Adaptive Finite Volume Toolbox Documentation

Release 0.1 alpha

SeHyoun Ahn

Feb 27, 2019

CONTENTS

This is a set of codes to help with using the adaptive finite volume method for solving Fokker-Planck equations in economic applications

$$\frac{dg}{dt} = -\sum_{i=1}^{d} \frac{d}{dx_i}(s_i(x) \cdot g(x)) + \sum_{i=1}^{d} \nu_i \frac{d^2 g(x)}{dx_i^2}$$

For an introduction to the finite-volume method, one can read any reference (There is one I have written for economists *[Ahn, 2019]*). However, working through the *Tutorials* should be sufficient for most applications.

The **advantages** of (adaptive) finite volume method are:

- Conservation of mass (of discretized equations)
- Positivity of the distribution function
- Local interaction of parameters (compared to global approximations such as Chebyshev polynomials)
  - More interpretable results from geometric nature of parameters
  - Adjusting approximation domain in a natural manner
  - Since parameters are local/interpretable in nature, a natural candidate for the function approximation under perturbation methods.

The **disadvantages** are:

- Grid points increase exponentially
  - adaptive refinement helps address this issue
- Keeping the structure of completely unstructured grid is costly.

Ultimately, the result of the cost-benefit analysis for a particular problem is hard to estimate without testing. The codes/toolbox should shorten the testing time for the finite-volume method for Fokker-Planck equations.

CHAPTER

ONE

GENERAL WORKFLOW

- Initialize Grid

```
grid = afv_grid;
```

- Set the dimensionality of the problem:

```
grid.set_dim(n);
```

- Split the grid for the first time:

```
grid.split_init(1, x_cuts);
```

- Collect Edges:

```
grid.extract_edges();
```

- Set drifts and variances:

```
grid.drift = stuff;
grid.diffusion = stuff;
```

- Compute the Transition Matrix:

```
A_FP = grid.compute_transition_matrix_modified();
A_FP_boundary = grid.compute_transition_matrix_boundary(ind, dir, flow,  is_left_
↪edge);
A_FP = A_FP + A_FP_boundary;
```

- Further refine grids:

```
grid.split(indices);
```

CHAPTER

TWO

CONTENTS

2.1 Installation

1. Download the files from github
2. Compile the mex files by running `compile_mex_files` included in the folder
    - This step might require setting up a MATLAB compatible C++ compiler. See MATLAB's documentation page for details.
3. Run `addpath('/location/to/the/folder/src');` at the beginning of the program execution to include relevant toolbox functions.

2.2 Tutorials

2.2.1 Grid Construction and Basic Interactions

Expected Read Time: XXXXX

To implement the adaptive finite volume methods, we need to keep:

- Connectivity/Neighbor structure of the cells
- Edge between cells

and define the drift and diffusion terms for the edges. *afv_grid* class is written to help working with this computation. To start, we initialize and set the dimensionality of the problem by

```
>> grid = afv_grid(2);
```

At this state, the grid is initialized with one cell of [0, 1] x [0, 1]. We can see this by checking the number of nodes via

```
>> disp(grid.num_n);

    1
```

and checking the boundaries of the one nodes

```
>> disp(grid.n2bd);

    0    0    1    1
```

The boundaries are ordered by lower boundaries and then upper boundaries (by coordinate dimension). Hence, the output is

```
[lower boundary dim 1, lower boundary dim 2, upper boundary dim 1, upper boundary dim␣
↪2]
```

Now, one can split the node by feeding in cut points, and using *split_init*

```
>> x_knots{1} = 1/2;
>> x_knots{2} = [1/3; 2/3];

>> grid.split_init(1, x_knots);
```

As we can see, we define cell array of knot points for each dimension. With 1 cut point in dimension 1 and 2 cut points in dimension 2, we end up with total of 6 cells. We can check this by checking *num_n* again

```
>> disp(grid.num_n);

    6
```

Checking boundaries of the nodes, we can see that nodes are split correctly

```
>> disp(grid.n2bd);

         0         0    0.5000    0.3333
    0.5000         0    1.0000    0.3333
         0    0.3333    0.5000    0.6667
    0.5000    0.3333    1.0000    0.6667
         0    0.6667    0.5000    1.0000
    0.5000    0.6667    1.0000    1.0000
```

Part of the requirement in computing the finite-volume discretization is keeping the "neighbor" structure of the grid so that one can compute the flows between cells. Keeping this structure together is the most difficult part of the implementation, and this structure is updated within the class as long as only supported functions (*split_init*, *split*, and *add_new_nodes*) are called. We can see the connectivity of the cells are updated properly by checking *n2n*

```
>> disp(grid.n2n);

(:,:,1) =

     []      []
    [1]      []
     []     [1]
    [3]     [2]
     []     [3]
    [5]     [4]


(:,:,2) =

    [2]     [3]
     []     [4]
    [4]     [5]
     []     [6]
    [6]      []
     []      []
```

where things correspond to (point, coordinate direction, forward=2/backward=1). For example, [2] in the (1,2,2) denotes that the neighbor forward of point 1 in the first coordinate direction is point 2. This neighbor information in

n2n is used to build the transition matrix. Because these steps are automated, one never needs to (and probably should not) work with low-level representation like *n2n* directly.[#]_

We can further split the first grid by calling

```
>> x_knots{1} = 1/4;
>> x_knots{2} = 1/6;

>> grid.split(1, x_knots);
```

This is the process of adaptively updating grid points. Now, to build the the transition matrix of the corresponding finite-volume method, we need the information on the edges/interfaces between nodes. The edges will be built based on *n2n* that has been kept consistent underneath the problem. One can setup the edges by calling *extract_edges*

```
>> grid.extract_edges();
```

This function call builds all internal edge related quantities. For example, we can see that there are 13 internal edges

```
>> disp(grid.num_e);

    13
```

Following similar syntax, we can check

- boundaries: *e2bd*
- connectivity to nodes: *e2n*
- direction of the edge: *e2dir*

Again, for most applications, one would not need to work with low-level grid representation like *e2n* or *e2dir*, but instead work with the center of edges by calling

```
>> grid.compute_edge_midpoints()

ans =

    0.2500    0.0833
    0.5000    0.5000
    0.5000    0.8333
    0.5000    0.0833
    0.2500    0.2500
    0.5000    0.2500
    0.1250    0.1667
    0.7500    0.3333
    0.2500    0.6667
    0.7500    0.6667
    0.3750    0.1667
    0.1250    0.3333
    0.3750    0.3333
```

In practice, these are the points one would use to compute the drift and diffusion for the Fokker-Planck equations. In this tutorial, a detailed description of the behavior of the grid is given, but in most applications, the underlying behavior of the grid does not have to be known. One can just follow a recipe to use the finite volume method as can be seen in the next section.

2.2.2 Steady-State Distribution of Ornstein-Uhlenbeck Process

Expected Read Time: XXXXX

In this tutorial, we will solve for the distribution of resulting from the Ornstein-Uhlenbeck Process

$$\frac{\partial f}{\partial t} = -\sum_{i=1}^{2} \theta_i \frac{\partial}{\partial x_i} \left[(\mu_i - x_i) \cdot f \right] + \sum_{i=1}^{2} \frac{\sigma_i^2}{2} \frac{\partial^2 f}{\partial x_i^2}$$

in 2-dimension. From an explicit calculation, one can get that the steady-state is given by

$$f(\mathbf{x}) = \prod_{i=1}^{2} \left[\sqrt{\frac{1}{\pi \sigma_i^2}} e^{-\theta \frac{(x_i - \mu_i)^2}{\sigma_i^2}} \right]$$

Since the actual program itself is not long, we will carry everything. First, we define the parameters of the Ornstein-Uhlenbeck process

```
>> n_dim = 2;
>> n_points = 20;
>> int_sig = 0.1;
>> make_plots = true;

>> mu = 0.495.*ones(1, n_dim);
>> theta = 1.*ones(n_dim,1);
>> sigma = int_sig.^2.*ones(n_dim, 1);
```

Hence, we have

$$\vec{\mu} = \begin{bmatrix} 0.495 \\ 0.495 \end{bmatrix}$$
$$\vec{\sigma} = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}$$
$$\vec{\theta} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

To implement this example, we initialize the *afv_grid* object.

