Ditto: Experimenting with Moose framework

Contents

[1 Useful mathematical Identities 3](#_Toc164069675)

[1.1. Continuity equation: 3](#_Toc164069676)

[1.2. Moose inner product notation: 3](#_Toc164069677)

[1.3. Integration by parts and divergence theorem 3](#_Toc164069678)

[2 Moose references 4](#_Toc164069679)

[3 How to create the “Ditto” moose project 6](#_Toc164069680)

[4 Examples 7](#_Toc164069681)

[4.1. Diffusion (steady) 7](#_Toc164069682)

[4.2. ADDiffusion (steady) 9](#_Toc164069683)

[4.3. ADDiffusion (transient) 10](#_Toc164069684)

[4.4. Parameterized diffusivity 12](#_Toc164069685)

[4.5. Introducing Materials: spatio-temporal diffusivity 13](#_Toc164069686)

[4.6. Initial condition (IC) 15](#_Toc164069687)

[4.7. Boundary conditions (BC) and post-processors. 18](#_Toc164069688)

[4.8. Adaptive grid 20](#_Toc164069689)

[4.9. Adaptive step 22](#_Toc164069690)

[4.10. Diffusion-ConservativeAdvection 23](#_Toc164069691)

[5 Paraview notes 25](#_Toc164069692)

# Useful mathematical Identities

## Continuity equation:



where the first term is the rate of change of *c*, **j** is the flux and *R* the source/sink term for *c*.

## Moose inner product notation:





## Integration by parts and divergence theorem

Let *ψ* be a scalar variable and **u** a vector function. The divergence of their produce is:



By integrating over a domain Ω and rearranging we get:



The divergence theorem transforms a volume integral into a surface integral:



with being an outward normal vector on surface **. By combining eqs and we get:



# Moose references

**About Moose**  
<https://mooseframework.inl.gov/workshop/index.html#/2>

**FEM**

<https://mooseframework.inl.gov/workshop/index.html#/8>

*Demonstration for the advection diffusion equation*

<https://mooseframework.inl.gov/workshop/index.html#/8/9>

*Shape functions*

<https://mooseframework.inl.gov/workshop/index.html#/9>

*Numerical implementation (quadrature, actual weak form)*

<https://mooseframework.inl.gov/workshop/index.html#/10>

**Solvers**

*Newton*

<https://mooseframework.inl.gov/workshop/index.html#/10/>7

*Automatic vs Manual Jabobian*

<https://mooseframework.inl.gov/workshop/index.html#/10/14>

**Kernel system**

<https://mooseframework.inl.gov/workshop/index.html#/19>

**Mesh system**

<https://mooseframework.inl.gov/workshop/index.html#/22>

**Output system**

<https://mooseframework.inl.gov/workshop/index.html#/23>

**Material System**

<https://mooseframework.inl.gov/workshop/index.html#/25>

**Function System**

<https://mooseframework.inl.gov/workshop/index.html#/26>

**Auxiliarly system**

<https://mooseframework.inl.gov/workshop/index.html#/30>

**Timestepper**

<https://mooseframework.inl.gov/workshop/index.html#/35>

**Boundary conditions**

<https://mooseframework.inl.gov/workshop/index.html#/37>

**Mesh adaptivity**

<https://mooseframework.inl.gov/workshop/index.html#/41>

**PostProcessors:**

<https://mooseframework.inl.gov/workshop/index.html#/44>

# How to create the “Ditto” moose project

1. Activate moose

conda activate moose

2. Generate the project files w/ stork

cd ~/projects

./moose/scripts/stork.sh Ditto

3. (Optional) Create GitHub repo called Ditto and push the project

cd ~/projects/Ditto

git remote add origin [git@github.com:[insert\_name]/Ditto.git](mailto:git@github.com:[insert_name]/Ditto.git)

git push -u origin main

4. Make the executable

make –j4 # e.g., with 4 processors

5. Run the tests

./run\_tests

6. Generate a problems folder

mkdir problems

# Examples

## Diffusion (steady)

We will generate a script for steady diffusion in 2D using the Dirichlet boundary conditions *u*left = 1 and *u*right = 0.

Let’s derive the weak form in inner product notation:

Strong form:



Multiply with a test function *ψ* and integrate over the domain



Integrate the first term by parts and apply the divergence theorem (eq. with **u** → ):



Expressing in the inner product notation:



We will set the domain to from -10, +10 along the *x* and *y* directions.

The problems is solved with the script ex01\_diffusion.i:

[Mesh]

type = GeneratedMesh

dim = 2

nx = 40

ny = 40

ymin = -10.0

ymax = 10.0

xmin = -10.0

xmax = 10.0

elem\_type = QUAD4

[]

[Variables]

[uu]

order = FIRST

family = LAGRANGE

[]

[]

[Kernels]

[diff]

type = Diffusion

variable = uu

[]

[]

[BCs]

[left\_uu]

type = DirichletBC

variable = uu

boundary = 'left'

value = 1

[]

[right\_uu]

type = DirichletBC

variable = uu

boundary = 'right'

value = 0

[]

[]

[Executioner]

type = Steady

solve\_type = 'PJFNK'

[]

[Outputs]

execute\_on = 'timestep\_end'

exodus = true

[]

We can run the script with the command:

./run.sh ex01\_diffusion.i

Paraview output:

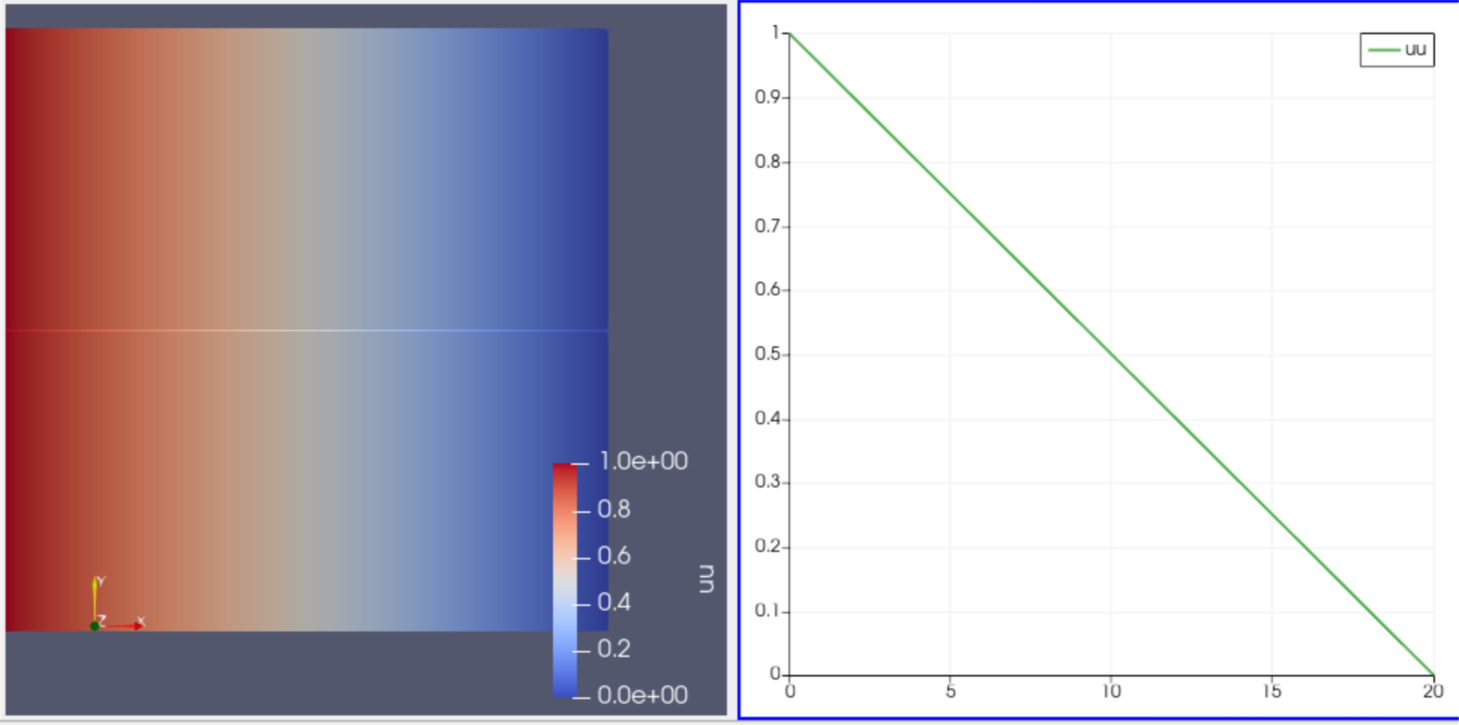
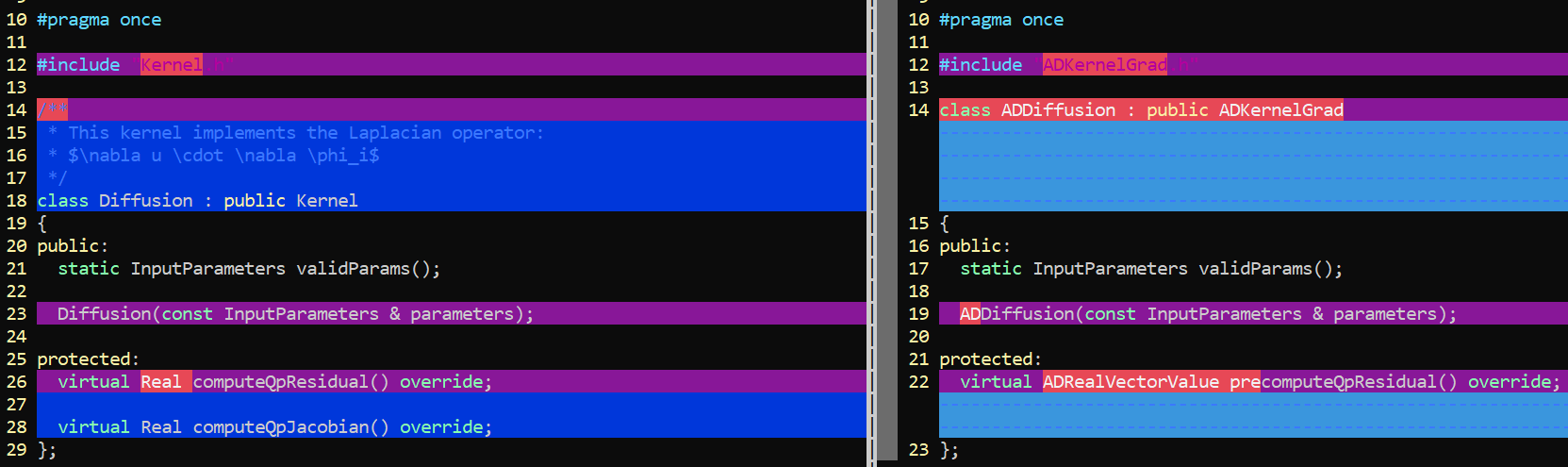


Figure . output from exodus file ex01\_diffusion\_out.e

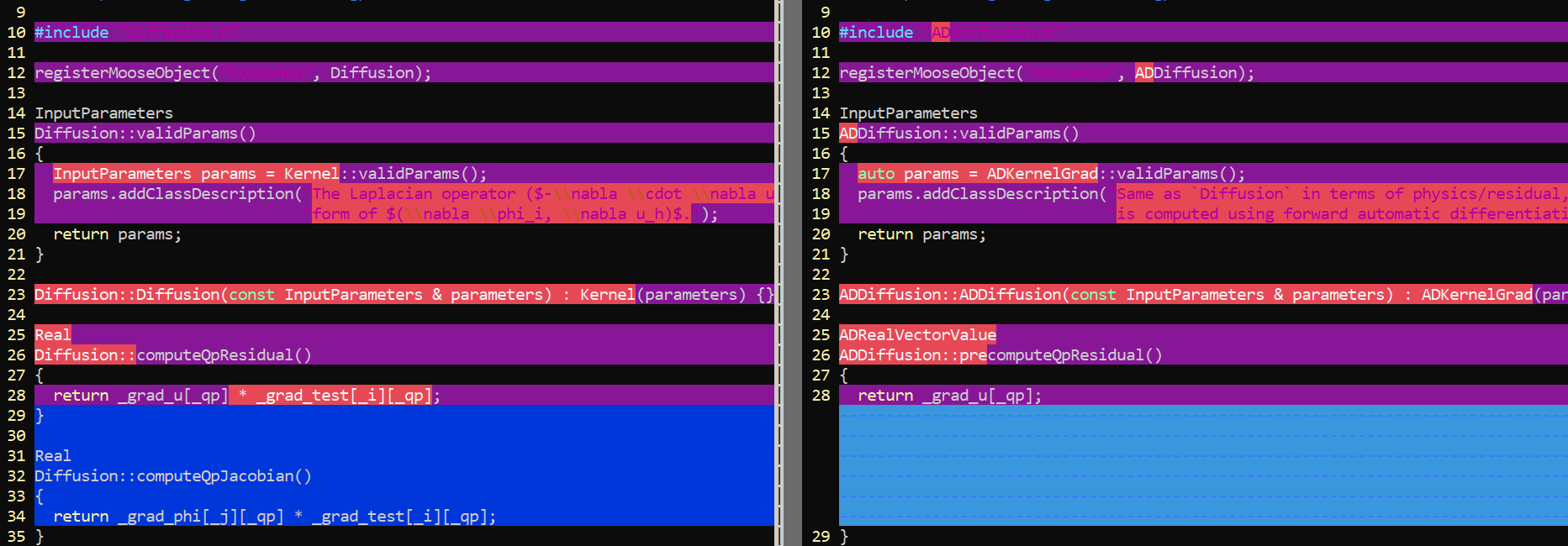
## ADDiffusion (steady)

