Nektar++: Spectral/hp Element Framework

Tutorials

July 16, 2016

Department of Aeronautics, Imperial College London, UK
Scientific Computing and Imaging Institute, University of Utah, USA

Contents

Introduction **iv**

1 Advection Diffusion Reaction (ADR) Solver **1**
1.1 Background 2
1.2 Problem description 3
1.3 Pre-processing 3
1.4 Configuring the expansion bases and the conditions 6
1.5 Running the solver 9
1.6 Post-processing 10
1.7 Summary 11
1.8 Exercises left to the user 11

Introduction

The tutorials presented in the following aim to show the users of Nektar++ [**?**] how to utilise the various solvers available within the library, from the pre-processing steps to the post-processing, passing through the setup and run of the simulation.

Chapter 1

Advection Diffusion Reaction (ADR) Solver

Introduction

Welcome to the Advection Diffusion Reaction (ADR) Solver tutorial for the Nektar++ library. This tutorial is intended to show the main features of the ADR solver in a simple and user-friendly format. If you have not already downloaded and installed Nektar++, please do so by visiting nektar.info, where you can also find the User-Guide with the instructions to install the library. This tutorial requires:

- Nektar++ ADRSolver and pre- and post-processing tools,
- the open-source mesh generator Gmsh,
- the visualisation tool Paraview.

Goals

After the completion of this tutorial, the user will be familiar with:

- the generation of a simple mesh in Gmsh and its conversion into a Nektar++-compatible format;
- the visualisation of the Jacobian distribution across the mesh in Paraview;
- the setup of the initial and boundary conditions, the parameters and the solver settings;
- running a simulation with the ADR solver; and
- the post-processing of the data and the visualisation of the results in Paraview.

Resources

The files necessary to run this tutorial are included in ??. Specifically, two files with extension `.xml` are needed as input for Nektar++:

- a file containing the mesh: `ADR_mesh.xml`; and
- a configuration file with the simulation settings `ADR_conditions_aligned.xml`.

In the same folder it is possible to find also the following additional material:

- a Gmsh file to generate the mesh, `ADR_mesh.geo`;
- a .msh file containing the mesh in Gmsh format, `ADR_mesh.msh`;
- a .xml file containing the mesh in Nektar++ format without the edges aligned for the periodic boundary conditions (see section 1.3), `ADR_mesh.xml`;
- the 11 output .chk binary files generated by running the simulation, `ADR_mesh_aligned_i.chk`;
- the 11 output files converted in a Paraview readable format (.vtu), `ADR_mesh_aligned_i.vtu`;
- the output .fld binary file generated at the last time-step of the simulation, `ADR_mesh_aligned.fld`; and
- a bash script to automatically convert the 11 .chk files into .vtu files, `convert.sh`.

Now that you are ready, let's start!

1.1 Background

The ADR solver can solve various problems, including the unsteady advection, unsteady diffusion, unsteady advection diffusion equation, etc. For a more detailed description of this solver, please refer to the User-Guide.

In this tutorial we focus on the unsteady advection equation

$$\frac{\partial u}{\partial t} + \mathbf{V} \cdot \nabla u = 0, \tag{1.1}$$

where u is the independent variable and $\mathbf{V} = [\mathrm{V}_x\ \mathrm{V}_y\ \mathrm{V}_z]$ is the advection velocity. The unsteady advection equation can be solved in one, two and three spatial dimensions. We will here consider a two-dimensional problem, so that $\mathbf{V} = [\mathrm{V}_x\ \mathrm{V}_y]$.

1.2 Problem description

The problem we want to run consists of a given initial condition (which depends on x and y) travelling in the x-direction at a constant advection velocity. To model this problem we create a computational domain also referred to as mesh or grid (see section 1.3) on which we apply the following two-dimensional function as initial condition and periodic as well as time-dependent Dirichlet boundary conditions at the mesh boundaries

$$\begin{aligned}
&\frac{\partial u}{\partial t} + \mathrm{V}_x \frac{\partial u}{\partial x} + \mathrm{V}_y \frac{\partial u}{\partial y} = 0, \\
&u(x, y;\ t = 0) = \sin(\kappa x)\cos(\kappa y), \\
&u(x_b = [-1, 1], y_b;\ t) = \text{periodic}, \\
&u(x_b, y_b = [-1, 1];\ t) = \sin(\kappa(x - \mathrm{V}_x\, t))\cos(\kappa(y - \mathrm{V}_y\, t)),
\end{aligned} \tag{1.2}$$

where x_b and y_b represent the boundaries of the computational domain (see section 1.4), $\mathrm{V}_x = 2, \mathrm{V}_y = 0$ and $\kappa = 2\pi$.

