DGANN solver

Created by: Deep Ray (Rice University, Houston), Niccolò Discacciati (EPFL, Lausanne)

Code available at:

`https://github.com/nickdisca/DGANN_AV/` `https://bitbucket.org/deepray/dgann/src/master/`

Date: 26 August, 2019

DGANN is a RKDG-solver written in MATLAB, which is also capable of using artificial neural networks trained to serve as a troubled-cell detector and/or artificial viscosity estimator. The main source code for the RKDG solver is based on the solvers available with the following texts:

1. Nodal Disontinuous Galerkin methods, by Jan S. Hesthaven and Tim Warburton.
2. Numerical Methods for Conservation Laws: From Analysis to Algorithms, by Jan S. Hesthaven.

Details about the design of the Multilayer Perceptron (MLP) troubled-cell indicator for 1D problems have been published in the paper "An artificial neural network as a troubled-cell indicator", while the extension to 2D problems is described here. Details about the MLP viscosity estimator can be found here.

NOTE: If the math symbols do not display properly in README_AV.md, have a look at README_AV.html or README_AV.pdf instead.

NOTE: As of today, the two MLPs have not been tested together. We used either limiting or artificial viscosity. Even though no issues should arise when combining the models, we advice the interested user to take particular care in case (s)he whants to do so.

Table of contents

Running the code

After cloning the git repository, execute **mypath.m** from the parent directory in MATLAB. This will set all the neccesary paths to use the solver. The various test cases need to be run from the **Examples** directory or its sub-directories.

Scripts for 1D problems

Currently, the 1D solver supports linear advection, Burgers' equation, the shallow water equations and the compressible Euler equations.

Scalar 1D

The basic structure of the example script is as follows.

```
CleanUp1D;

clc
clear all
close all


model = 'Advection';
test_name = 'Sine';
u_IC =@(x) sin(10*pi*x);

bnd_l     = 0.0;
bnd_r     = 1.0;
mesh_pert = 0.0;
bc_cond   = {'P',0.0,'P',0.0};
FinalTime = 0.2;
CFL       = 0.2;
K         = 100;
N         = 4;
RK        = 'LS54';

Indicator = 'MINMOD';
nn_model  = 'MLP_v1';
Limiter   = 'MINMOD';

Visc_model    = 'NONE';
nn_visc_model = 'MLP_visc';

plot_iter  = 200;
save_iter  = 1;
save_soln  = true;
save_ind   = true;
save_visc  = true;
save_plot  = true;
ref_avail  = true;
ref_fname  = 'ref_soln.dat';
rk_comb    = false;
var_ran    = [-1.2,1.5];

% Call code driver
ScalarDriver1D;
```

- `CleanUp1D` removes temporary file paths added, which is especially important if the previous run of the script terminated prematurely. This must be the first line of every script. DO NOT REMOVE IT!
- The `model` flag sets the type of scalar model which is being solved. The following scalar models are currently available:
  - `'Advection'` : Linear advection equation with the advection speed set to 1.
  - `'Burgers'` : Burgers equation with the flux $u^2/2$.
  - `'BuckLev'` : Buckley-Leverett equation with flux contant set to 0.5.
- `test_name` is used to declare the name of the test. This is used for creating various save files.
- `u_IC` is used to set the initial condition for the problem.
- `bnd_l` and `bnd_r` define the left and right domain boundaries, which is descritized using `K` number of elements. The degree of the polynomial in each element/cell is set using `N`.
- `mesh_pert` is used to randomly perturb the interior cell-interfaces using the following algorithm.

$$x_{i+\frac{1}{2}} \to x_{i+\frac{1}{2}} + \text{mesh\_pert}\; h\; \omega_{i+\frac{1}{2}}, \qquad \omega_{i+\frac{1}{2}} \in \mathcal{U}[-0.5, 0.5], \qquad i = 1, \ldots, K-1$$

  where h is the mesh size of the initial uniform grid.

- `bc_cond` is used to set the left and right boundary conditions. It is a cell of the form `{LEFT_BC_TYPE, LEFT_BC_VAL, RIGHT_BC_TYPE,RIGHT_BC_VAL}`. The `BC_TYPES` can be set as:
  - `'P'` : Periodic boundary conditions. In this case, both boundary type must be set to `'P'`.
  - `'N'` : Neumann boundary condition.
  - `'D'` : Dirichlet boundary condition, with the imposed Dirichlet value given by `BC_VAL`. Note that `BC_VAL` is ignored if `BC_TYPE` is not set to `'D'`.
- The final simulation time is set using `FinalTime`, while the time step is chosen using `CFL`.
- `K` is the number of elements/cells in the mesh.
- `N` sets the order of the basis.
- `RK` sets the time integration scheme. The current implementation supports the five-stage fourth-order low-storage Runge-Kutta scheme `'LS54'` and the third order

strong stability preserving Runge-Kutta `'SSP3'`.