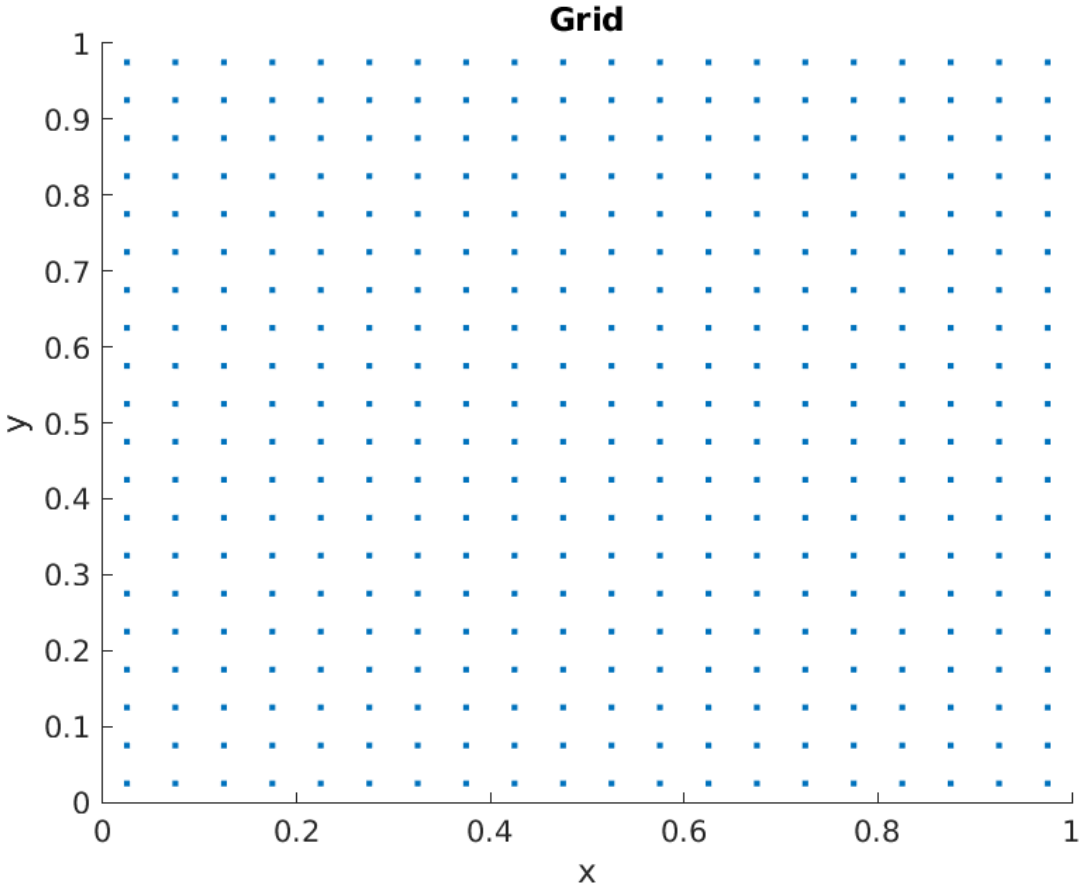
```
>> grid = afv_grid(2);
```

First, we will split the grid uniformly into 20 grid points using *split_init*. This function only allows tensor structure division of the grid (regular grid), and the splitting cut points are given as cells of vectors of dimension.

```
>> cut_points = cell(n_dim, 1);
>> cut_points_1d = linspace(0, 1, n_points+1);
>> cut_points_1d = cut_points_1d(2:end-1)';
>> for iter_dim = 1:n_dim
>>     cut_points{iter_dim} = cut_points_1d;
>> end
>> grid.split_init(1, cut_points);
```

At this points, [0,1]x[0,1] has been split into 400 grid points (20 per dimension). We can visualize this by making a scatter plot of center of the cells. One can compute the center of the cells via *node_midpoints*.

```
>> x_i = grid.node_midpoints();
>> scatter(x_i(:,1), x_i(:,2));
```

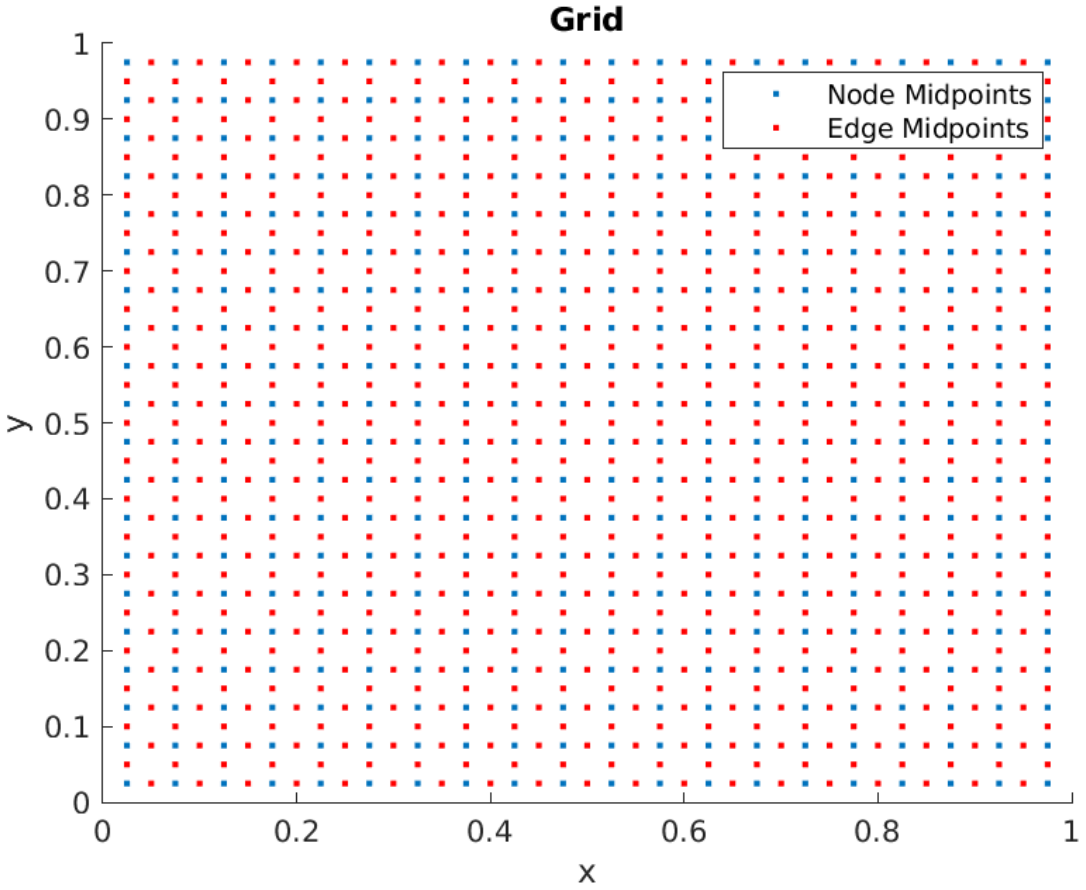


At this point, the grid only has the information on the cells. It keeps the neighbor structure of the cells updated, but not the edges. The edge information can be updated via *extract_edges*. Whenever the grid structure changes, *extract_edges* needs to be called to make sure that the edge structure is consistent with the updated grid.

```
>> grid.extract_edges();
```

We can visualize edges by making a scatter plot of their midpoints (using *edge_midpoints*) as well

```
>> hold on;
>> x_i = grid.edge_midpoints();
>> scatter(x_i(:,1), x_i(:,2), 'r.');
```



Now, we are ready to implement the Ornstein-Uhlenbeck process. The equation is completely defined with drifts and diffusion terms corresponding to the edges. The data for edges are stored as a vector. However, the drift values needs to be different depending on which direction the edge faces. The normal direction of the edge are stored as *e2dir*. Hence, the drift terms can be implemented with

```
>> x_i = grid.edge_midpoints();
>> for iter_dim = 1:n_dim
>>     cur_ind = find(grid.e2dir(1:grid.num_e) == iter_dim);
>>     grid.drift(cur_ind) = -theta(iter_dim).*(x_i(cur_ind, iter_dim) - mu(iter_
↪dim));
>> end
```

For example, with `iter_dim = 1`, the drift value of the edges facing the first coordinate directions are updated. This is why x_1 (= `x_i(cur_ind, iter_dim)`) is used to compute the drift.

Similarly diffusion can be implemented,

```
>> for iter_dim = 1:n_dim
>>     cur_ind = find(grid.e2dir(1:grid.num_e) == iter_dim);
>>     grid.diffusion(cur_ind) = sigma(iter_dim);
>> end
```

note that the value that's defined for diffusion is σ_i^2, without the factor $\frac{1}{2}$. As the fraction $\frac{1}{2}$ shows up in all equations, it is taken as implicit when the transition matrix is built.

Now, once we have set the drift and diffusion terms to the edges, we can build the transition matrix using

compute_transition_matrix_modified.

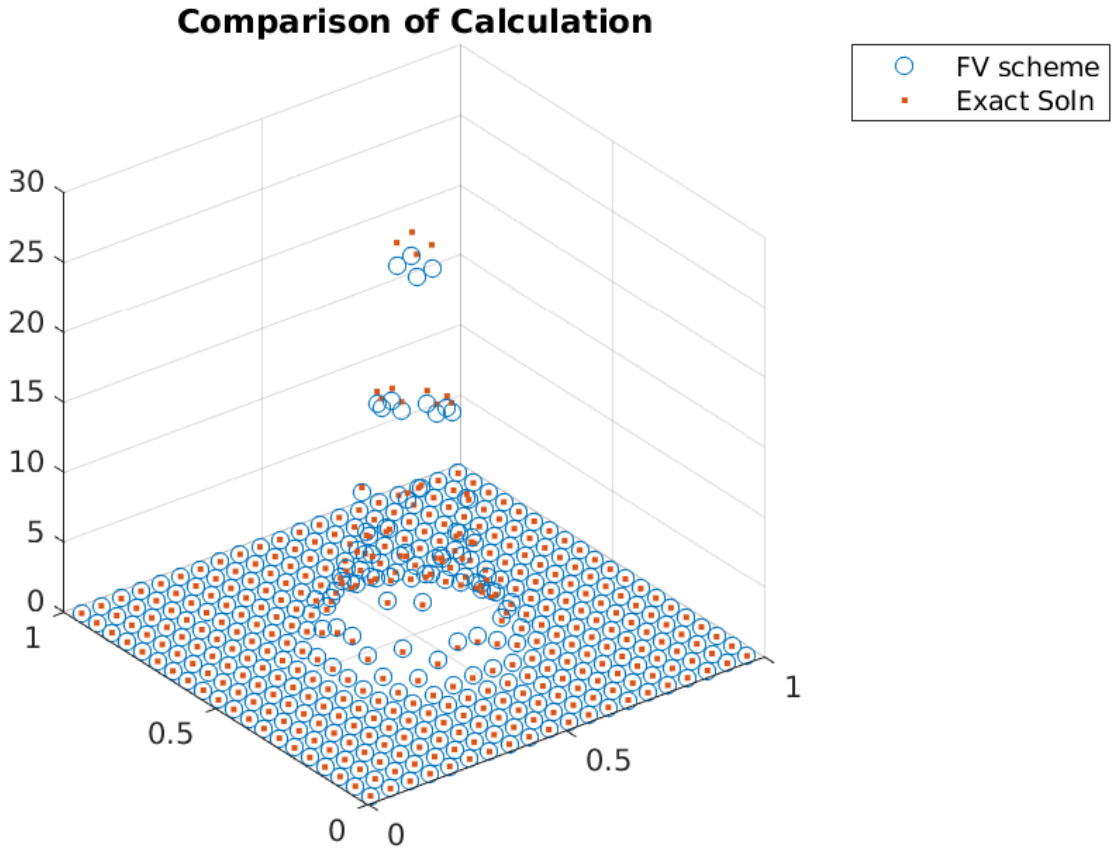
```
>> A = grid.compute_transition_matrix_modified();
```

For the steady-state, we find a solution to $Ag = 0$. Hence, we can use any eigenvalue solver to do so.

```
>> [g, ~] = eigs(A_FP, 1, 'sm');
>> g = g./sum(g);
```

Finally, we can make a plot to see the accuracy of the scheme

```
>> x_i = grid.node_midpoints();
>> scatter3(x_i(:,1),x_i(:,2),g./grid.node_weights);
>> hold on;
>> scatter3(x_i(:,1),x_i(:,2),g_true(x_i),'.');
```



For this particular set of parameters, the grid is not good enough to approximate with high accuracy. One can see that this is largely due to picking wrong domain for the parameter values. This will be fixed by adjusting domain or using adaptive refinements. The adaptive refinements will be shown in the next tutorial, but in the mean time, let's solve a problem with nicer parameters.