Same with example 4.1 but we will use automatic differentiation.

The implementations of the kernels in path are shown below:









**Notes**

* Automatic differentiation (AD) does not require specifying the Jacobian. The latter is determined numerically and is guaranteed to be correct. This comes very handy in complex PDEs (e.g., with spatio-temporal variations of the material properties)
* The downside is that it is ~1.5 times slower.
* The general guideline is to develop the model with AD, and then proceed into developing the Jacobian.
* The test function has been already multiplied in the precomputeQpResidual.

## ADDiffusion (transient)

Let’s derive the weak form in inner product notation:

Strong form:



Multiply with a test function *ψ* and integrate over the domain



Integrate the first term by parts and apply the divergence theorem (eq. with **u** → ):



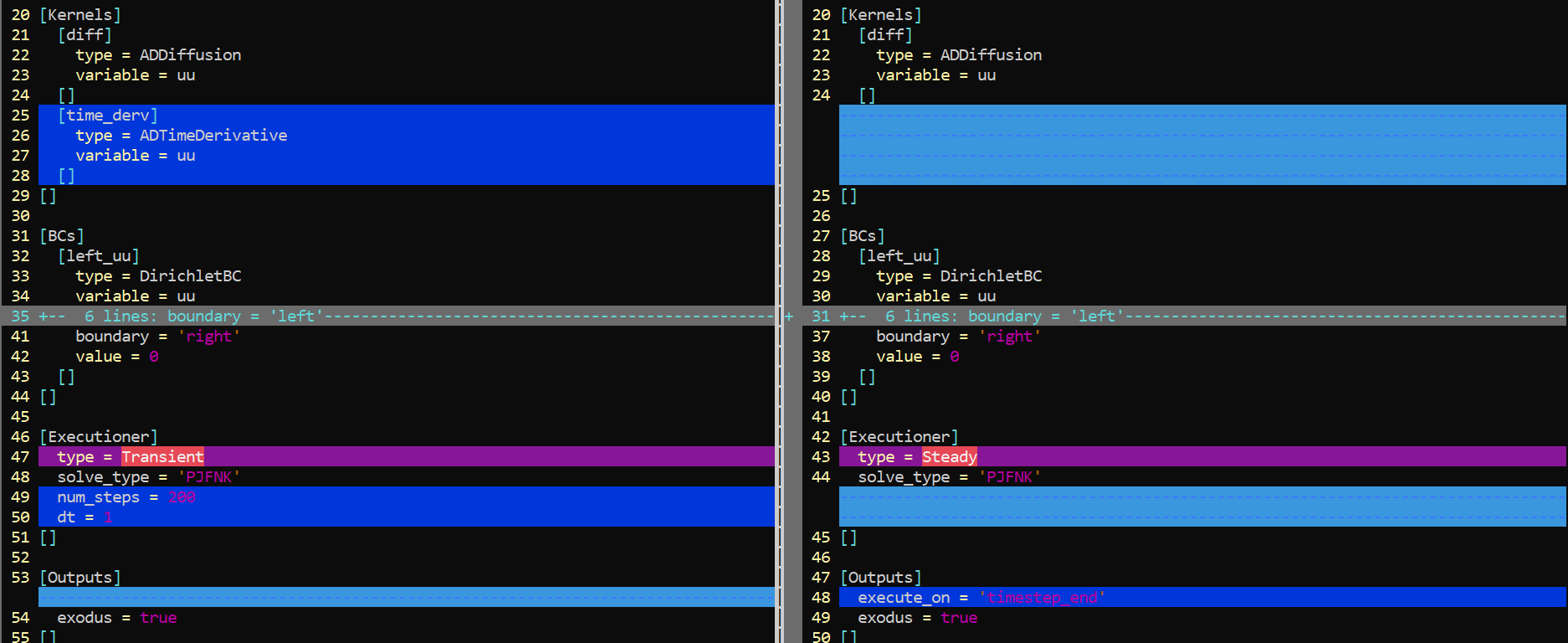
Expressing in the inner product notation:



We will set the domain to from -10, +10 along the *x* and *y* directions.

The kernel in invoked in example ex03\_transient.i.

Difference wrt example 4.2:



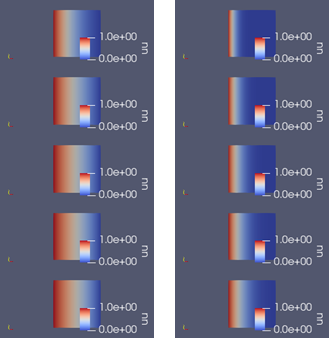


**Notes:**

* Add kernel ADTimeDerivative
* Change executioner to Transient

**Output:**

The evolution of the solution is shown in the left panel of **Figure 2**.



**Figure 2.** Diffused quantity *uu* in steps 0, 40, 80, 120, 160 and 200. Diffusion coefficient = (left) 1 and (right) 0.1.