- The troubled-cell indicator is set using `Indicator`. The following options are currently available:
  - `'NONE'`: No cells are flagged.
  - `'ALL'`: All cells are flagged.
  - `'MINMOD'`: Uses the basic minmod limiter.
  - `'TVB'`: Uses the modified minmod-type TVB limiter. If this is chosen, then one also needs to set the variable `TVBM` to a positive number. Note that if this constant is set to 0, then the limiter reduces to the usual minmod limiter.
  - `'NN'`: Uses the trained neural network. The network name is set using the variable `nn_model`. The available networks are described below in the section **Using the MLP troubled-cell indicator**.
- The limiter usd to reconstruct the solution in each troubled-cell,is set using `Limiter`. The following options are currently available:
  - `'NONE'`: No limiting is applied.
  - `'MINMOD'`: MUSCL reconstruction using minmod limiter.
- The artificial viscosity model is set using `Visc_model`. The following options are currently available:
  - `'NONE'`: No artificial viscosity is added.
  - `'EV'`: Uses the entropy viscosity model. If this is chosen, two nonnegative parameters `c_E` and `c_max` have to be specified.
  - `'MDH'`: Uses the highest modal decay model. If this is chosen, three nonnegative parameters `c_A`, `c_k`, and `c_max` have to be specified.
  - `'MDA'`: Uses the averaged modal decay model. If this is chosen, a nonnegative parameter `c_max` has to be specified.
  - `'NN'`: Uses the trained neural network. The network name is set using the variable `nn_visc_model`. The available networks are described below in the section **Using the MLP artificial viscosity estimator**.
- The flag `plot_iter` is used for visualization purposes. The solution plots are shown after every `plot_iter` number of iterations during the simulation.
- The flag `save_iter` controls the frequency with which data are saved to files. This applies to both the troubled cells and the artificial viscosity.
- If solution at the final time needs to be saved, the flag `save_soln` must be set to `true`. If this is not the case, set this flag to `false`. The solution files are save in the directory **OUTPUT**. The filename has the format: `<model>1D_<test_name>_P<N>_N<K>_IND_<Indicator>_LIM_<Limiter>_VISC_<Visc_model>.dat`. If mesh perturbation is used, then the filename will contain the tag `pert`. The data in the file has the following format:
  - Column 1: x-coordinates.
  - Column 2: solution value at corresponding x-coordinate
- If the time-history of the troubled-cells needs to be viewed/saved, the flag `save_ind` must be set to `true`. If this is not the case, set this flag to `false`. The time-history files are also save in the directory **OUTPUT**. The filename has the format: `<model>1D_<test_name>_P<N>_N<K>_IND_<Indicator>_LIM_<Limiter>_VISC_<Visc_model>_tcells.dat`. If mesh perturbation is used, then the filename includes the tag `pert`. The data in the file has the following format:
  - Each row of the file contains the time, followed by the mid-points of the cells flagged as troubled-cells at that time.
  - The first row corresponds to the cell flagged at time $t = 0$. This is essentially done for the initial condition.
  - Following the first row, the rows can be grouped in sets of size r, to list the cells flagged after each sub-stage of the r-stage time-integration scheme. Depending on the selected Runge-Kutta scheme, r will be equal to 3 or 5.
- If the time-history of the artificial viscosity needs to be viewed/saved, the flag `save_visc` must be set to `true`. If this is not the case, set this flag to `false`. The time-history files are also save in the directory **OUTPUT**. The filename has the format: `<model>1D_<test_name>_P<N>_N<K>_IND_<Indicator>_LIM_<Limiter>_VISC_<Visc_model>_visc.dat`. If mesh perturbation is used, then the filename will include the tag `pert`. The data in the file has the following format:
  - Each row of the file contains the time, followed by the value of the artificial viscosity in each x coordinate at the current time. Unlike the troubled-cell indicator, the viscosity is updated at the beginning of the RK sub-loop. Thus, the number of rows is equal to the number of timesteps.
- If `save_plot` is set to `true`, then the solution plots are generated and saved in **OUTPUT**. **NOTE**: This needs `save_soln`, `save_ind` and `save_visc` to be set as `true`.
- If a reference/exact solution data file is available, then set `ref_avail` to `true` and the (relative) file name of the referene solution in `ref_fname`.
- `var_ran` is used to set the ylim for the solution plot. This should be a array of size (1,2).
- The main driver script `ScalarDriver1D` is called once all the flags have been set.