One nice feature of the finite volume discretization is that the structure of the transition matrix is independent of the

parameter values, so only the drift and diffusion have to be updated to reflect the new parameters.

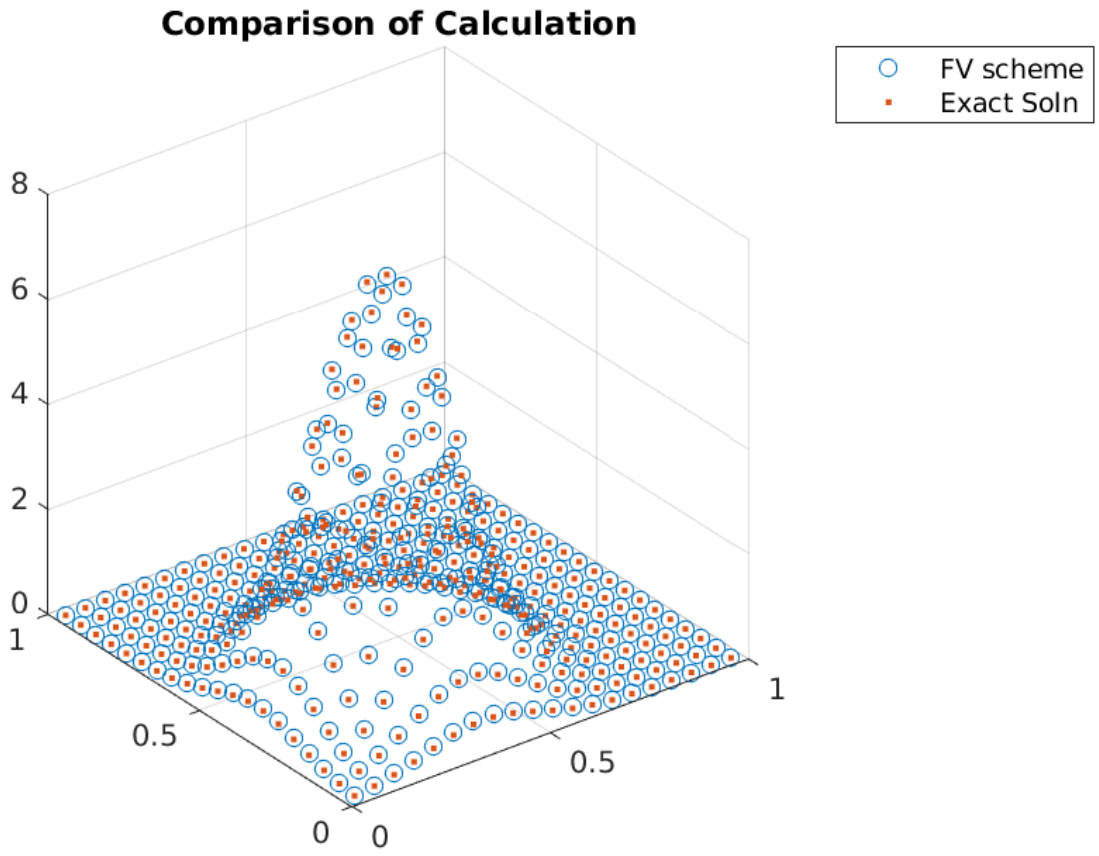
$$\vec{\mu} = \begin{bmatrix} 0.35 \\ 0.35 \end{bmatrix}$$
$$\vec{\sigma} = \begin{bmatrix} 0.2 \\ 0.2 \end{bmatrix}$$
$$\vec{\theta} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

To implement these parameters, we can just update the drifts and diffusion

```
>> int_sig = 0.2;
>> make_plots = true;

>> mu = 0.35.*ones(1, n_dim);
>> theta = 1.*ones(n_dim,1);
>> sigma = int_sig.^2.*ones(n_dim, 1);

>> x_i = grid.edge_midpoints();
>> for iter_dim = 1:n_dim
>>     cur_ind = find(grid.e2dir(1:grid.num_e) == iter_dim);
>>     grid.drift(cur_ind) = -theta(iter_dim).*(x_i(cur_ind, iter_dim) - mu(iter_
↪dim));
>>     grid.diffusion(cur_ind) = sigma(iter_dim);
>> end
```

and compute the transition matrix and the steady-state distribution

```
>> A = grid.compute_transition_matrix_modified();
>> [g, ~] = eigs(A_FP, 1, 'sm');
>> g = g./sum(g);
```



To conclude, one can see that using the finite volume discretization is simple with minimal coding. One just needs to call

1. Initialize grid *afv_grid*
2. Split the grid *split_init*
3. Extract Edges *extract_edges*
4. Define drifts and diffusion
5. Build transition matrix *compute_transition_matrix_modified*

Technical Disclaimer: In the tutorial there was no mention of the boundary conditions. Implicitly it is being assumed that the boundary is a reflecting boundary. Hence, if σ_i's are too large, the exact solution is not the exact solution to the PDE being solved by the finite-volume method. Allowing "flows" at the boundaries will be considered later (*compute_transition_matrix_boundary*.)

2.2.3 OU Process with Adaptive Refinements

Expected Read Time: XXXXX

In this tutorial, we will work through an example of applying adaptive refinements and as a starting point will take the bad parameter values from the previous tutorial. Recall that we are solving for the steady-state distribution of resulting

from the Ornstein-Uhlenbeck Process

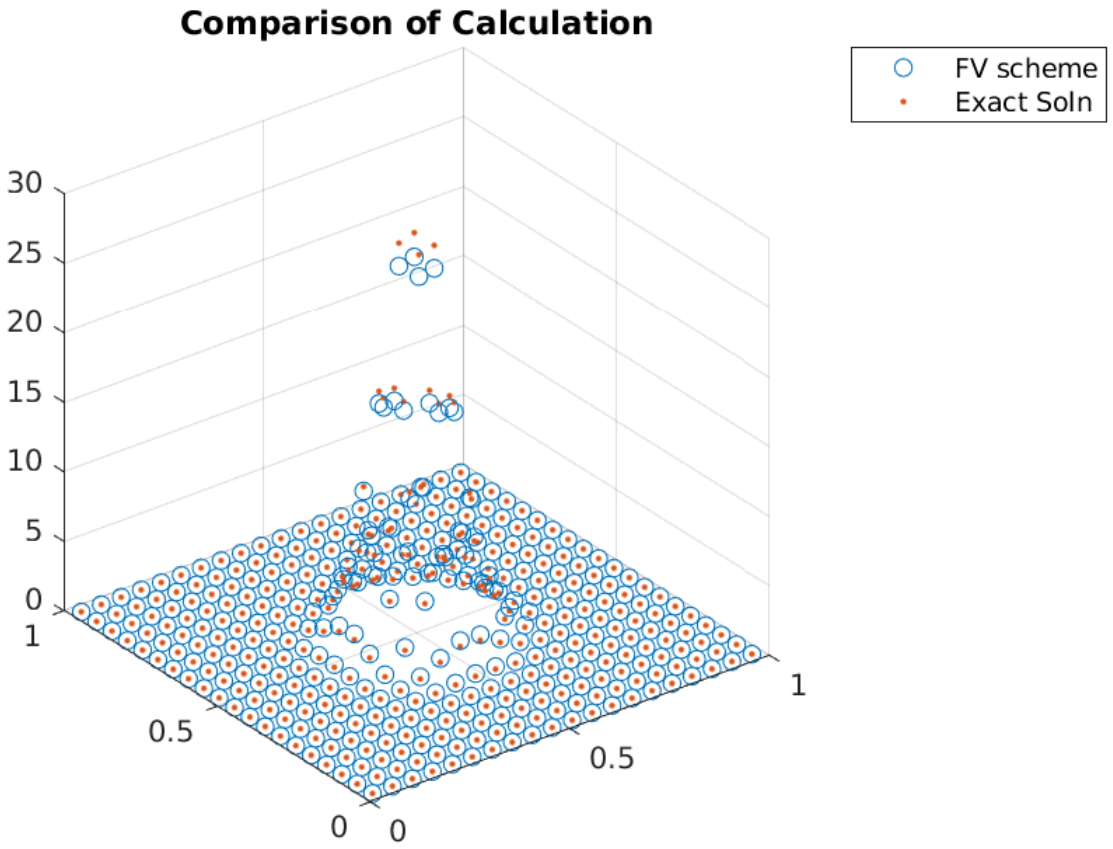
$$\frac{\partial f}{\partial t} = -\sum_{i=1}^{2} \theta_i \frac{\partial}{\partial x_i} \left[(\mu_i - x_i) \cdot f \right] + \sum_{i=1}^{2} \frac{\sigma_i^2}{2} \frac{\partial^2 f}{\partial x_i^2}$$

in 2-dimension. Since the code is short, we reproduce it here, but we have computed the distribution of corresponding to certain parameter values, and for those values, the approximation was not great.

```
>> n_dim = 2;
>> n_points = 20;
>> int_sig = 0.1;
>> make_plots = true;

>> mu = 0.495.*ones(1, n_dim);
>> theta = 1.*ones(n_dim,1);
>> sigma = int_sig.^2.*ones(n_dim, 1);

>> grid = afv_grid(2);

>> cut_points = cell(n_dim, 1);
>> cut_points_1d = linspace(0, 1, n_points+1);
>> cut_points_1d = cut_points_1d(2:end-1)';
>> for iter_dim = 1:n_dim
>>     cut_points{iter_dim} = cut_points_1d;
>> end
>> grid.split_init(1, cut_points);

>> grid.extract_edges();

>> x_i = grid.edge_midpoints();
>> for iter_dim = 1:n_dim
>>     cur_ind = find(grid.e2dir(1:grid.num_e) == iter_dim);
>>     grid.drift(cur_ind) = -theta(iter_dim).*(x_i(cur_ind, iter_dim) - mu(iter_
↪dim));
>> end

>> for iter_dim = 1:n_dim
>>     cur_ind = find(grid.e2dir(1:grid.num_e) == iter_dim);
>>     grid.diffusion(cur_ind) = sigma(iter_dim);
>> end

>> A = grid.compute_transition_matrix_modified();

>> [g, ~] = eigs(A_FP, 1, 'sm');
>> g = g./sum(g);
```