We successively setup the other parameters of the problem, such as the time-step, the time-integration scheme, the I/O configuration, etc. (see section 1.4). We finally run the solver (see section 1.5) and post-process the data in order to visualise the results (see section 1.6).

1.3 Pre-processing

The pre-processing step consists in generating the mesh in a Nektar++ compatible format. For doing so we can first use the open-source mesh-generator Gmsh to first create the geometry, that in our case is a square and successively the mesh. The mesh format provided by Gmsh shown in Fig. (1.1) - i.e. .msh extension - is not compatible with the Nektar++ solvers and, therefore, it needs to be converted. To do so, we need

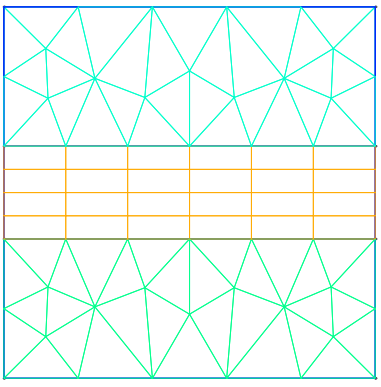

Figure 1.1 Mesh generated by Gmsh.

to run the pre-processing routine called `MeshConvert` within Nektar++. This routine requires two line arguments, the mesh file generated by Gmsh, `ADR_mesh.msh`, and the name of the Nektar++-compatible mesh file that `MeshConvert` will generate, for instance `ADR_mesh.xml`. The command line for this step is

```
nektar++/builds/utilities/MeshConvert/MeshConvert \
    ADR_mesh.msh  ADR_mesh.xml
```

The generated .xml mesh file is reported below. It contains 5 tags encapsulated within the `GEOMETRY` tag, which describes the mesh. The first tag, `VERTEX`, contains the spatial coordinates of the vertices of the various elements of the mesh. The second tag, `EDGE` contains the lines connecting the vertices. The third tag, `ELEMENT`, defines the elements (note that in this case we have both triangular - e.g. `<T ID="0">` - as well as quadrilateral - e.g. `<Q ID="85">` - elements). The fourth tag, `COMPOSITE`, is constituted by the physical regions of the mesh called **composite**, where the composites formed by elements represent the solution sub-domains - i.e. the mesh sub-domains where we want to solve the linear advection problem (note that we will use these composites to define expansion bases on each sub-domain in section 1.4) - while the composites formed by edges are the boundaries of the domain where we need to apply suitable boundary conditions (note that we will use these composites to specify the boundary conditions in section 1.4). Finally, the fifth tag, `DOMAIN`, formally specifies the overall solution domain as the union of the three composites forming the three solution subdomains (note that the specification of three subdomain - i.e. composites - in this case is necessary since they are constituted by different element shapes). For additional details on the `GEOMETRY` tag refer to the User-Guide.

```
<?xml version="1.0" encoding="utf-8" ?>
<NEKTAR>
    <GEOMETRY DIM="2" SPACE="2">
        <VERTEX>
            <V ID="0">2.00000000e-01 -1.00000000e+00 0.00000000e+00</V>
            <V ID="1">5.09667784e-01 -6.15240515e-01 0.00000000e+00</V>
            ...
            <V ID="68">-1.00000000e+00 1.25000000e-01 0.00000000e+00</V>
        </VERTEX>
        <EDGE>
            <E ID="0"> 0 1 </E>
            <E ID="1"> 1 2 </E>
            ...
            <E ID="153"> 40 68 </E>
        </EDGE>
        <ELEMENT>
            <T ID="0"> 0 1 2 </T>
            <T ID="1"> 3 4 5 </T>
            ...
            <Q ID="85"> 146 93 153 151 </Q>
       </ELEMENT>
        <COMPOSITE>
            <C ID="1"> T[0-30] </C>
            <C ID="2"> Q[62-85] </C>
```

```
25            <C ID="3"> T[31-61] </C>
26            <C ID="100"> E[46,12,20,10,45] </C>
27            <C ID="200"> E[50,32,108,111,114,117,87,103] </C>
28            <C ID="300"> E[100,64,74,66,99] </C>
29            <C ID="400"> E[49,33,148,150,152-153,86,104] </C>
30        </COMPOSITE>
31        <DOMAIN> C[1,2,3] </DOMAIN>
32    </GEOMETRY>
33 </NEKTAR>
```