back to table of contents

Euler 1D

The basic structure of the example script is as follows.

```
CleanUp1D;

clc
clear all
close all

Globals1D_DG;
Globals1D_MLP;

model     = 'Euler';
gas_const = 1.0;
gas_gamma = 1.4;
test_name = 'ShockEntropy';
rho_IC    =@(x) (x<-4)*3.857143 + (x>=-4).*(1 + 0.2*sin(5*x));
vel_IC    =@(x) (x<-4)*2.629369;
pre_IC    =@(x) (x<-4)*10.33333 + (x>=-4)*1.0;


bnd_l     = -5.0;
bnd_r     = 5.0;
mesh_pert = 0.0;
bc_cond   = {'D',3.857143,'N',0.0;
             'D',10.141852,'D',0.0;
             'D',39.166661,'N',0.0};  % For conserved variables
FinalTime = 1.8;
CFL       = 0.1;
K         = 200;
N         = 4;
RK        = 'LS54';

Indicator = 'NN';
ind_var   = 'prim';
nn_model  = 'MLP_v1';
Limiter   = 'MINMOD';
lim_var   = 'con';

nn_visc_model = 'MLP_visc';
Visc_model='NN';
visc_var='density';

plot_iter  = 100;
save_soln  = true;
save_ind   = true;
save_visc  = true;
save_plot  = true;
ref_avail  = true;
ref_fname  = 'ref_soln.dat';
rk_comb    = true;
var_ran    = [0,1.2; -0.2,1.5; 0,1.2];

% Call code driver
EulerDriver1D;
```

Most of the structure is similar to the Scalar 1D script. The differences are described below.

- The `model` needs to be set as `'Euler'`. This should not be changed.
- The gas constant and ratio of specific heats is set using `gas_const` and `gas_gamma`.
- The initial density, velocity and pressure are set using `rho_IC`, `vel_IC` and `pre_IC`.
- The `bc_cond` has the same format as earlier, although now it has three rows of parameters. The first row corresponds to density, the second corresponds to the momentum, and the third to energy. Note that the boundary condition are set for the conserved variables.
- For systems of conservation laws, there are various choices for the variables to be used for troubled-cell detection. For the Euler equations, `ind_var` can be set as (the troubled-cells flagged for each variable is pooled together):
    - `'density'`: Only the density is used
    - `'velocity'`: Only the velocity is used
    - `'pressure'`: Only the pressure is used
    - `'prim'`: The primitive variables i.e., density, velocity and pressure, are used.
    - `'con'`: The conserved variables i.e., density, momentum and energy, are used.
- As was the case with detection, there are several options for the variables which can be reconstructed. This is set using the flag `lim_var`, with the following options:
    - `'prim'`: The primitive variables i.e., density, velocity and pressure, are recontructed.
    - `'con'`: The conserved variables i.e., density, momentum and energy, are recontructed.
    - `'char_cell'`: The local characterictic variables are reconstructed. These are obtained cell-by-cell using the linearized transformation operators. More precisely, the

transformation matrix in each cell is evaluated using the cell-average value, following which the conserved variables are transformed to the characteristic variables in that cell. The same transformation is used to retrieve the conserved variables after limiting the characteristic variables.
    - `'char_stencil'`: The local characterictic variables obtained cell-by-cell can introduce spurious oscillations in the solution. One can also obtain the local characteristic variables, stencil-by stencil. More precisely, for a given reconstruction stencil of 3-cells, the transformation matrix is evaluated using the cell-average value of the central cell, following which the conserved variables are transformed to the characteristic variables in every cell of that stencil. The transformed variables are used to obtain the reconstructed characteristic variables in the central cell. Note that this approach can be 3 times more expensive than the `'char_cell'` approach.

- For systems of conservation laws, there are various choices to estimate the artificial viscosity. The current implementation of Euler equations estimates the dissipation using the representative variable set in `visc_var`. Only `'density'` is currently supported. After the viscosity estimation, the same value is applied to all equations.
- The solution filename has the format:
  `<model>1D_<test_name>_P<N>_N<K>_IND_<Indicator>_IVAR_<ind_var>_LIM_<Limiter>_LVAR_<lim_var>_VISC_<Visc_model>_VVAR_<visc_var>.dat`.
  The data in the file has the following format:
    - Column 1: x-coordinates
    - Column 2: the value of density, velocity and pressure (in that order) at corresponding x-coordinate.

- The troubled-cell time-history filename has the format:
  `<model>1D_<test_name>_P<N>_N<K>_IND_<Indicator>_IVAR_<ind_var>_LIM_<Limiter>_LVAR_<lim_var>_VISC_<Visc_model>_VVAR_<visc_var>_tcells`
- The artificial viscosity time-history filename has the format:
  `<model>1D_<test_name>_P<N>_N<K>_IND_<Indicator>_IVAR_<ind_var>_LIM_<Limiter>_LVAR_<lim_var>_VISC_<Visc_model>_VVAR_<visc_var>_visc.da`
  As the same viscosity is injected in all equations, it is saved only for the first one.
- `var_ran` is used to set the ylim for the solution plots, with the format
  `[rho_min,rho_max ; velocity_min, velocity_max ; pressure_min, pressure_max]`.
- The main driver script `EulerDriver1D` is called once all the flags have been set. The troubled-cells flagged for each variable is pooled together.