The accuracy seen from this scheme was not great.

```
>> x_i = grid.node_midpoints();
>> scatter3(x_i(:,1),x_i(:,2),g./grid.node_weights);
>> hold on;
>> scatter3(x_i(:,1),x_i(:,2),g_true(x_i),'.');
```



One should reduce the domain for this problem instead of using adaptive refinements, but in this problem, we will adaptively refine the grid by splitting the grid points where necessary. This requires one to use some sort of metric to consider the expected gain from splitting a cell. The normalized weight of a cell works well, i.e.,

$$\text{metric} = g \cdot \text{drift}$$

We can compute the normalized weight via

```
>> drift_i = -theta'.*(x_i - mu);
>> drift = max(abs(drift_i), [], 2);
>> metric = g.*drift;
```

Given the metric, one can split a cell when it is bigger than a given number. To not adjust the values in iteration, we adapt the nodes with highest metric values by

```
>> [~, ind_adapt] = sort(metric, 'descend');
>> ind_adapt = ind_adapt(1:floor(adapt_fraction*length(ind_adapt)));
```

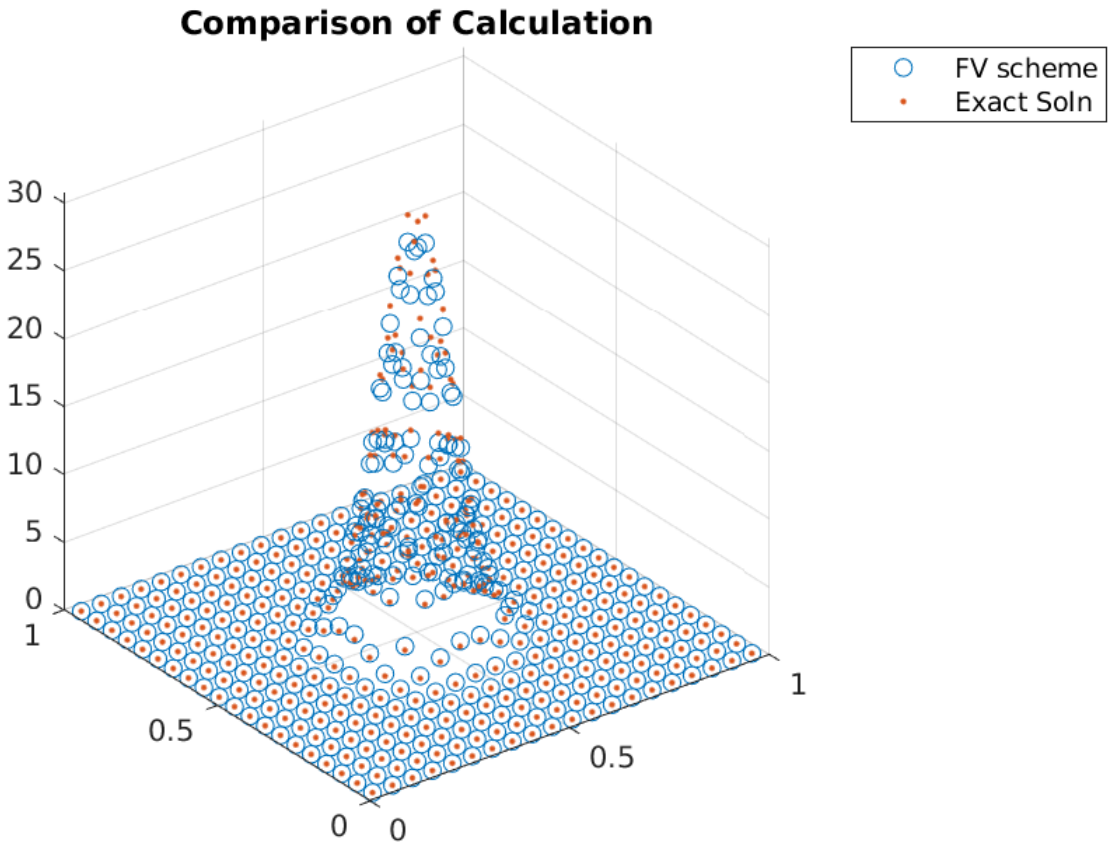
To actually split the nodes, we use the *split* function. This function requires the node to split and the cut points in each coordinate direction. Using the midpoints as cut points, we can apply the function by

```
>> grid.split(ind_adapt, mat2cell(x_i(ind_adapt,:), ones(length(ind_adapt),1), ones(n_
↪dim, 1)));
```

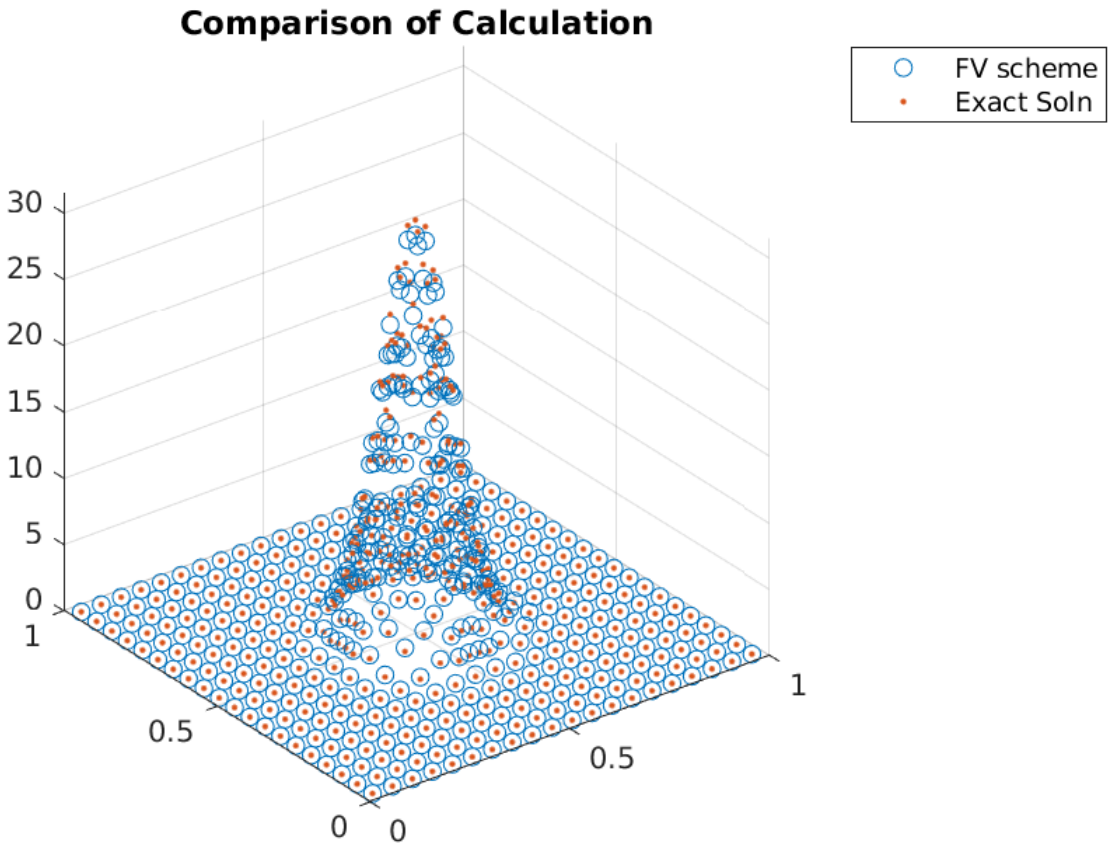
Given the updated grid, one has to update the edge information by calling *extract_edges*, but otherwise, the computation is the same as before, i.e.,