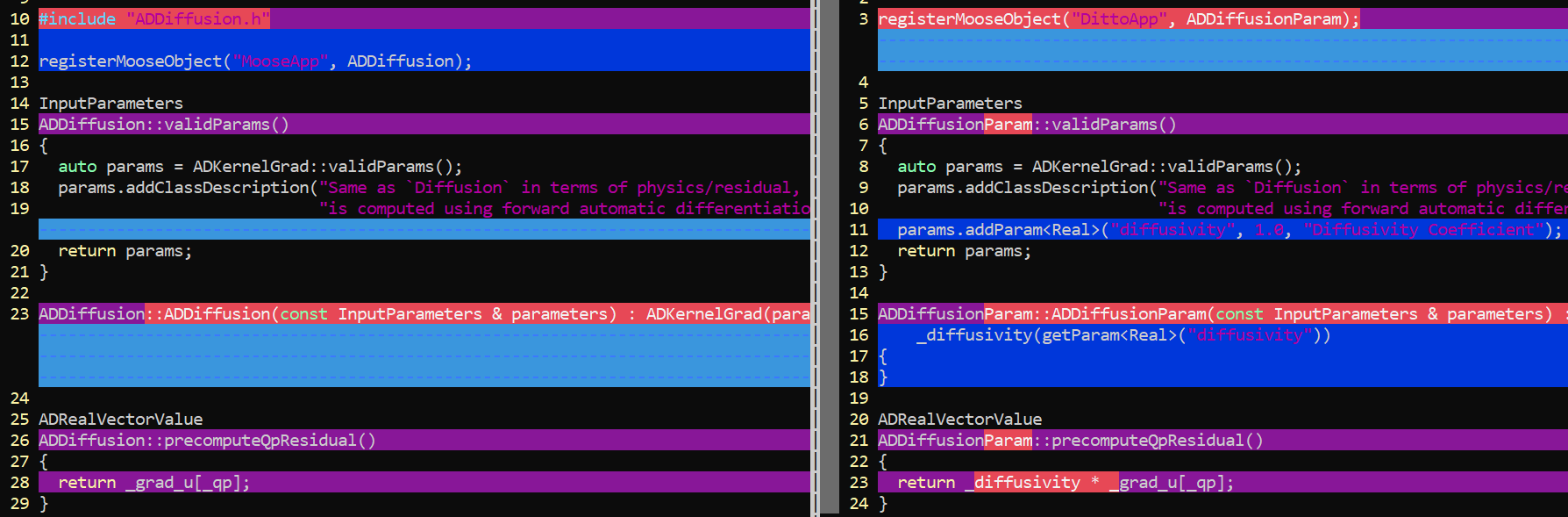
## Parameterized diffusivity

Same with example 4.3 but we will allow for varying the diffusion coefficient.

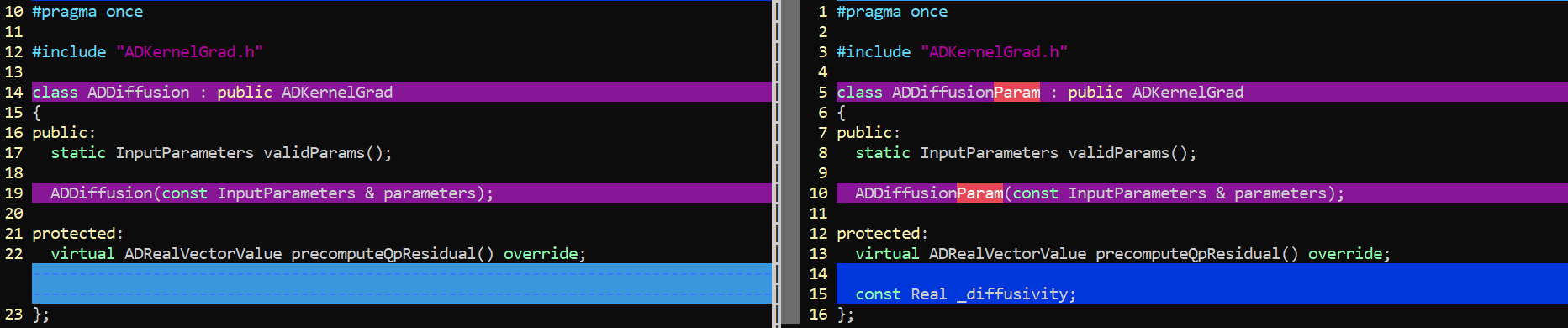
* The following files have been added in src/kernel/ADDiffusionParam.C and include/kernel/ADDiffusionParam.h
* The source was recompiled (Make –j6)

The files are similar with the default ADDiffusion kernels in moose framework with the exception that the diffusion coefficient has been made a parameter.

Comparisons are shown below:









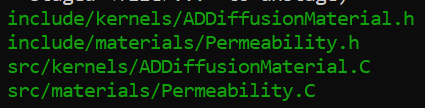
**Output:**

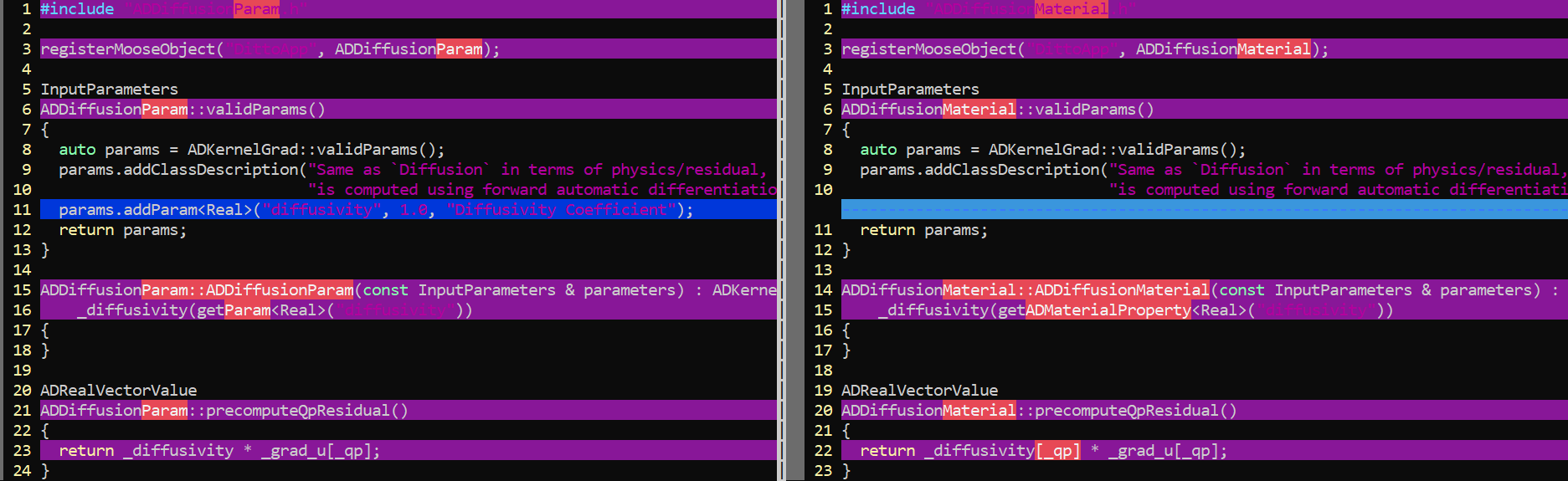
The evolution of the solution is shown in the right panel of **Figure 2**.