After having generated the mesh file in a Nektar++-compatible format, `ADR_mesh.xml`, we can visualise the Jacobian distribution across the mesh in order to evaluate its quality. This step can be done by using the following Nektar++ built-in post-processing routine:

```
nektar++/builds/utilities/XmlToVtk -j ADR_mesh.xml
```

where the optional command -j calculate the Jacobian distribution for each element of the mesh. This will produce a `ADR_mesh.vtu` file which can be directly read by the open-source visualisation tool called Paraview. In Fig. 1.2 we show the Jacobian distribution for the mesh considered in this tutorial, `ADR_mesh.xml`. Before configuring

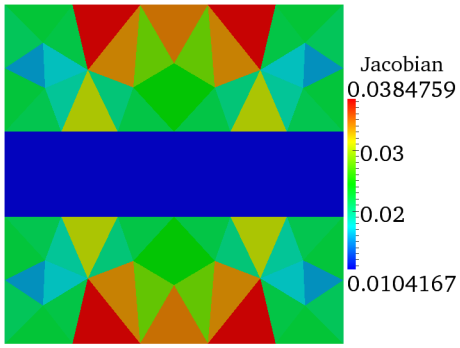

Figure 1.2 Jacobian distribution.

the input files, if we want to use periodic boundary conditions, we need to make sure that the edges of the two periodic boundaries (i.e. $x_b = [-1, 1], y_b$) are properly aligned. Gmsh and the `MeshConvert` routine within Nektar++ does not guarantee proper alignment. However, `MeshConvert` provides a module, called `peralign`, that enforces the reordering of pair of edges (for more details refer to the User-Guide. We can apply this by using the following command:

```
nektar++/builds/utilities/MeshConvert/MeshConvert \
-m peralign:surf1=200:surf2=400:dir=x ADR_mesh.xml ADR_mesh_aligned.xml
```

where `-m peralign` is selecting the module for aligning the edges which are specified by `surf1` and `surf2` (their IDs in this case are 200 and 400) and `dir` is the direction to which the two periodic edges are perpendicular (in this case x).

After having typed the last command, we have a mesh, `ADR_mesh_aligned.xml`, which is fully compatible with Nektar++ and which allows us applying periodic boundary conditions without encountering errors.

We can therefore now configure the conditions: initial conditions, boundary conditions, parameters and solver settings.

1.4 Configuring the expansion bases and the conditions

To set the various parameters, the solver settings and the initial and boundary conditions needed, we create a new file called `ADR_conditions.xml`, which can be found within the resources provided for this tutorial. This new file contains the `CONDITIONS` tag where we can specify the parameters of the simulations, the solver settings, the initial conditions, the boundary conditions and the exact solution and contains the `EXPANSIONS` tag where we can specify the polynomial order to be used inside each element of the mesh, the type of expansion bases and the type of points.