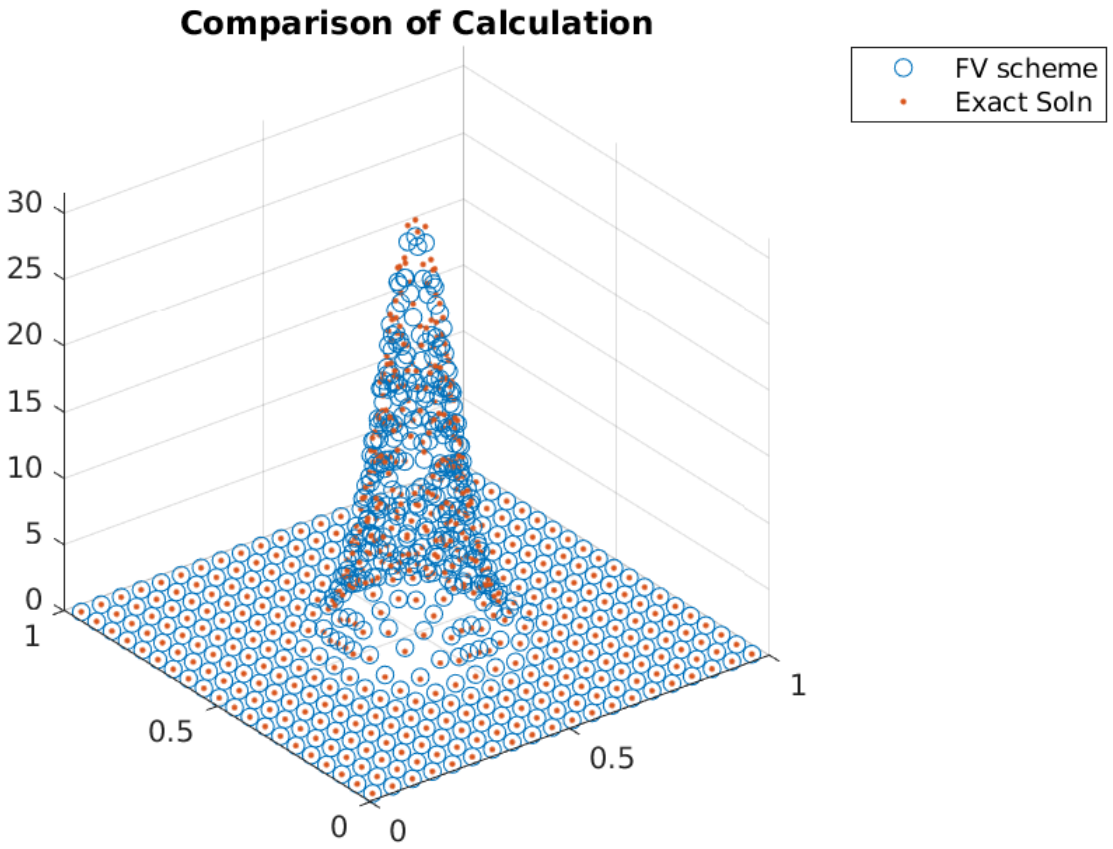
```
>> grid.extract_edges();

>> x_i = grid.edge_midpoints();
>> for iter_dim = 1:n_dim
>>     cur_ind = find(grid.e2dir(1:grid.num_e) == iter_dim);
>>     grid.drift(cur_ind) = -theta(iter_dim).*(x_i(cur_ind, iter_dim) - mu(iter_
↪dim));
>> end

>> for iter_dim = 1:n_dim
>>     cur_ind = find(grid.e2dir(1:grid.num_e) == iter_dim);
>>     grid.diffusion(cur_ind) = sigma(iter_dim);
>> end

>> A = grid.compute_transition_matrix_modified();

>> [g, ~] = eigs(A_FP, 1, 'sm');
>> g = g./sum(g);
```

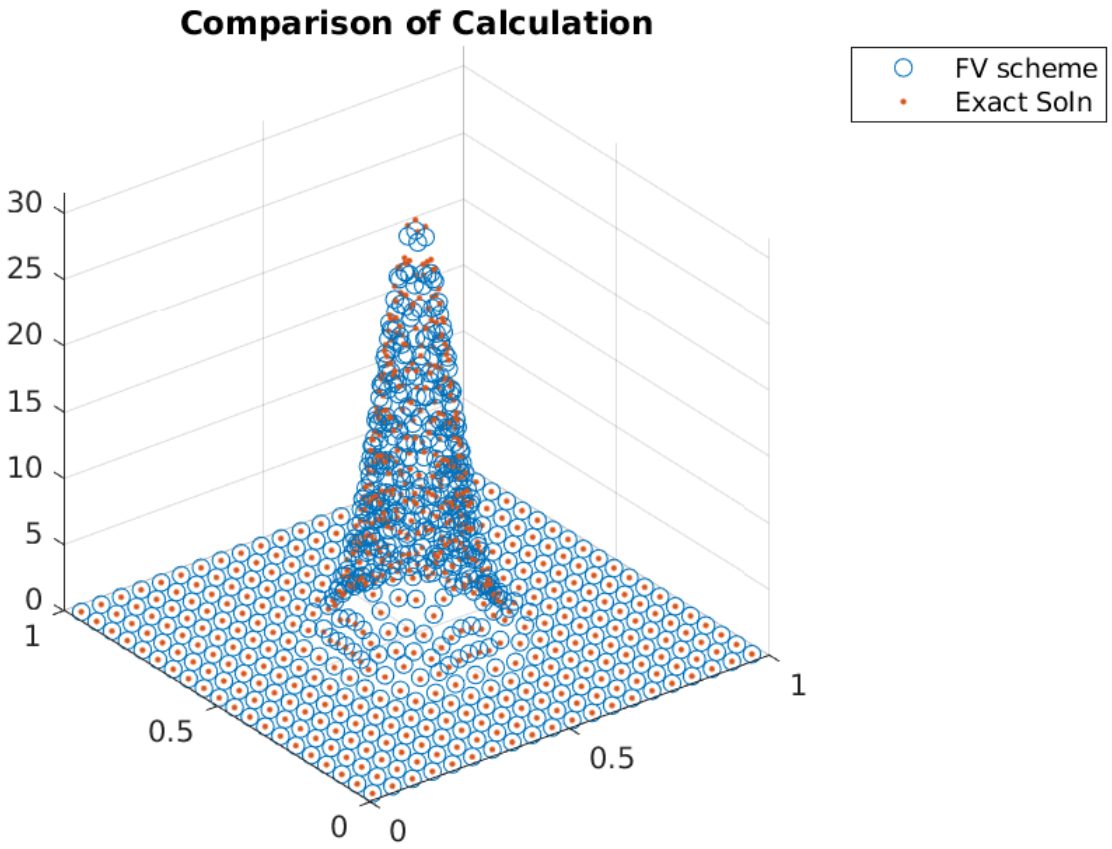
resulting in

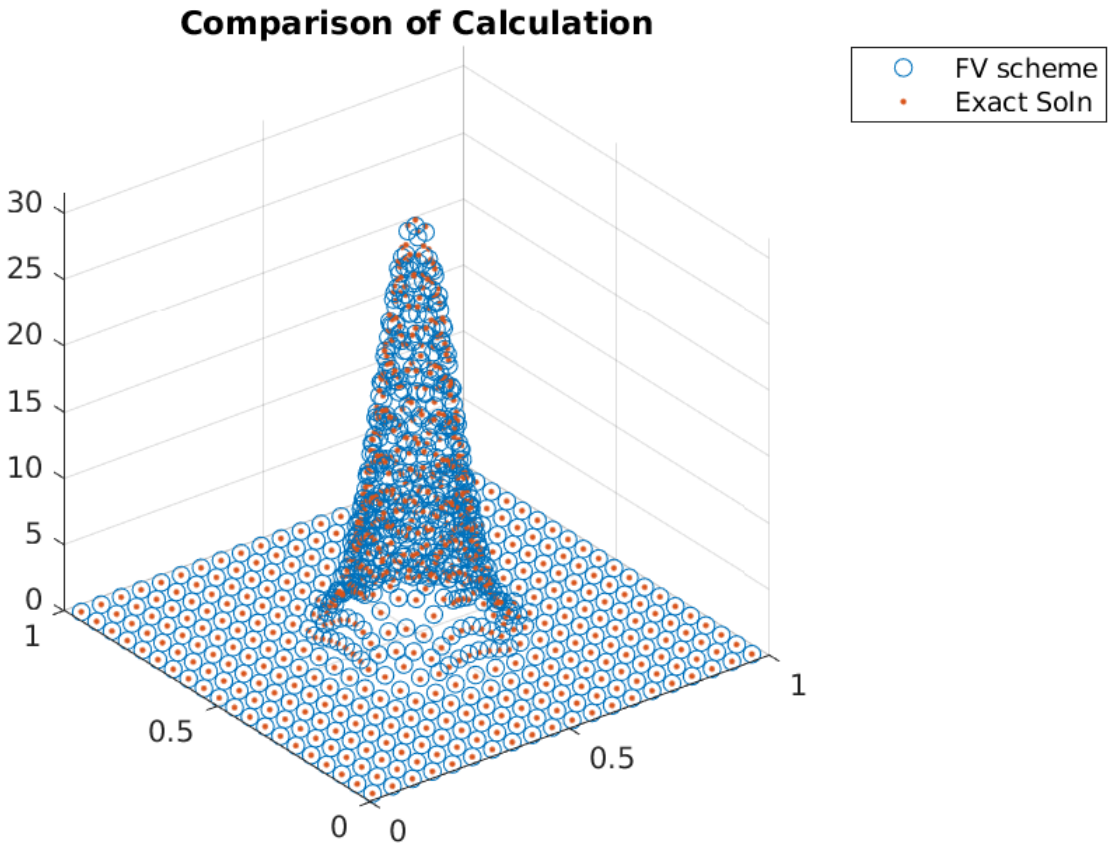


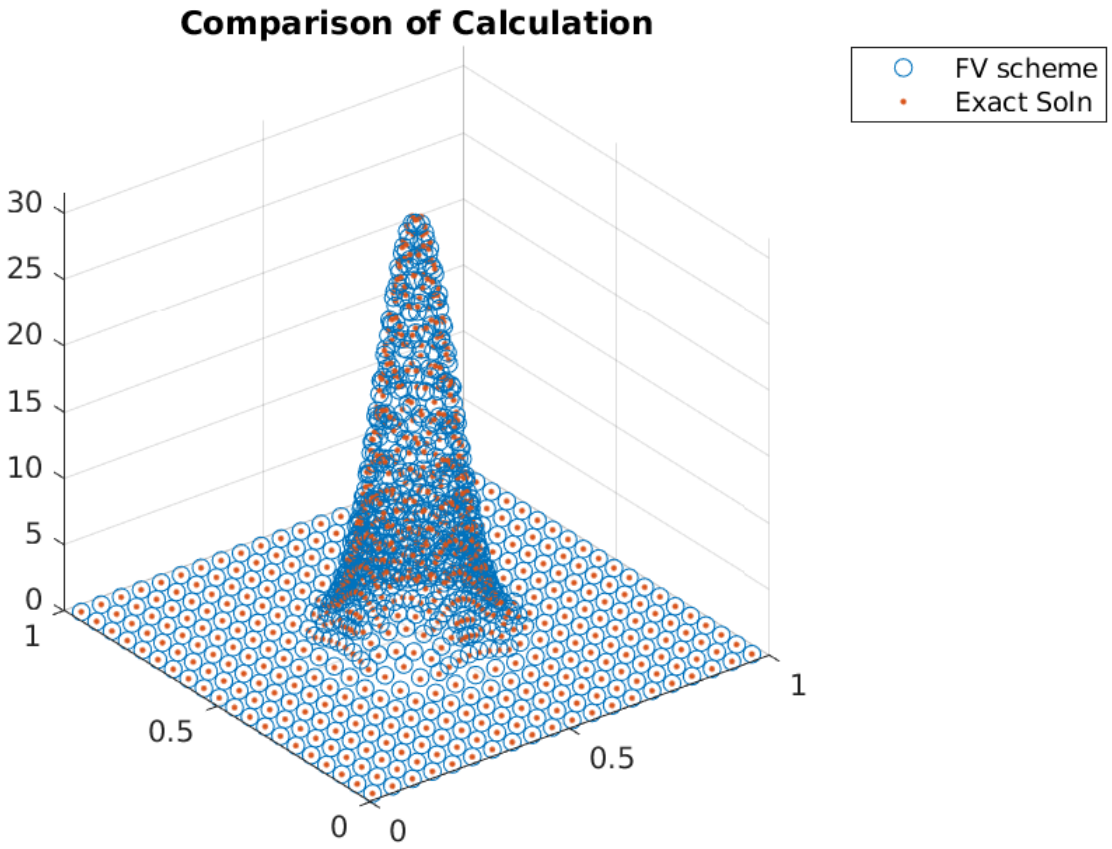
Iterating a few more times, we can clearly see that the adaptive refinements lead to accuracy gains without adding points in regions that already have zero mass.