## Introducing Materials: spatio-temporal diffusivity

We will introduce a material object whose properties change with time and space, based on a parsed function.

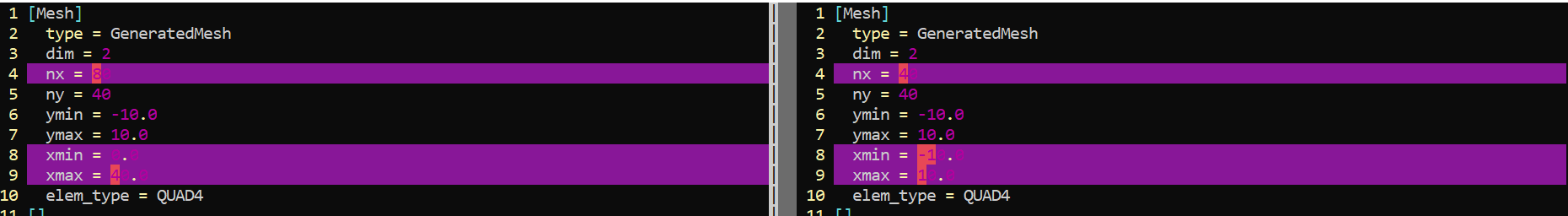
The following files have been introduced to the project:

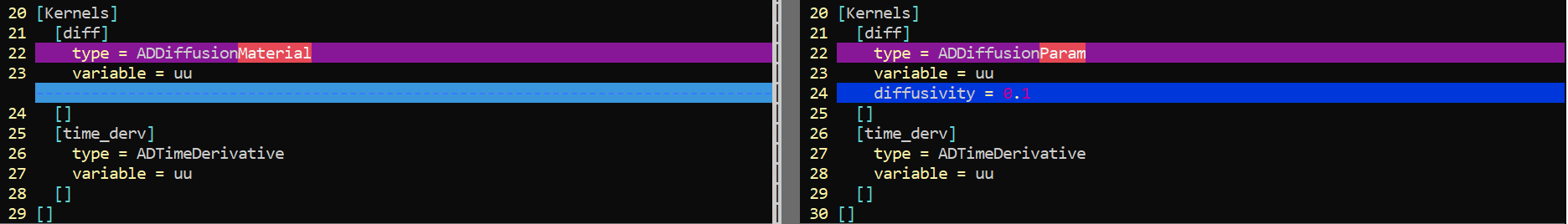


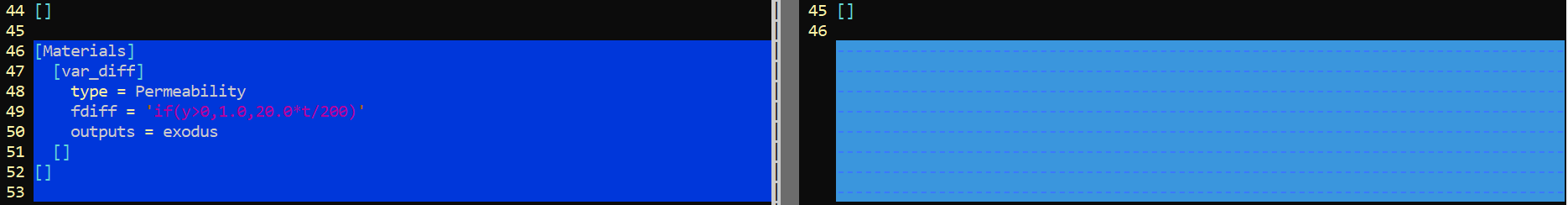
Let’s compare the changes to the diffusion kernel:  




Let’s compare the input of the present example wrt the one from example 4.4.









**Notes:**

* The box size have been increased to 80 along x-axis
* The diffusion term is described with the kernel ADDiffusionMaterial
* The properties of the material change with time and space according the parsed function in line 49:



In other words, the diffusion coefficient increases with time for y < 0, and equals to 1 for y > 0.

**Output:**

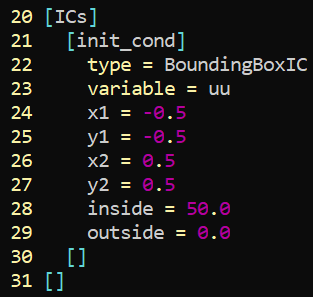


Figure . Steps 1, 2, 10, 20, 30, 40. At short times the diffusion at y<0 is 10 times smaller than that at y>0. The situation is reversed at long times since the diffusion at y<0 becomes 20 times larger than that at y>0.

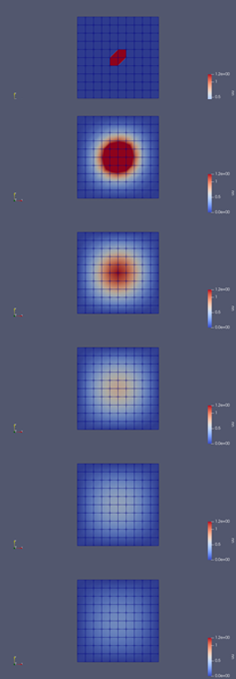
## Initial condition (IC)

The example ex06\_ic\_pulse.i applies an initial condition *u* = 50 across a unit square at the center of a box with dimensions 10×10 and *nx*=*ny*=10. Dirichlet BC, *u* = 0 are applied to the edges of the simulation box.