We begin to describe the `ADR_conditions.xml` file from the `CONDITIONS` tag, and in particular from the boundary conditions, initial conditions and exact solution sections:

```
<CONDITIONS>
    ...
    ...
    ...
    <VARIABLES>
        <V ID="0"> u </V>
    </VARIABLES>

    <BOUNDARYREGIONS>
        <B ID="0"> C[100] </B>
        <B ID="1"> C[200] </B>
        <B ID="2"> C[300] </B>
        <B ID="3"> C[400] </B>
    </BOUNDARYREGIONS>

    <BOUNDARYCONDITIONS>
        <REGION REF="0">
            <D VAR="u" USERDEFINEDTYPE="TimeDependent"
            VALUE="sin(k*(x-advx*t))*cos(k*(y-advy*t))" />
        </REGION>
        <REGION REF="1">
            <P VAR="u" VALUE="[3]" />
        </REGION>
        <REGION REF="2">
            <D VAR="u" USERDEFINEDTYPE="TimeDependent"
            VALUE="sin(k*(x-advx*t))*cos(k*(y-advy*t))" />
        </REGION>
```

```
        <REGION REF="3">
            <P VAR="u" VALUE="[1]" />
        </REGION>
    </BOUNDARYCONDITIONS>

    <FUNCTION NAME="InitialConditions">
        <E VAR="u" VALUE="sin(k*x)*cos(k*y)" />
    </FUNCTION>

    <FUNCTION NAME="AdvectionVelocity">
        <E VAR="Vx" VALUE="advx" />
        <E VAR="Vy" VALUE="advy" />
    </FUNCTION>

    <FUNCTION NAME="ExactSolution">
        <E VAR="u" VALUE="sin(k*(x-advx*t))*cos(k*(y-advy*t))" />
    </FUNCTION>
</CONDITIONS>
```

In the above piece of `.xml`, we first need to specify the non-optional tag called `VARIABLES` that sets the solution variable (in this case u).

The second tag that needs to be specified is `BOUNDARYREGIONS` through which the user can specify the regions where to apply the boundary conditions. For instance, `<B ID="0"> C[100] </B>` indicates that composite 100 (which has been introduced in section 1.3) has a **boundary ID** equal to 0. This boundary ID is successively used to prescribe the boundary conditions.

The third tag is `BOUNDARYCONDITIONS` by which the boundary conditions are actually specified for each **boundary ID** specified in the `BOUNDARYREGIONS` tag. The syntax `<D VAR="u"` corresponds to a `D`irichlet boundary condition for the variable `u` (note that in this case we used the additional tag `USERDEFINEDTYPE="TimeDependent"` which is a special option when using time-dependent boundary conditions), while `<P VAR="u"` corresponds to `P`eriodic boundary conditions. For additional details on the various options possible in terms of boundary conditions refer to the User-Guide.

Finally, `<FUNCTION NAME="InitialConditions">` allows the specification of the initial conditions, `<FUNCTION NAME="AdvectionVelocity">` specifies the advection velocities in both the x- and y-direction (for this two-dimensional case) and is a non-optional parameters for the unsteady advection equation and `<FUNCTION NAME="ExactSolution">` permits us to provide the exact solution, against which the L_2 and L_∞ errors are computed.

After having configured the `VARIABLES` tag, the initial and boundary conditions, the advection velocity and the exact solution we can complete the tag `CONDITIONS` prescribing the parameters necessary (`PARAMETERS`)and the solver settings (`SOLVERINFO`):

```
<CONDITIONS>
    <PARAMETERS>
        <P> FinTime   = 1.0   </P>
        <P> TimeStep  = 0.001 </P>
```

```
        <P> NumSteps   = FinTime/TimeStep </P>
        <P> IO_CheckSteps = 100   </P>
        <P> IO_InfoSteps  = 100   </P>
        <P> advx    = 2.0   </P>
        <P> advy    = 0.0   </P>
        <P> k    = 2*PI   </P>
    </PARAMETERS>

    <SOLVERINFO>
        <I PROPERTY="EQTYPE"   VALUE="UnsteadyAdvection"   />
        <I PROPERTY="Projection"   VALUE="DisContinuous"   />
        <I PROPERTY="AdvectionType"   VALUE="WeakDG"   />
        <I PROPERTY="UpwindType"  VALUE="Upwind"   />
        <I PROPERTY="TimeIntegrationMethod"  VALUE="ClassicalRungeKutta4" />
    </SOLVERINFO>
    ...
    ...
    ...
```

In the `PARAMETERS` tag, `FinTime` is the final physical time of the simulation, `TimeStep` is the time-step, `NumSteps` is the number of steps, `IO_CheckSteps` is the step-interval when a output file is written, `IO_InfoSteps` is the step-interval when some information about the simulation are printed to the screen, `advx` and `advy` are the advection velocities V_x and V_y, respectively and `k` is the κ parameter. Note that `advx`, `advy` and `k` are used in the boundary and initial conditions tags as well as in the specification of the advection velocities.