Comparison of Calculation
FV scheme
Exact Soln
30
25
20
15
10
5
0
1
0.5
0
0
0.5
1


Comparison of Calculation
FV scheme
Exact Soln
30
25
20
15
10
5
0
1
0.5
0
0
0.5
1


Comparison of Calculation
FV scheme
Exact Soln
30
25
20
15
10
5
0
1
0.5
0
0
0.5
1

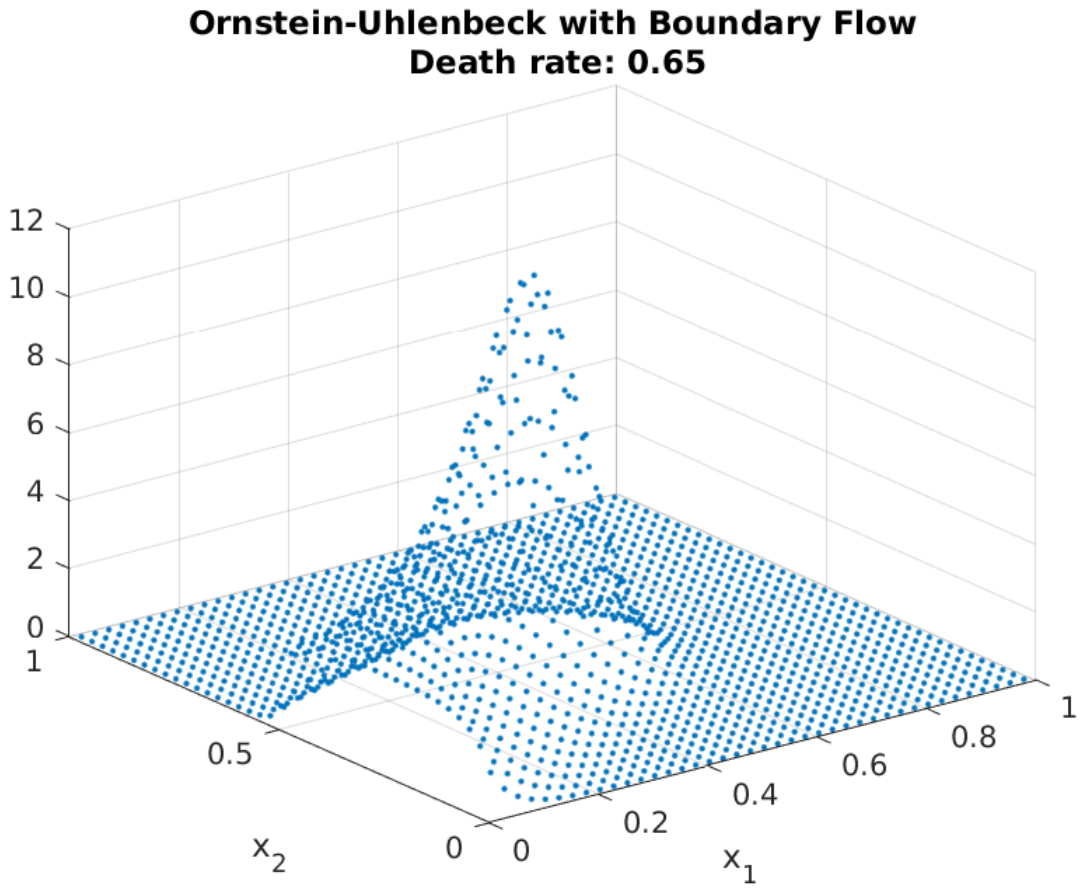

Comparison of Calculation
FV scheme
Exact Soln
30
25
20
15
10
5
0
1
0.5
0
0
0.5
1



2.2.4 OU Process with Boundary Conditions

Expected Read Time: XXXXX

In this tutorial, we will work through how to handle the boundary condition. We will still work with the Ornstein-Uhlenbeck process, but introduce "birth" of agents the edge $x_1 = 0$ and $x_2 \in [0, 0.5]$. To make the equation to have a steady state distribution, we also introduce death rate (constant proportion). Looking at the steady-state distribution would actually be easier to follow. In the figure below, we have the resulting steady-state distribution from the dynamics.



One see clear mass of people in the "birth edge," and mass moving toward the center of the OU process. The actual shape of the resulting steady-state will depend on the actually parameter values.

Implementing these additions are also simple using the *compute_transition_matrix_boundary* function.

Continuing from tutorial 2 of the grid definition for the OU-process, we only need to introduce transition matrix. We will handle "death" first. The syntax for this is

```
>> A_FP = A_FP + grid.compute_transition_matrix_boundary(1:grid.num_n, ones(grid.num_
↪n, 1), -death_rate*ones(grid.num_n, 1), ones(grid.num_n, 1));
```

where *compute_transition_matrix_boundary* takes (node to add flow, direction of flow, flow rate, normal direction). Hence, this function will take `death_rate` from each cell without accounting them into a different cell.

With the inflow, the flow-rate is not dependent on the current internal distribution. Hence, the functional form we need to handle is

where b stands for the inflow. We can compute this flow again with the same function. However, first we need to find the relevant edge, which we do so by

```
>> left_boundary = grid.n2bd(:,1) == 0 & (grid.n2bd(:,2) < 0.5);
>> ind_left = find(left_boundary);
```

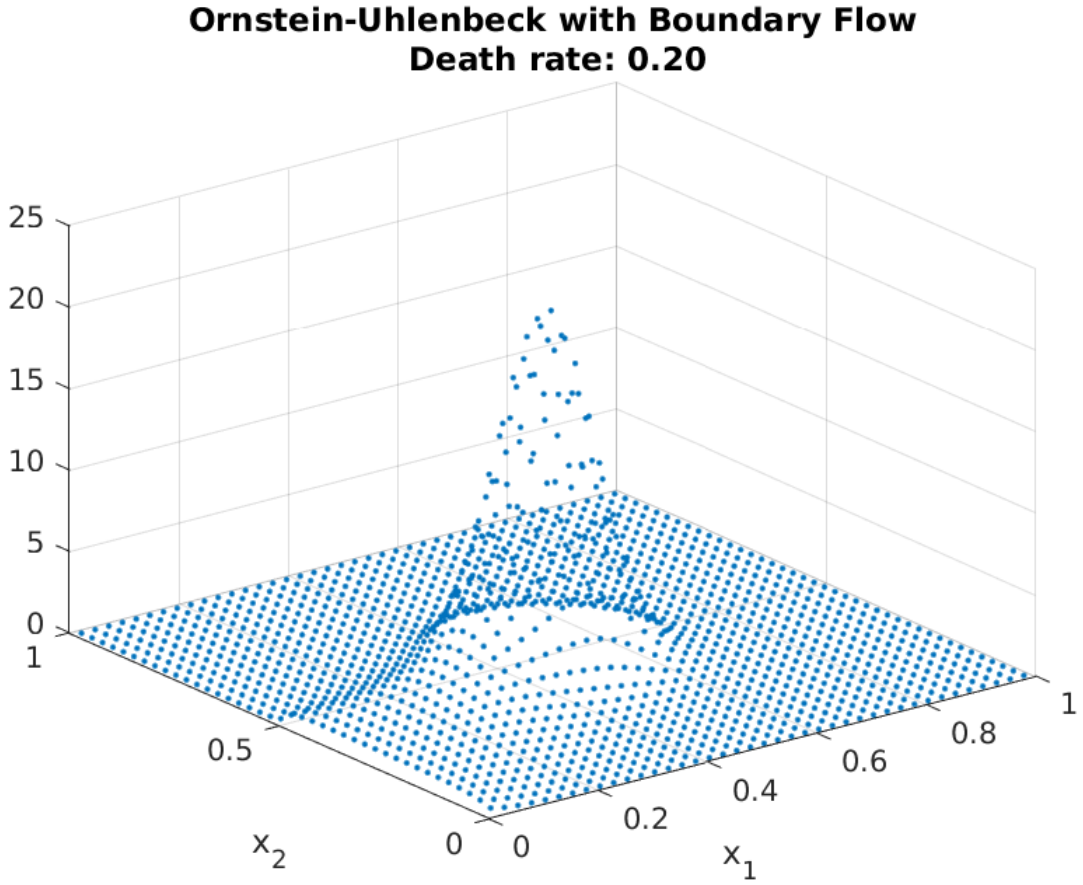
which check whether $x_1 = 0$ and $x_2 \in [0, 0.5]$. Given this, we fill in the relevant information for the *compute_transition_matrix_boundary* function.