A corresponding snapshot is shown below:

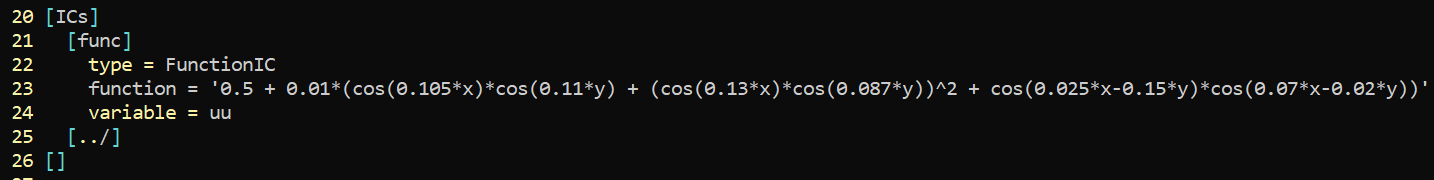


The evolution of the solution is illustrated in **Figure 4**.

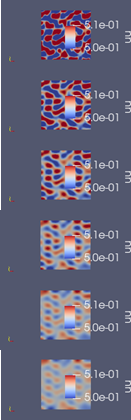


**Figure 4.** Evolution of the solution at steps 0, 2, 4, 6, 8 and 10.

The example ex06\_ic\_function\_neumann.i applies an initial condition based on the following parsed function:

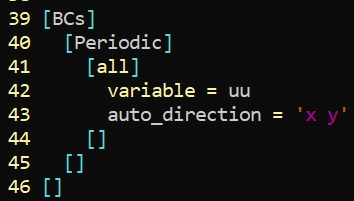


Neumann BC are applied at the edges of the domain with zero flux. The evolution of the solution is illustrated in **Figure 5**.



**Figure 5.** Evolution of the solution at steps 0, 40, 80, 120, 160 and 200.

The example ex06\_ic\_function\_per.i implements the periodic boundary conditions; e.g., see snippet below:

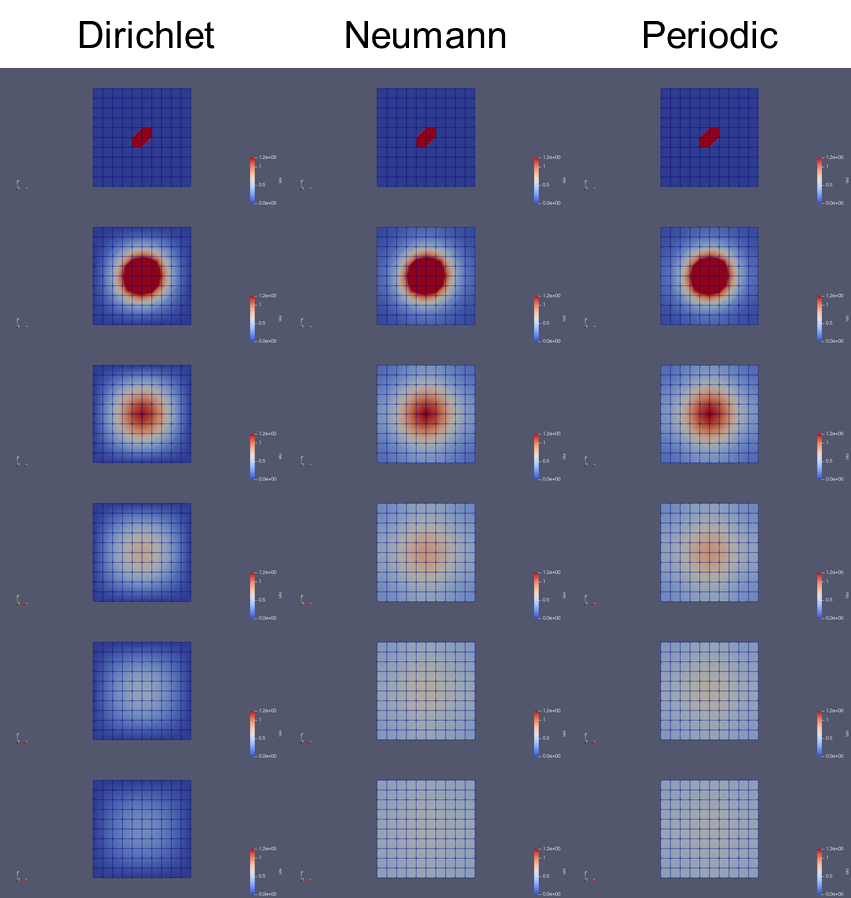


## Boundary conditions (BC) and post-processors.

Implementations of various boundary conditions are demonstrates in the examples:

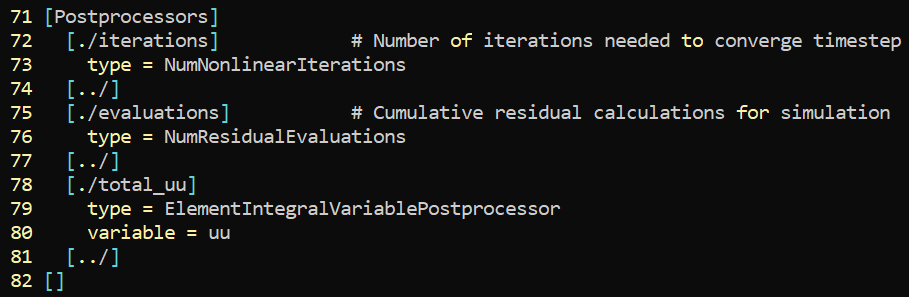
* ex07\_ic\_pulse\_dirichlet0
* ex07\_ic\_pulse\_neumann
* ex07\_ic\_pulse\_periodic

In each case, an initial condition *u* = 50 has been applied across a unit square at the center of a box with dimensions 10×10 and *nx*=*ny*=10. The evolution of the solution for each case is illustrated in **Figure 6**.



**Figure 6.** Evolution of the solution at steps 0, 2, 4, 6, 8 and 10.

In addition, the examples make use of the Moose post-processors; e.g., see indicative snippet:



In this example, the post-processor ElementIntegralVariablePostprocessor integrates the solution across the simulation domain. The results are shown in Figure 7. We notice that the integral of the solution is conserved when enforcing the Neumann and Periodic boundary conditions.