In the `SOLVERINFO` tag, `EQTYPE` is the type of equation to be solved, `Projection` is the spatial projection operator to be used (which in this case is specified to be 'DisContinuous'), `AdvectionType` is the advection operator to be adopted (where the `VALUE` 'WeakDG' implies the use of a weak Discontinuous Galerkin technique), `UpwindType` is the numerical flux to be used at the element interfaces when a discontinuous projection is used, `TimeIntegrationMethod` allows selecting the time-integration scheme. For additional solver-setting options refer to the User-Guide.

Finally, we need to specify the expansion bases we want to use in each of the three composites or sub-domains (`COMPOSITE=".."`) introduced in section 1.3:

```
<EXPANSIONS>
    <E COMPOSITE="C[1]" NUMMODES="5" TYPE="MODIFIED" FIELDS="u" />
    <E COMPOSITE="C[2]" NUMMODES="5" TYPE="MODIFIED" FIELDS="u" />
    <E COMPOSITE="C[3]" NUMMODES="5" TYPE="MODIFIED" FIELDS="u" />
</EXPANSIONS>
```

In particular, for all the composites, `COMPOSITE="C[i]"` with i=1,2,3 we select identical basis functions and polynomial order, where `NUMMODES` is the number of coefficients we want to use for the basis functions (that is commonly equal to P+1 where P is the polynomial order of the basis functions), `TYPE` allows selecting the basis functions

`FIELDS` is the solution variable of our problem and `COMPOSITE` are the mesh regions created by Gmsh. For additional details on the `EXPANSIONS` tag refer to the User-Guide.

1.5 Running the solver

Now that we have the mesh file compatible with Nektar++ and periodic boundary conditions, `ADR_mesh_aligned.xml`, and we have completed the condition file, `ADR_conditions.xml`, we can run the solver by using the following command:

```
nektar++/builds/solvers/ADRSolver/ADRSolver \
    ADR_mesh_aligned.xml ADR_conditions.xml
```

Note that we have written the mesh in a separate file from the conditions. This is generally more efficient because it allows reopening just the condition file which is much smaller in size than the mesh file (especially for large problems). However, we could also have written both the mesh and the conditions in unique file and used the same command as above for running the solver (in this case with just one file instead of two as line argument).

As soon as the file finishes running, we should see the following screen output:

```
=======================================
               EquationType: UnsteadyAdvection
               Session Name: ADR_mesh_aligned
               Spatial Dim.: 2
         Max SEM Exp. Order: 5
             Expansion Dim.: 2
             Riemann Solver: Upwind
             Advection Type:
            Projection Type: Discontinuous Galerkin
                  Advection: explicit
                  Diffusion: explicit
                  Time Step: 0.001
               No. of Steps: 1000
        Checkpoints (steps): 100
           Integration Type: ClassicalRungeKutta4
=======================================
Initial Conditions:
  - Field u: sin(k*x)*cos(k*y)
Writing: "ADR_mesh_aligned_0.chk"
Steps: 100      Time: 0.1          CPU Time: 0.435392s
Writing: "ADR_mesh_aligned_1.chk"
Steps: 200      Time: 0.2          CPU Time: 0.430588s
Writing: "ADR_mesh_aligned_2.chk"
Steps: 300      Time: 0.3          CPU Time: 0.428503s
Writing: "ADR_mesh_aligned_3.chk"
Steps: 400      Time: 0.4          CPU Time: 0.428529s
Writing: "ADR_mesh_aligned_4.chk"
```

```
Steps: 500      Time: 0.5          CPU Time: 0.430142s
Writing: "ADR_mesh_aligned_5.chk"
Steps: 600      Time: 0.6          CPU Time: 0.429481s
Writing: "ADR_mesh_aligned_6.chk"
Steps: 700      Time: 0.7          CPU Time: 0.433232s
Writing: "ADR_mesh_aligned_7.chk"
Steps: 800      Time: 0.8          CPU Time: 0.431088s
Writing: "ADR_mesh_aligned_8.chk"
Steps: 900      Time: 0.9          CPU Time: 0.427919s
Writing: "ADR_mesh_aligned_9.chk"
Steps: 1000     Time: 1            CPU Time: 0.436098s
Writing: "ADR_mesh_aligned_10.chk"
Time-integration  : 4.31097s
Writing: "ADR_mesh_aligned.fld"
-------------------------------------------
Total Computation Time = 4s
-------------------------------------------
L 2 error (variable u) : 0.00863475
L inf error (variable u) : 0.0390366
```