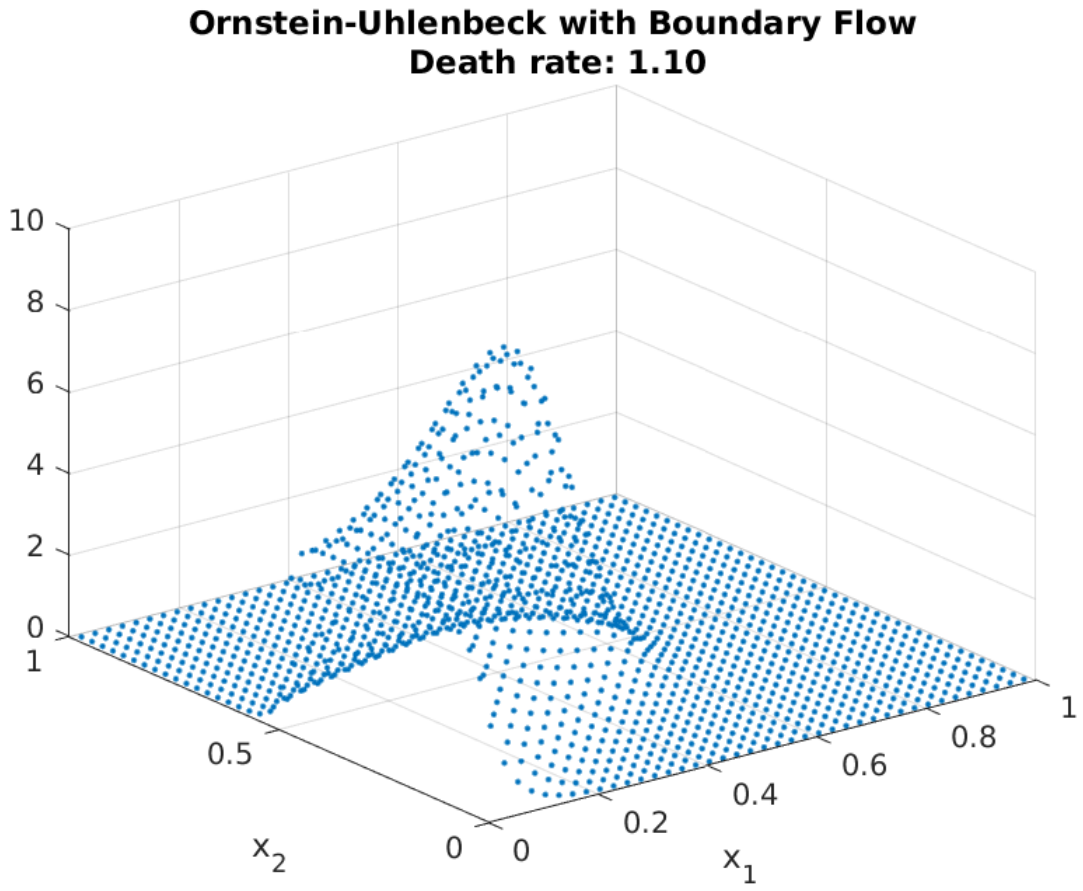
```
>> n_ind = length(ind_left);
>> flow = flow_rate*ones(n_ind, 1);
>> direction = ones(n_ind, 1);
>> is_left = true(n_ind, 1);
```

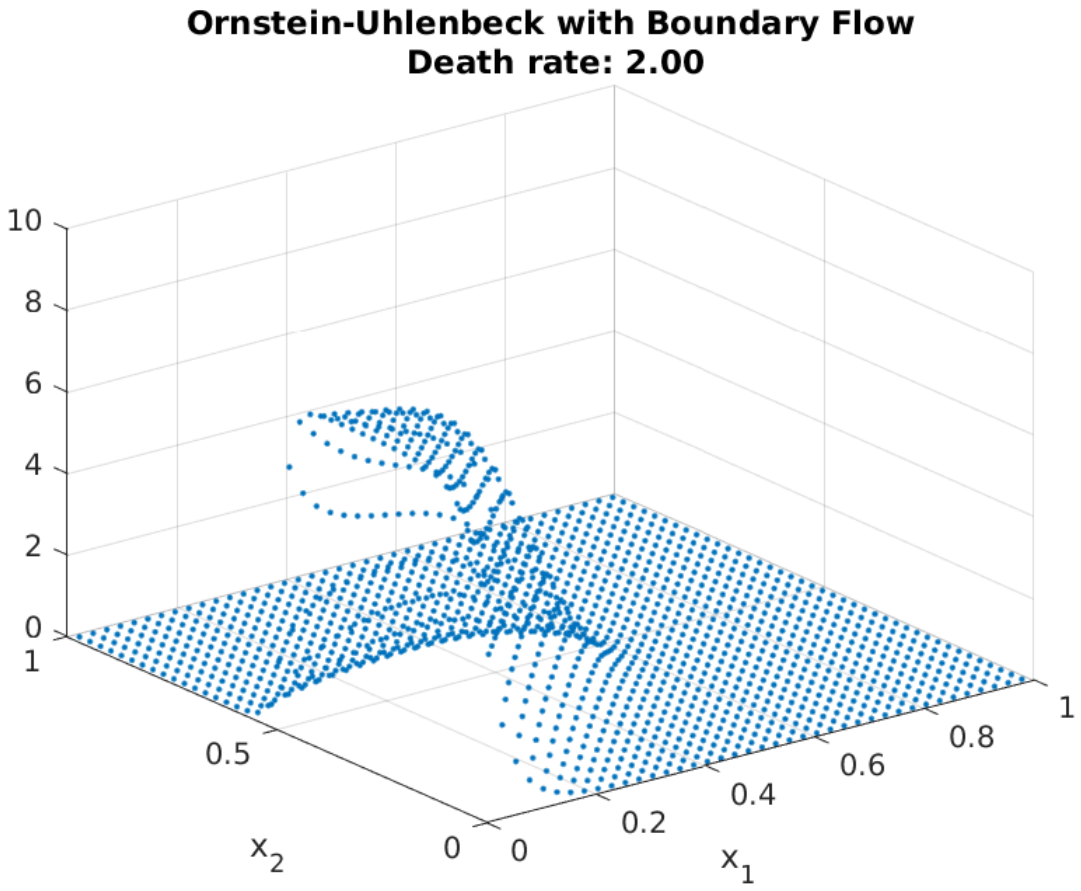
and compute the distribution as usual

```
>> g = A_FP\(grid.compute_transition_matrix_boundary(ind_left, direction, flow, is_
↪left)*ones(grid.num_n, 1));
>> g = g./sum(g);
```

Resulting in the distribution given above. As with before, adjusting for different parameter values are straight forward. As can be seen from the following plot, the distribution mirrors the steady-state of the OU-process with low death rate, but mirrors the entry more strongly with high death rates (as agents do not have enough time to approach the steady-state locations)




Ornstein-Uhlenbeck with Boundary Flow
Death rate: 1.10
10
8
6
4
2
0
1
0.5
0
x_2
0
0.2
0.4
0.6
0.8
1
x_1



This concludes the tutorials, and should be sufficient for most economic applications. The *afv_grid* is written to abstract away all implementation details, but if you find an application where the abstraction is too restrictive, one can refer to the *Technical Documentation* to see the underlying implementations.

2.3 Technical Documentation

This is the technical documentation page. See *Tutorials* to get started.

2.3.1 Properties

afv_grid.**diffusion**

Type *num_e*×1 vector of doubles

Description Diffusion across the given edge. It is set by the users before computing transition matrices

afv_grid.**drift**

Type *num_e*×1 vector of doubles

Description Drift across the given edge. It is set by the users before computing transition matrices

afv_grid.**e2bd**

Type ($num\_e \times (2 \cdot afv\_grid.n\_dim)$)-matrix of doubles

Description Boundary values of edges

Note boundaries are stacked in lexicographic order of (coordinate direction, left/right boundary)

afv_grid.**e2dir**

Type ($num\_e \times 1$)-vector of integers

Description Normal/coordinate direction of the edge

afv_grid.**e2n**

Type ($num\_e \times 2$)-matrix of integers

Description Connectivity graph from edges to nodes

afv_grid.**e_weights**

Type ($num\_e \times 1$)-vector of doubles

Description (integral) weights of the edges

afv_grid.**n2bd**

Type ($afv\_grid.num\_e \times (2 \cdot afv\_grid.n\_dim)$)-matrix of doubles

Description Boundary values of the cell

Note boundaries are stacked in lexicographic order of (left/right boundary, coordinate direction)

afv_grid.**n2e**

Type ($afv\_grid.num\_e \times afv\_grid.n\_dim \times 2$) cell array of vector of integers

Description Connectivity graph from nodes to edges.

afv_grid.**n2n**

Type ($num\_n \times n\_dim \times 2$) cell of integers

Description Neighbor structure of nodes. The third index corresponds to

- 1: left neighbors
- 2: right neighbors

afv_grid.**n_dim**

Type integer

Description Dimensionality of the grid

afv_grid.**n_points**

Type integer

Description Total number of nodes

Warning This is DEPRECATED, and will be eventually removed. Kept for code-reusability with Mean Field Games Toolbox. Use num_e or num_n instead.

afv_grid.**n_weights**

Type ($num\_n \times 1$) vector of doubles

Description Size of the nodes

afv_grid.**num_e**

Type integer

Description Total number of edges

afv_grid.**num_n**

Type integer

Description Total number of nodes

afv_grid.**num_nb**

Type (*num_n*×*n_dim*×2) array of integers

Description Number of neighbors of nodes. The third index corresponds to

- 1: left neighbors
- 2: right neighbors

afv_grid.**x_i**

Type (*n_dim*×1) cell array of vectors of doubles.

Description Coordinate value of edges or nodes. Users can set this directly.

See also XXXXXXXX

2.3.2 Functions

afv_grid.**add_new_nodes**(*obj*, *n2bd_new*, *ind_to_consider*, *flag_compute_left*)
Add new nodes to the grid

Parameters

- **n2bd_new** – boundaries of new node points
- **ind_to_consider** (*vector of indices*) – (optional) indices of nodes that have interface with new nodes
- **flag_compute_left** (*bool*) – (default: false) whether to compute left neighbors as well.

Returns

[n2n] (implicit in class)

- **n2n** : (in class) neighbor structure

Note: The computing neighbor structure from scratch is an expensive operation. Therefore, one should use new_ind whenever possible. This behavior is guaranteed if one computes using *split_init* or *split*, so one should use that function whenever it is feasible.

---

afv_grid.**afv_grid**(*n_dim*)
Empty Constructor

Parameters **n_dim** (*integer*) – Dimensionality of the grid

See also:

set_dim

afv_grid.**compute_diffusion_distance**(*obj*, *ind*)
INTERNAL METHOD: Calculate the diffusion distance to approximate $\frac{dg}{dx}$ for diffusion in finite volume method

Parameters **ind** (*vector of indices*) – indices of the node to compute the diffusion distance

Returns the distance to be used in diffusion computation.

