poisson_mode_X1_X2.vtu** Solution of the Poisson problem (vtk-format) owned by process X2 for X2 = 0, 1, 2 and 3. X1 denotes the Polynomical Chaos modes, and X1 = 0, 1,..., 19.

- Log files:

  - **uq_poisson_mpi.info** is a list of parameters and some helpful informations to control the program, for example information about the residual of the linear and non-linear solver.

4.3 Visualization of the Solution

HiFlow^3 only generates output data, see section 3.4.12, but does not visualize. The mesh/geometry data as well as the solution data can be visualized by any external program which can handle the `vtk` data format as e.g. the program ParaView[3].

5 Numerical Example

5.1 Two-Dimensional Example

As described in section 2, in this example, we analyse in a unit-square domain with 3 uncertain parameters, i.e. $\boldsymbol{\xi} := (\xi_1, \xi_2, \xi_3)$, and truncating the polynomial degree till 3, namely $p = 3$. Therefore, this example contains 20 Polynomial Chaos modes ($P = 19$) which could be obtained by applying (12).

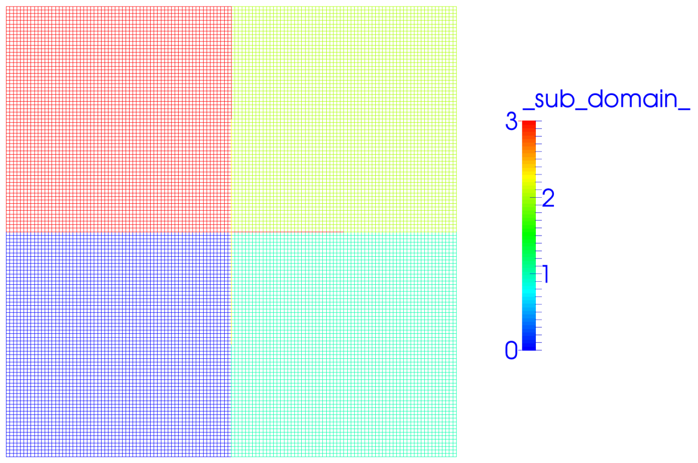
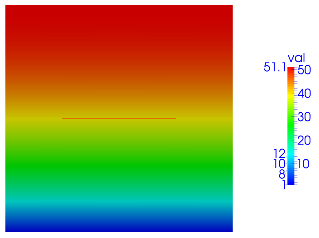
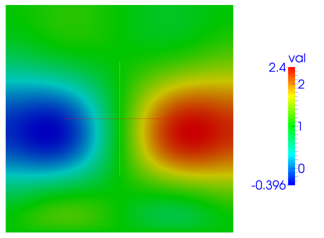


Figure 1: Refined 2-dimensional mesh used for the example.

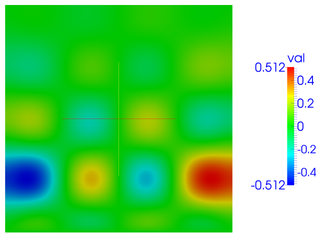
The Figure 1 shows the discrete mesh and how it is distributed over four processors, and the Figure 2 presents selected results of the first 4 modes (over 20). Moreover, the result of mode 0 corresponds to the mean value of the stochastic poisson system.



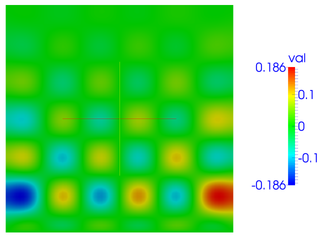
(a) PC mode 0



(b) PC mode 1



(c) PC mode 2



(d) PC mode 3

Figure 2: The first 4 Polynomial Chaos (PC) mode results.

References

[1] The mesage passing interface (mpi) standard. http://www.mcs.anl.gov/research/projects/mpi.

[2] O.P.L. Maître and O.M. Knio. *Spectral Methods for Uncertainty Quantification: With Applications to Computational Fluid Dynamics*. Scientific Computation. Springer, 2010.

[3] A. H. Squillacote. *The ParaView guide: a parallel visualization application*. Kitware [Clifton Park, NY], 2007.

[4] Norbert Wiener. The homogeneous chaos. *American Journal of Mathematics*, 60(4):897–936, October 1938.

[5] Dongbin Xiu and George Em Karniadakis. The wiener–askey polynomial chaos for stochastic differential equations. *SIAM J. Sci. Comput.*, 24(2):619–644, February 2002.