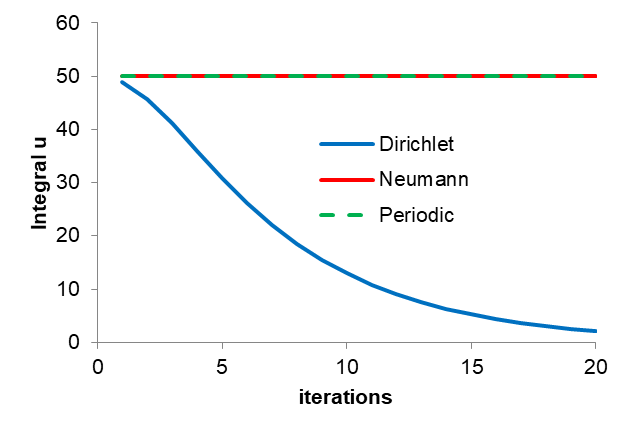
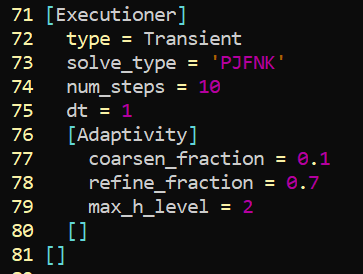


Figure . Solution integral during the course of the simulations.

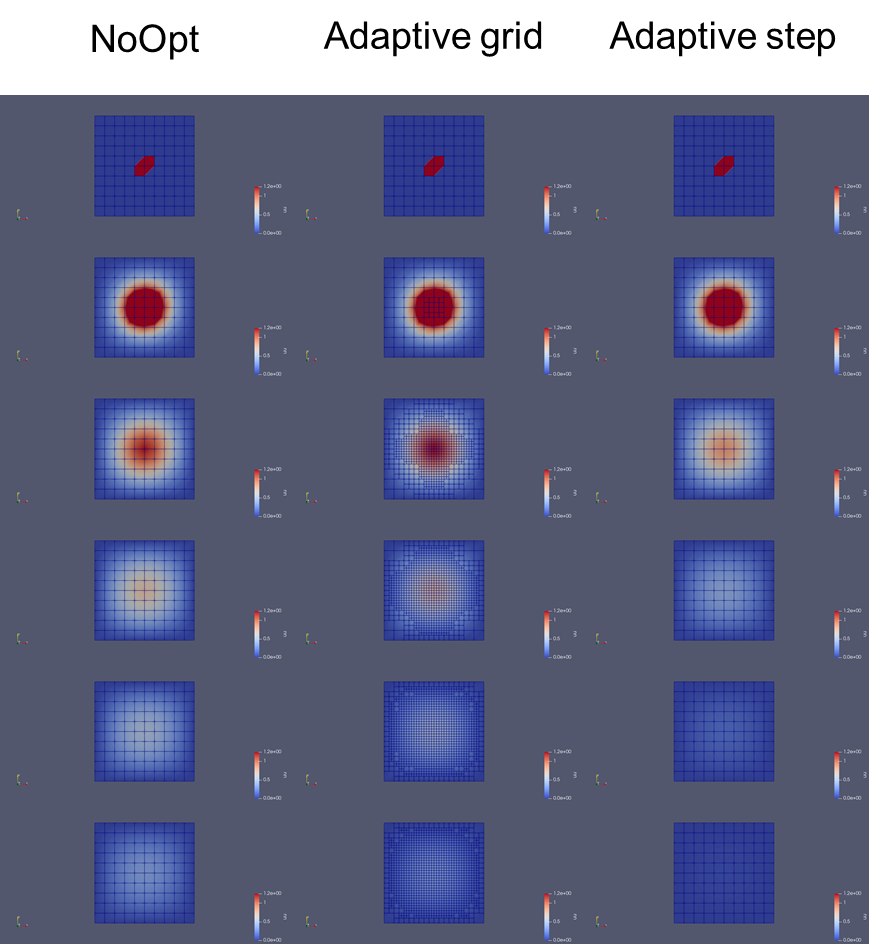
## Adaptive grid

The example ex08\_adaptive\_grid.i demonstrates the implementation of the adaptive grid. For demonstration purposes we apply the scheme to the pulse from the example ex06\_ic\_pulse.i in section 4.6.

A snippet of the Executioner section is shown below:



The evolution of the pulse is shown in **Figure 8**.



**Figure 8.** Evolution of the pulse during the course of the simulation. (left) original grid; (middle) adaptive grid; (right) original grid with adaptive step.

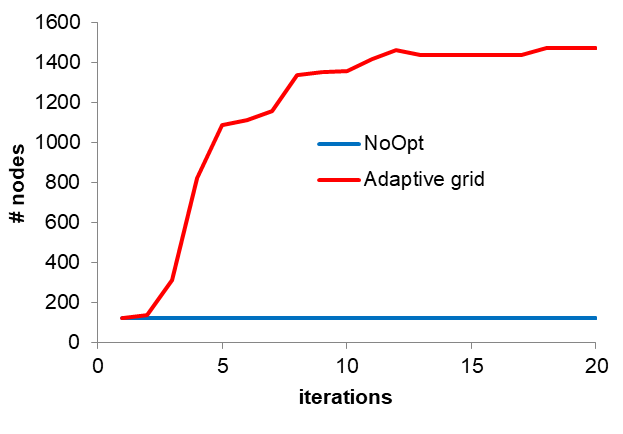
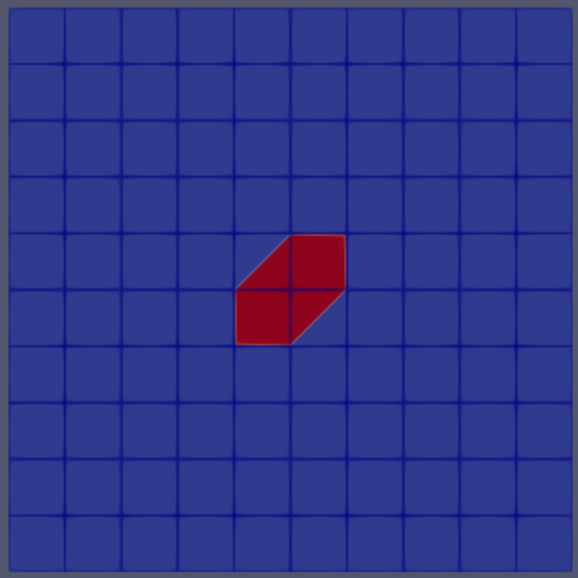
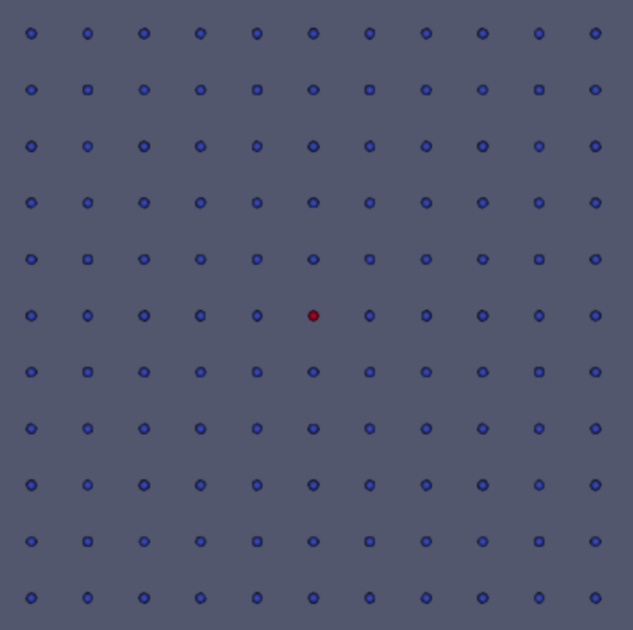