Return type r_dist (vector of doubles)

afv_grid.**compute_edge_midpoints**(*obj*, *ind*, *dir*, *is_left_edge*)
Compute the midpoint of the given edge of a node

Parameters

- **ind** (*n-vector of indices*) – indices of the **node** to compute the mid-point of the edge
- **dir** (*n-vector of integers*) – coordinate direction of the edge to consider
- **is_left_edge** (*n-vector of bools*) – whether computing the mid-points of the left edge or not

Returns midpoints of the edge

Return type midpts (n × *n_dim* of doubles)

Note: This function is to calculate external edges of the node that does not face a different node. For internal nodes, consider using function *edge_midpoints*

---

afv_grid.**compute_node_bds**(*obj*, *x_knots*)
INTERNAL FUNCTION: computes boundaries of new nodes

Parameters **x_knots** (*cell of vector of doubles*) – the boundary knots that define the tensor cut split of a node

Returns boundaries of the nodes

Return type output (matrix of doubles)

afv_grid.**compute_num_neighbors**(*obj*)
Compute the number of neighbors

Parameters **n2n** – (implicit) *n2n*

Returns [num_nb] implicit in class

afv_grid.**compute_transition_matrix_boundary**(*obj*, *ind*, *dir*, *flow*, *is_left_edge*)
Compute the transition matrix corresponding to the boundary of the given node

Parameters

- **ind** (*vector of indices*) – indices of the node to compute the edge dynamics
- **dir** (*vector of integers*) – coordinate directions of the edges/flows to consider
- **flow** (*vector of doubles*) – flow/drift rate across the edges
- **is_left_edge** (*vector of bool*) – whether the edge is a left edge or not

Returns the transition matrix of the FPK-equation for the given flows across the edges.

Return type A_eFP (*num_n*×*num_n* sparse matrix of doubles)

Note: There is *compute_transition_matrix_modified* function that handles the construction of the transition matrix for internal edges facing other cells. This function is supposed to be used for creating the flows for the boundary conditions.

afv_grid.**compute_transition_matrix_center**(*obj*)
Compute the transition matrix corresponding to the Fokker Planck equation

$$\frac{dg}{dt} = -\frac{d}{dx}\left(s(x) \cdot g(x)\right) + \nu \frac{d^2 g}{dx^2}$$

using central difference approximation.

Parameters

- **e2n** – implicit in class
- **drift** – implicit in class
- **diffusion** – implicit in class
- **node_weights** – implicit in class
- **e_weights** – implicit in class

Returns the transition matrix from the FPK-equation

Return type A_FP (*num_n*×*num_n* sparse matrix of doubles)

Note:

- Many separate parts are needed before this function can be called properly. Instead of setting them manually, try to use the given functions that guarantee the internal structure.
- This function is for internal edges, for external edges, use *compute_transition_matrix_boundary*.
- Central value approximation is not guaranteed to be stable, if the solution do not behave well consider using *compute_transition_matrix_modified*.

afv_grid.**compute_transition_matrix_modified**(*obj*, *weighter*)
Compute the transition matrix corresponding to the Fokker-Planck equation

$$\frac{dg}{dt} = -\frac{d}{dx}\left(s(x) \cdot g(x)\right) + \nu \frac{d^2 g}{dx^2}$$

using the modified upwind scheme (equation 2.3) of *[Axelsson & Gustafsson, 1979]*.

Parameters

- **e2n** – implicit in class
- **drift** – implicit in class
- **diffusion** – implicit in class
- **node_weights** – implicit in class
- **e_weights** – implicit in class

Returns the transition matrix from the FPK-equation

Return type A_FP (*num_n*×*num_n* sparse matrix of doubles)

Note:

- Many separate parts are needed before this function can be called properly. Instead of setting them manually, try to use the given functions that guarantee the internal structure.
- This function is for internal edges, for external edges, use *compute_transition_matrix_boundary*.

References

- *[Axelsson & Gustafsson, 1979]*

afv_grid.**compute_transition_matrix_upwind**(*obj*)

Compute the transition matrix corresponding to the Fokker Planck equation

$$\frac{dg}{dt} = -\frac{d}{dx}\left(s(x) \cdot g(x)\right) + \nu \frac{d^2 g}{dx^2}$$

using upwind approximation.

Parameters

- **e2n** – implicit in class
- **drift** – implicit in class
- **diffusion** – implicit in class
- **node_weights** – implicit in class
- **e_weights** – implicit in class

Returns the transition matrix from the FPK-equation

Return type A_FP (*num_n*×*num_n* sparse matrix of doubles)

Warning: Upwind introduced numerical diffusion, so the solution is not accurate compared to other methods. Use *compute_transition_matrix_modified* unless there is a strong reason to use the upwind scheme.

Note:

- Many separate parts are needed before this function can be called properly. Instead of setting them manually, try to use the given functions that guarantee the internal structure.
- This function is for internal edges, for external edges, use *compute_transition_matrix_boundary*.

afv_grid.**edge_midpoints**(*obj*, *ind*)

Compute mid-points of the edges

Parameters **ind** (*vector of indices*) – indices of the edges to compute the mid-points

Returns mid-points of the edges

Return type e_midpoints (vector of doubles)

See also:

- *compute_edge_midpoints*

afv_grid.**edge_weights**(*obj*, *ind*)
Compute weight of the given edge

Parameters **ind** (*vector of indices*) – indices of the edges to compute the surface area

Returns surface area of the edges

Return type e_weight (vector of doubles)

afv_grid.**extract_edges**(*obj*)
Extract edges from nodal connections. This function only finds internal edges

Parameters

- **n2n** – (implicit) *n2n*
- **n2bd** – (implicit) *n2bd*

Returns

[e2bd, e2n, e2dir, e_weights, num_e] (implicit in class)

- **e2bd** : (in class) boundaries of the edges
- **e2n** : (in class) connectivity of the edges to the nodes
- **e2dir** : (in class) normal direction of the edge
- **e_weights** : (in class) surface area of the edge
- **num_e** : (in class) total number of edges

afv_grid.**find_neighbor**(*obj*, *ind*, *dir*, *is_left*, *varargin*)
Find the neighboring node for the given (single) node

Parameters

- **ind** (*integer*) – index of the node to find the neighbors of
- **dir** (*integer*) – direction to find the neighbors in
- **is_left** (*bool*) – whether to look for left neighbors or not
- **varargin{1}** (*vector of indices*) – indices of nodes to consider for neighbors (assumed unique)
- **n2bd** – (implicit in class) *n2bd*
- **n_dim** – (implicit in class) *n_dim*

Returns nbs (vector of indices): indices of the neighbors

Note: `find_neighbor_structure()` runs this function for all nodes.

---

afv_grid.**find_neighbor_structure**(*obj*, *new_ind*, *flag_compute_left*)
Construct neighbor structure

Parameters

- **new_ind** (**sorted** vector of indices) – (optional) indices of new nodes to append to the neighbor structure
- **flag_compute_left** (*bool*) – (default: false) whether to compute left neighbors as well

Returns

[n2n] (implicit in class)

- **n2n** : (in class) neighbor structure

Note: The computing neighbor structure from scratch is an expensive operation. Therefore, one should use new_ind whenever possible. This behavior is guaranteed if one computes using *split_init* or *split*, so one should use that function whenever it is feasible.

---

afv_grid.**node_midpoints**(*obj*, *ind*)
Compute mid-points of the nodes

Parameters **ind** (*vector of indices*) – (default: ':') indices of the nodes to compute the mid-points

Returns n_midpoints (vector of doubles): mid-points of the nodes

afv_grid.**node_weights**(*obj*, *ind*)
Compute the weight of the given node

Parameters **ind** (*vector of indices*) – indices of the nodes to compute the volume

Returns volume of the nodes

Return type n_weight (vector of doubles)

afv_grid.**set_dim**(*obj*, *n*)
Sets dimensionality of the problem. This function works as the initializer for the class

Parameters **n** (*int*) – dimensionality of the problem

Returns (implicit in class)

Note: Due to object-oriented design, the set of codes assumes a certain internal consistency of states. Hence, one should always set dimentionality of the problem through *set_dim*.

---

afv_grid.**split**(*obj*, *ind*, *x_cuts*, *flag_compute_left*)
Split nodes

Parameters

- **ind** (*vector of indices*) – indices of the nodes to split
- **x_cuts** (*cell of vector of double*) – for each (node, dimension) the cut points to split the node
- **flag_compute_left** (*bool*) – (default: false) whether to compute left neighbors as well

Returns updates all node structure and edge structure of the grid

afv_grid.**split_init**(*obj*, *node_ind*, *x_cuts*)
Split a given node which is a starting point

Parameters

- **node_ind** (*integer*) – index of the cell to split
- **x_cuts** (*cell of vectors*) – points to introduce new edges

Returns (implicit in class) internally update states to introduce new cells for the given cell

2.4 License

2.5 Bibliography

You can cite this toolbox with the bib-entry

```
@misc{ahn2018afvtoolbox,
    author={Ahn, SeHyoun},
    title={Adaptive Finite Volume Toolbox (Version 0.1) [Computer Software]},
    url={https://github.com/sehyoun/adaptive_finite_volume},
    year={2017-2018},
}
```

after updating to the correct version number, e.g., *[Ahn, 2017-2019]*.[1]

[1] One can also use `software` entry, but not all bibtex recognize `software` entries. Just make sure to include the version number for the replicability.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

[Ahn, 2017-2019] SeHyoun Ahn. Adaptive finite volume toolbox (version 0.1) [computer software]. 2017-2019. URL: https://github.com/sehyoun/adaptive_finite_volume.

[Ahn, 2019] SeHyoun Ahn. Adaptive finite volume methods for economic modelling. *Norges Bank Working Paper*, 2019.

[Axelsson & Gustafsson, 1979] Owe Axelsson and Ivar Gustafsson. A modified upwind scheme for convective transport equations and the use of a conjugate gradient method for the solution of non-symmetric systems of equations. *IMA Journal of Applied Mathematics*, 23(3):321–337, 1979.

INDEX