back to table of contents

Scripts for 2D problems

Currently, the 2D solver supports linear advection, Burgers' equation, the KPP equation, and the compressible Euler equations. For each problem, we need the following files:

- A main script (with the default name `Main.m`) to set the various problem parameters.
- The initial condition is defined using the script/function (with the default name `IC.m`).
- When physical boundary conditions are used, an additional script/function (**must** be named `BC.m`) needs to be created. This file is not needed when all boundary conditions are periodic.
- Finally, a mesh file is needed, which is currently generated using Gmsh. Each example is already provided with the Gmsh geometry file (with the extension `.geo`). To generate the mesh file (with the extension `.msh`), you could either use the Gmsh GUI or create the file non-interactively from the terminal as
  `$ gmsh -2 mymeshfile.geo`

Scalar 2D

The basic structure of the various scripts are as follows:

Main.m

```
% Remove NN diretory paths if they still exist
CleanUp2D;

close all
clear all
clc

model             = 'Advection';
AdvectionVelocity = [1,1]; % Used for linear advection only
test_name         = 'Smooth';
InitialCond       = @IC;
BC_cond           = {100001,'P'; 100002,'P'; 100003,'P'; 100004,'P'};


FinalTime         = 0.5;
CFL               = 0.3;
tstamps           = 2;
N                 = 1;
RK                = 'LS54';

% Set type of indicator
Indicator       = 'NONE';
Filter_const    = true;
nn_model        = 'MLP_v1';
Limiter         = 'NONE';

%Set viscosity model
nn_visc_model = 'MLP_visc';
Visc_model='EV'; c_E=1; c_max=0.25;

% Mesh file
msh_file        = 'square_trans.msh';

% Output flags
plot_iter  = 50;
show_plot  = true;
xran       = [0,1];
yran       = [0,1];
clines     = linspace(0,2,30);
save_soln  = true;

% Call main driver
ScalarDriver2D;
```

- `CleanUp2D` removes temporary file paths added, which is especially important if the previous run of the script terminated prematurely. This must be the first line of every script. DO NOT REMOVE IT!
- The `model` flag sets the type of scalar model which is being solved. The following scalar models are currently available:
  - `'Advection'` : Linear advection equation with the advection speed set set using `AdvectionVelocity`.
  - `'Burgers'` : Simple extension of the Burgers' equation to 2D, with the flux $u^2/2$ in each coordinate direction.
  - `'KPP'` : KPP problem in 2D with flux $(\sin(u), \cos(u))$.
- `test_name` is used to declare the name of the test. This is used for creating various save files.
- `InitialCond` is used to set the initial condition for the problem (see the description of IC.m below).
- `BC_cond` is used to specify the boundary conditions for each face of the domain. This must be a MATLAB cell array of size $NBfaces \times 2$, where $NBfaces$ is the number of boundary curves specified in the mesh geometry file (with file-extension `.geo`). The first element of each row must be the physical face tag of a boundary curve in the geometry file, while the second element must be one of the following flags
  - `'P'` : Periodic boundary conditions. Note there must be an even number of periodic boundary faces.
  - `'D'` : Dirichlet boundary condition. The actual Dirichlet conditions are specified in BC.m
  - `'Sym'` : Symmteric boundary conditions.
    Boundary conditions require the creation of ghost cells, with the value of the solution in these cells depending on the type of boundary condition.
- The final simulation time is set using `FinalTime`, while the time step is chosen using a constant `CFL` or a setting a step `fixed_dt`. NOTE: Either `CFL` can be mentioned or the fixed time-step `fixed_dt`, but not both at the same time.
- `tstamps` is used to specify the number of uniform time-instances (exluding the initial time) at which the solution files are saved. This must be a positive integer. For instance if `tstamps=2` and `FinalTime = 2`, then the solution evaluated at times closest (less than half the local time-step) to $t = 0, 1, 2$ are saved. NOTE: It might happen that pen-ultimate time instance in the simulation is very close to `FinalTime`, with the difference being much smaller than the pen-ultimate time-step. In this case, the solver saves the solution at this pen-ultimate as the solution representing the final solution. Thus, we also save the solution at the actual `FinalTime`, leading to a total of `tstamps` +2 save points in time. This becomes crucial when comparing the troubled-cells flagged at the final time-step, since the number of cell flagged is sensitive to the size of the time-step taken.
- `N` sets the order of the basis.