where the L2 and L inf errors are evaluated against the `<FUNCTION NAME="ExactSolution">` provided in the `ADR_conditions.xml` file. To have a more detailed view on the solver settings and parameters used, it is possible to use the `-v` option (which stands for verbose) as follows:

```
nektar++/builds/solvers/ADRSolver/ADRSolver -v \
    ADR_mesh_aligned.xml ADR_conditions.xml
```

The simulation has now produced 11 `.chk` binary files and a final `.fld` binary file (which in this case is identical to the tenth `.chk` file). These binary files contain the result of the simulation every 100 time-steps. This output interval has been chosen through the parameter `IO_CheckSteps` in `ADR_conditions.xml`, which was set equal to 100. Also, it is possible to note that every 100 time-steps the solver outputs the physical time of the simulation and the CPU time required for doing 100 time-steps. The interval of 100 time-steps is decided through the parameter `IO_InfoSteps`, which was also equal to 100.

1.6 Post-processing

Now that the simulation has been completed, we need to post-process the file in order to visualise the results. In order to do so, we can use the built-in post-processing routines within Nektar++. In particular, we can use the following command

```
nektar++/builds/utilities/FieldConvert/FieldConvert \
    ADR_mesh_aligned.xml ADR_conditions.xml \
    ADR_mesh_aligned_0.chk ADR_mesh_aligned_0.vtu
```

which generates a `.vtu` file that is a readable format for the open-source package Paraview. Note that we typically have to specify both the mesh `.xml` file and the condition `.xml` file. We can now open the `.vtu` file just generated (which corresponds to the initial condition, being the number '0' `.chk` file) and visualise it with Paraview. This produces the image in Fig. (1.3). It is possible to use the same post-processing

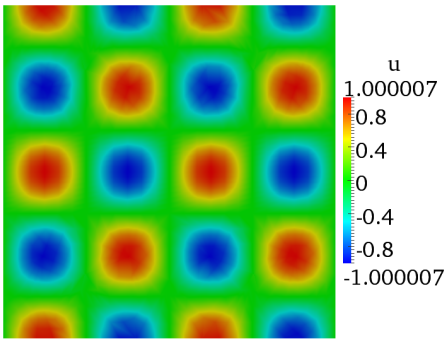

Figure 1.3 Initial solution

command for visualising the other `.chk`, thus monitoring the evolution of the simulation in time.

1.7 Summary

The user should be now familiar with the following topics:

- Generate a simple mesh in Gmsh and convert it in a Nektar++-compatible format;
- Visualise the Jacobian distribution across the mesh in Paraview;
- Setup the initial and boundary conditions, the parameters and the solver settings;
- Run the ADR solver; and
- Post-process the data in order to visualise results in Paraview.

1.8 Exercises left to the user

1. Increase the polynomial order and plot the L_2 error vs. the polynomial order in a semilogarithmic scale.
2. Change the projection operator for a fixed polynomial order and look at the error.
3. Increase the time-step for a fixed polynomial order and look at the error.
4. If the solver is compiled with the MPI option, then try running the case in parallel with `mpirun -np 2`.

5. Change the Projection Operator to Continuous to see the same problem running with a CG solver.

6. Change the solver type to AdvectionDiffusion and CG to change the problem type. You also need to update the AdvectionType to NonConservative. Noting that solution would need to be updated to . . . ?

Tip

To check the additional settings and parameters that can be used for this solver, check the folder: nektar++/solvers/ADRSolver/Tests/ where you can find several tests associated to the ADR solver.

Bibliography

[1] CD Cantwell, D Moxey, A Comerford, A Bolis, G Rocco, G Mengaldo, D De Grazia, S Yakovlev, J-E Lombard, D Ekelschot, et al. Nektar++: An open-source spectral/hp element framework. *Computer Physics Communications*, 192:205–219, 2015.