Figure . Evolution of the number of nodes for fixed (blue) and adaptive (red) grids.

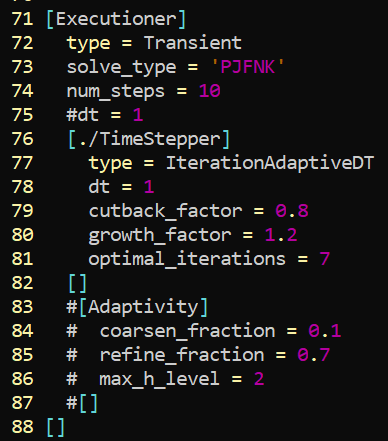
**Note:**

* The irregular shape at *t* = 0 is due to an artifact of Paraview; e.g., compare *Surface* and *Point Gaussian* options.  
   

## Adaptive step

The example ex09\_adaptive\_step.i demonstrates the implementation of the adaptive step. For demonstration purposes we apply the scheme to the pulse from the example ex06\_ic\_pulse.i in section 4.6.

A snippet of the Executioner section is shown below:



The evolution of the step size is shown in Figure 10and that of the solution (pulse) in **Figure 8** (right). We note that the evolution is much faster since the time step increases during the course of the simulation.

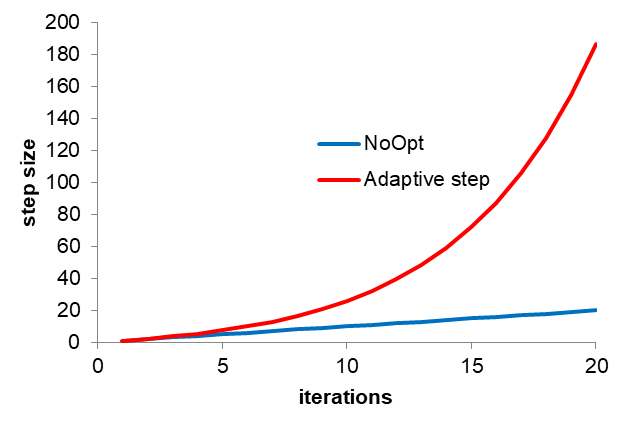
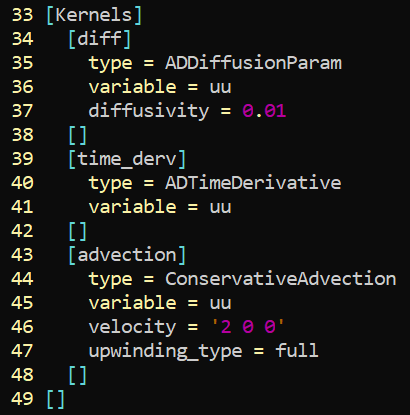


Figure . Evolution of step size during the course of the simulation.

## Diffusion-ConservativeAdvection

The example ex10\_diffusion\_advection\_pulse\_per.i solves the diffusion-advection equation with the same initial condition in the example ex06\_ic\_pulse.i.

A snippet of the kernel section is shown below:



In this case, the diffusivity has been set to 0.01 and the velocity field to **v** = (2, 0, 0).

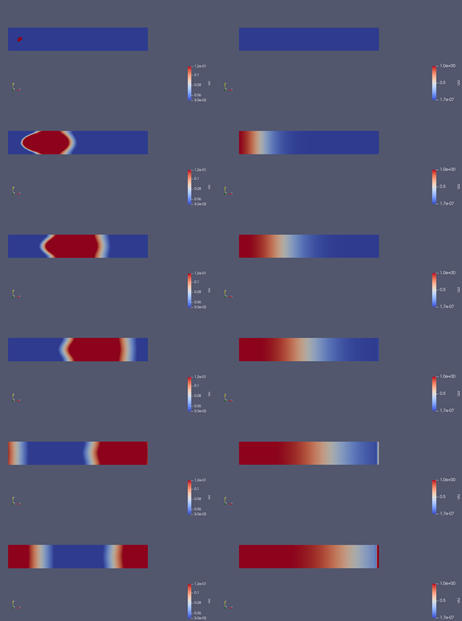


Figure . (left) Evolution of the advected initial pulse at steps 0, 10, 20, 30, 40 and 50. (right) Simulation with the InFlow boundary condition with subject to a velosity field **v** = (2, 0, 0).

The example ex10\_diffusion\_advection\_inflow.i solves the diffusion-advection equation with a fluid flowing within the simulation box from the left side with **v** = (2, 0, 0); boundary condition in file src/bcs/InflowBC.C.

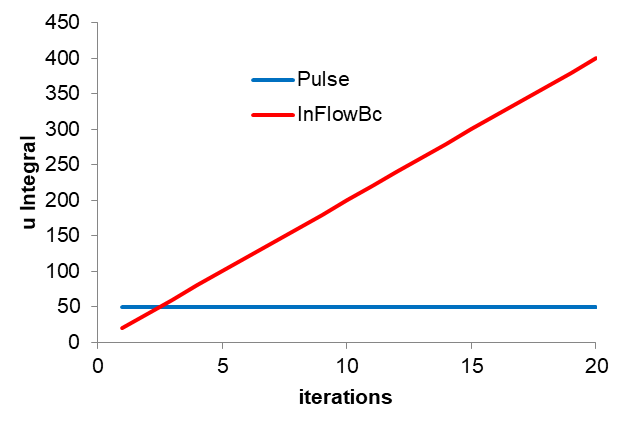


Figure . Solution integral during the course of the simulations.

# Paraview notes

TODO