- `RK` sets the time integration scheme. The current implementation supports the five-stage fourth-order low-storage Runge-Kutta scheme `'LS54'` and the third order strong stability preserving Runge-Kutta `'SSP3'`.
- The troubled-cell indicator is set using `Indicator`. The following options are currently available:
  - `'NONE'` : No cells are flagged.
  - `'ALL'` : All cells are flagged.
  - `'TVB'` : Uses the 2D minmod-type TVB limiter. If this is chosen, then one also needs to set the variables `TVBM` and `TVBnu` as positive numbers. See the paper for details.
  - `'NN'` : Uses the trained neural network. The network name is set using the variable `nn_model`. The available networks are described below in the section **Using the MLP troubled-cell indicator**.
- Set `Filter_const` to `true`, if you want to avoid flagging almost constant cells. This is especially important for speeding up the performance with the network. See the paper for details.
- The limiter usd to reconstruct the solution in each troubled-cell,is set using `Limiter`. The following options are currently available:
  - `'NONE'` : No limiting is applied.
  - `'BJES'` : Barth-Jespersen limiter.
- The artificial viscosity model is set using `Visc_model`. The following options are currently available:
  - `'NONE'` : No artificial viscosity is added.
  - `'EV'` : Uses the entropy viscosity model. If this is chosen, two nonnegative parameters `c_E` and `c_max` have to be specified.
  - `'MDH'` : Uses the highest modal decay model. If this is chosen, three nonnegative parameters `c_A`, `c_k`, and `c_max` have to be specified.
  - `'MDA'` : Uses the averaged modal decay model. If this is chosen, a nonnegative parameter `c_max` has to be specified.
  - `'NN'` : Uses the trained neural network. The network name is set using the variable `nn_visc_model`. The available networks are described below in the section **Using the MLP artificial viscosity estimator**.
- The flag `plot_iter` is used for visualization/verbose purposes. The solution plots are shown after every `plot_iter` number of iterations during the simulation.
- Set `show_plot` to `true` if you want solution plots to be generated during the simulation.
- `xran` and `yran` are used to crop the x and y axes of the solution plots (if `show_plot` is set to `true`). This is particulalry helpful when an extended domain is used to impose artificial boundary conditions, for instance when solving 2D Riemann problem.
- `clines` is used to set the contour lines for the solution contour plots (if `show_plot` is set to `true`).
- The solution variables are saved in the **OUTPUT** directory if `save_soln` is set to `true`. The filename has the format: `<model>2D_<test_name>_P<N>_IND_<Indicator>_LIM_<Limiter>_VISC_<Visc_model>_DATA.mat`. If `Filter_const` is set to `true`, then `ConstFilt` also appears in the filename. The following MATLAB variables are saved:
  - The `x` and `y` coordinates of all degrees of freedom in the mesh.
  - Inverse of the Vandermonde matrix `invV`, the Jacobian of the geometric mapping `J`, the mass matrix `Mass` in the reference element.
  - `Save_times` is the array of exact time-instances at which the solution is saved. This is of length `tstamps` +2.
  - The solutions evaluated at the time-instances listed in `Save_times` is stored in the MATLAB-cell `Q_save`.
  - The troubled-cells flagged in the mesh at each time-instance listed in `Save_times` is stored in the MATLAB-cell `ind_save`.
  - The artificial viscosity at each time-instance listed in `Save_times` is stored in the MATLAB-cell `visc_save`.
  - `ptc_hist` stores the time-history of the percentage of the total number mesh cells flaged as troubled-cells.
  - `maxvisc_hist` stores the time-history of the maximum value of the artificial viscosity.
  - `t_hist` stores all the time-instances attained during the simulation.
  - `sim_time` stores the full simulation time (excludes time taken to generate mesh data structures).
- The main driver script `ScalarDriver2D` is called once all the flags have been set.

IC.m

The initial condition function must take as input the arrays x and y, where each of these is of the shape $m \times n$. The output should be a array of dimension $m \times n \times 1$. Almost always, m will denote the number of DOFs per cell in the mesh, while n will be the number of cells. We give an example of this function below:

```
function Q = IC(x, y)

    Q(:,:,1) = sin(4*pi*x).*cos(4*pi*y);

return;
```

BC.m

When using non-periodic boundary conditions, a function script called BC.m is also needed. An example function is as follows:

```
function uG = BC(ckey,time)

    u1 = 1.0;  u2 = -1.0, c=3.0;

    if(ckey == 101 || ckey == 103)
        uG   =@(x,y) u1*ones(size(x));
    elseif(ckey == 102)
        uG =@(x,y) u2*ones(size(x));
    elseif(ckey == 104)
        rhoG =@(x,y) u1*(y > x+c*time) + u1*(y < x+c*time);
    end

return;
```

where the function takes in as input a boundary physical tag `ckey` and the curent simulation-time. The output is a function of x and y (and perhaps implicitly of time).

back to table of contents

Euler 2D

The basic structure of the various scripts are as follows:

Main.m

```
% Remove NN diretory paths if they still exist
CleanUp2D;

close all
clear all
clc

%Model
model          = 'Euler';
gas_const      = 1.0;
gas_gamma      = 1.4;
test_name      = 'SV';
InitialCond    = @IC;
BC_cond        = {100001,'P'; 100002,'P'; 100003,'P'; 100004,'P'};

FinalTime      = 0.8;
CFL            = 0.8;
tstamps        = 1;
N              = 4;
RK               = 'LS54';

% Mesh file
msh_file      = 'nonuniform_m1p3_H002.msh';

% Set type of indicator
Indicator     = 'NONE';
ind_var       = 'con';
nn_model      = 'MLP_v1';
Limiter       = 'NONE';
lim_var       = 'con';
Filter_const  = true;

%Set viscosity model
nn_visc_model = 'MLP_visc';
Visc_model='EV'; c_E=1; c_max=0.25;
visc_var='density';


plot_iter  = 50;
show_plot  = true;
xran       = [0,2];
yran       = [0,1];
plot_var   = {'density'};
clines     = {linspace(0.85,1.35,30)};
save_soln  = true;

% Call main driver
EulerDriver2D;
```

Most of the structure is similar to the Scalar 2D script. The differences are described below. Note that most of the differences coincide with the 1D case.

- The `model` flag must be set to `Euler`.
- The gas constant and ratio of specific heats is set using `gas_const` and `gas_gamma`.
- The following flags
  - `'P'` : Periodic boundary conditions.
  - `'D'` : Dirichlet boundary condition.
  - `'S'` : Slip boundary conditions.
  - `'I'` : Inflow boundary.
  - `'O'` : Outflow boundary
- The troubled-cells are detected using the variables mentioned in `ind_var`. The following options are available :
  - `'density'`
  - `'pressure'`
  - `'velocity'` : Both components of velocity are use.
  - `'prim'` : The primitive variables are used.
  - `'con'` : The conserved variables are used.
- The limiting variables are set using the flag `lim_var`, with the following options being avaialable:
  - `'prim'`
  - `'con'`
- The viscosity is estimated using the variable mentioned in `visc_var`. Currently, only `'density'` is supported. The same value of the dissipation is injected in all equations of the system.
- `plot_var` is a MATLAB-cell which listing the variables that should be plotted. The following options are available:
  - `'density'`
  - `'pressure'`
  - `'velx'` : x-velocity component.
  - `'vely'` : y-velocity component.
  - `'energy'`
- `clines` is also a MATLAB-cell array, with each element setting the contour lines for each variable mentioned in `plot_var`.
- The solution variables are saved in the **OUTPUT** directory, with the filename now also mentioning the `lim_var` and `ind_var` used. The `gas_const` and `gas_gamma` variables are also saved in the file. Furthermore, the variable `Q_save` saves all the conserved variables.
- The main driver script `EulerDriver2D` is called once all the flags have been set.

IC.m

The initial condition function must take as input the arrays `x`, `y`, `gas_gamma`, and `gas_const`, where `x` and `y` are of the shape $m \times n$. The output should be the conserved variables array `Q` of dimension $m \times n \times 4$. Once again, m generally denotes the number of DOFs per cell in the mesh, while n will be the number of cells. We give an example of this function below:

```
function Q = IC(x, y, gas_gamma, gas_const)

    xc = 1/6; yc = 0; aos = pi/3;

    p1 = 116.5;  rho1 = 8.0;    u1 = 7.14471;     v1 = -4.125;
    p2 = 1.0;    rho2 = 1.4;    u2 = 0.0;     v2 = 0;

    % Initial profile
    pre = p1*(y>(x-xc)*tan(aos) + yc) + p2*(y<=(x-xc)*tan(aos) + yc);
    u   = u1*(y>(x-xc)*tan(aos) + yc) + u2*(y<=(x-xc)*tan(aos) + yc);
    v   = v1*(y>(x-xc)*tan(aos) + yc) + v2*(y<=(x-xc)*tan(aos) + yc);
    rho = rho1*(y>(x-xc)*tan(aos) + yc) + rho2*(y<=(x-xc)*tan(aos) + yc);

    Q(:,:,1) = rho;
    Q(:,:,2) = rho.*u;
    Q(:,:,3) = rho.*v;
    Q(:,:,4) = Euler_Energy2D(rho,u,v,pre,gas_gamma);

return;
```

where the function `Euler_Energy2D` is an available custom function used to determine the total energy from the primitive variables. Similarly, the function `Euler_Pressure2D(Q,gas_gamma)` is available to evaluate the pressure from the conserved variables.

BC.m

When using non-periodic boundary conditions, a function script called BC.m is also needed. An example function is as follows:

```
function [rhoG,uG,vG,preG] = BC(ckey,time,gas_gamma,gas_const)

    xc = 1/6; yc = 0; aos = pi/3; M = 10;
    pre1 = 116.5;  rho1 = 8.0;    u1 = 7.14471; v1 = -4.125;
    pre2 = 1.0;    rho2 = 1.4;    u2 = 0.0;     v2 = 0;

    xs = time*M/sin(aos);

    if(ckey == 101 || ckey == 103)
        rhoG =@(x,y) rho1*ones(size(x));
        uG   =@(x,y) u1*ones(size(x));
        vG   =@(x,y) v1*ones(size(x));
        preG =@(x,y) pre1*ones(size(x));
    elseif(ckey == 102)
        rhoG =@(x,y) rho2*ones(size(x));
        uG   =@(x,y) u2*ones(size(x));
        vG   =@(x,y) v2*ones(size(x));
        preG =@(x,y) pre2*ones(size(x));
    elseif(ckey == 104)
        rhoG =@(x,y) 0*x;
        uG   =@(x,y) 0*x;
        vG   =@(x,y) 0*x;
        preG =@(x,y) 0*x;
    elseif(ckey == 105)
        rhoG =@(x,y) rho1*(y>(x-xc-xs)*tan(aos) + yc) + rho2*(y<=(x-xc-xs)*tan(aos) + yc);
        uG   =@(x,y) u1*(y>(x-xc-xs)*tan(aos) + yc) + u2*(y<=(x-xc-xs)*tan(aos) + yc);
        vG   =@(x,y) v1*(y>(x-xc-xs)*tan(aos) + yc) + v2*(y<=(x-xc-xs)*tan(aos) + yc);
        preG =@(x,y) pre1*(y>(x-xc-xs)*tan(aos) + yc) + pre2*(y<=(x-xc-xs)*tan(aos) + yc);
    end

return;
```

where the function takes in as input a boundary physical tag ckey, the curent simulation-time, `gas_gamma` and `gas_const`. The output is four functions to determine each of the primitive variables on the boundary.

back to table of contents

Using the MLP troubled-cell indicator

For those interested in using the trained indicator in their own solvers, we explain the various components of the network. The descriptor files for the various trained networks are available under the folder **Trained_networks**. For each network, the following files exist:

- ***model_parameters.dat***: Lists the dimensions of the input (IDIM) and output (ODIM) layers for the network, the number of hidden layers (NHL), and network hyperparameters (the Leaky ReLU factor, etc.).
- ***w_h{i}.dat***: Weights for the i'th hidden layer.
- ***w_out.dat***: Weights for the output layer
- ***b_h{i}.dat***: Biases for the i'th hidden layer
- ***b_out.dat***: Biases for the output layer.
- ***Scaling.m***: Script for scaling the input data before passing it through the network.

Consider the input X to be an array of size $m \times n$, where m is the dimension of each data sample, while n is the number of samples. The algortihm for the network with L hidden-layers having N_l neurons in the l-th hidden layer, is as follows:

$$
\begin{aligned}
X_0 &= \text{Scaling}(X), \quad N_0 = m \\
X_l &= f_{\text{activation}}(W_l X_{l-1} + b_l \mathbb{1}(n)), \quad W_l \in \mathbb{R}^{N_l \times N_{l-1}}, \quad b_l \in \mathbb{R}^{N_l}, \qquad l = 1, \ldots, L \\
Y &= \text{Softmax}(W_{out} X_L + b_{out} \mathbb{1}(n)), \quad W_{out} \in \mathbb{R}^{2 \times N_L}, \quad b_{out} \in \mathbb{R}^2
\end{aligned}
$$

where the final output Y is of dimension $2 \times n$, and $\mathbb{1}(n)$ is a row vector of n ones. The activation function is chosen as a suitable non-linear function, such as the Logistic function, hyperbolic tangent, ReLU, etc. The Softmax function is used a the output function. The first row of Y gives the probability of the sample corresponding to a troubled-cell, while the second row gives the probability of the sample corresponding to a good-cell. Note that the sum along each column of Y equals 1. The indices of the troubled-cells can be obtained as

$$
ind = \text{find}(Y(1,:) > 0.5).
$$

The MATLAB scripts to read and run the networks can be found under the folder **MLP_scripts**. For instance, the following scripts for 1D problems are available in the **1D** sub-directory:

- **read_mlp_param1D.m** reads the various weights and biases for a given network.
- **ind_MLP1D.m** runs the network for a given input `x`.

The **Common** sub-directory contains additional scripts needed to run the networks, such as the implementation of the leaky ReLU activation function.

back to table of contents

Network for 1D problems

The following is a list of the available networks for 1D problems. The latest recommended network is MLP_v1.

- **MLP_v1**: This network has an input layer of size 5. In particular, the input for the classification of cell i is $X = [\bar{u}_{i-1}, \bar{u}_i, \bar{u}_{i+1}, u^+_{i-\frac{1}{2}}, u^-_{i+\frac{1}{2}}]$, where the first 3 quantities are the cell averages of the solution in the cells $i-1, i, i+1$ and the last two entries are the left and right cell interface values of the approximating polynomial in cell i. There are 5 hidden layers, whose widths are 256, 128, 64, 32 and 16, going from the input to the output layer. The activation function is taken to be the Leaky ReLU activation function

$$f_{\text{activation}}(U) = max(0, U) - \nu(0, -U),$$

with the parameter ν. The details and results with this indicator are published here.

Network for 2D problems

The following is a list of the available networks for 2D problems. The latest recommended network is MLP_v1.

- **MLP_v1**: This network has an input layer of size 12. In particular, the input is the linear modal coefficient of each triangle in a 4-cell patch. There are 5 hidden layers of width 20 each. The activation function is taken to be the Leaky ReLU activation function. This network should be used with the filter to remove almost constant cells. The details and results with this indicator are available here.

back to table of contents

Using the MLP artificial viscosity estimator

For those interested in using the artificial viscosity network in their own solvers, we explain the various components of the network. The descriptor files for the various trained networks are available under the folder **Trained_networks**. A different network is trained for each polynomial degree $N \in \{1, \ldots, 4\}$. For each network, the following files exist:

- ***model_parameters.dat***: Contains the number of hidden layers (NHL), and network hyperparameters (the Leaky ReLU factor, etc.).
- ***wh{i}m{N}.dat***: Weights for the i'th hidden layer.
- ***woutm{N}.dat***: Weights for the output layer.
- ***bh{i}m{N}.dat***: Biases for the i'th hidden layer.
- ***boutm{N}.dat***: Biases for the output layer.
- ***Scaling.m***: Script for scaling the input data before passing it through the network.
- ***Scaling_inverse.m***: Script for scaling the output data returned by the network. As discussed in the report, it returns a constant value equal to the maximum among the components of each output vector. Note that among the input arguments, a characteristic mesh size and a local wave speed have to be provided.

Consider the input X to be an array of size $m \times n$, where m is the dimension of each data sample, while n is the number of samples. The algortihm for the network with L hidden-layers having N_l neurons in the l-th hidden layer, is as follows:

$$\begin{aligned}
X_0 &= \text{Scaling}(X), \quad N_0 = m \\
X_l &= f_{\text{activation}}(W_l X_{l-1} + b_l \mathbb{1}(n)), \quad W_l \in \mathbb{R}^{N_l \times N_{l-1}}, \quad b_l \in \mathbb{R}^{N_l}, \qquad l = 1, \ldots, L \\
Y &= \text{Softplus}(W_{out} X_L + b_{out} \mathbb{1}(n)), \quad W_{out} \in \mathbb{R}^{m \times N_L}, \quad b_{out} \in \mathbb{R}^m \\
\mu &= \text{Scaling_inverse}(Y)
\end{aligned}$$

where the network output Y is of dimension $m \times n$, while the artificial viscosity has dimension $1 \times n$ in the current implementation. The activation function is chosen as a suitable non-linear function, here chosen as the Leaky ReLU function. The Softplus function is used as the output function to ensure nonnegativity.

The MATLAB scripts to read and run the networks can be found under the folder **MLP_scripts**. For instance, the following scripts for 1D problems are available in the **1D** sub-directory:

- **read_mlp_param1D_visc.m** reads the various weights and biases for a given network. Note that `N` has to be provided as input to the function.
- **visc_MLP1D.m** runs the network for a given input `x`.

The **Common** sub-directory contains additional scripts needed to run the networks, such as the implementation of the leaky ReLU and Softplus activation function.

back to table of contents

Network for 1D problems

The following is a list of the available networks for 1D problems. The latest recommended network is MLP_visc.

- **MLP_visc**: This network has an input layer of size `N+1`. In particular, the input to estimate artificial viscosity of cell i consists of the solution degrees of freedom in the same cell. There are 5 hidden layers, with 10 neurons each, and Leaky ReLU activation function. The output layer has again dimension `N+1`, and the Softplus activation function is used.

Network for 2D problems

The following is a list of the available networks for 2D problems. The latest recommended network is MLP_visc.

- **MLP_visc**: This network has an input layer of size `(N+1)(N+2)/2`. In particular, the input to estimate artificial viscosity of cell i consists of the solution degrees of freedom in the same cell. There are 5 hidden layers, with 20 neurons each, and Leaky ReLU activation function. The output layer has again dimension `(N+1)(N+2)/2`, and the

Softplus activation function is used.

Remarks

We emphasise that:

- The network input and output are scaled versions of the solution and viscosity values. This is why the functions **Scaling.m** and **Scaling_inverse.m** play an important role.
- With the current implementation, we predict a constant viscosity coefficient in each cell. This is chosen as the maximum among the local values of the dissipation.
- To ensure continuity of μ and enhance subcell resolution, the pointwise values of the viscosity are obtained after a smoothing algorithm (see, e.g., **Scalar1D_smooth_viscosity.m**).

back to table of contents