Compack user's guide

Ulrik Skre Fjordholm

Contents

CHAPTER 1

Introduction

1.1. Who is this for

Compack (**Co**nservation law **M**ATLAB **pack**age) is a MATLAB package for solving hyperbolic conservation laws in one and two spatial dimension. Emphasis has been put on accessibility to inexperienced programmers and people new to the field of conservation laws, while at the same time being easy to extend and modify. The program implement solvers for everything from linear scalar equations to non-linear systems of equations in one and two spatial dimensions on both Cartesian, rectangular and triangular (unstructured) meshes.

Compack has been tested and works for MATLAB 7.9, and should work for version 7.6 and later.

1.2. A note on code organization

The program is organized into MATLAB *packages*, which are folders that begin with a + character. To call functions within a package, the function must be prefixed by the package name, followed by a . character. For instance, to call the `plotSolution()` function in the `Plot` package, type `Plot.plotSolution()`. For more information, see the MATLAB help page on packages.

CHAPTER 2

Getting started

2.1. A first example

The easiest way of learning how to use Compack is through the examples in the `Examples` package. In this section we go through the `linAdv()` example, which computes an approximate solution to the linear advection equation

$$u_t + au_x = 0$$

for an $a \in \mathbb{R}$ on the periodic domain $x \in [-1, 1]$. Run this function now by navigating to the Compack base directory and typing `Examples.linAdv()`. You should see an animation of a moving sine wave slowly decaying in magnitude. If you open the `linAdv.m` file you will see the following:

```
1  function soln = linAdv
2      %%% Set configurations
3      conf = Configuration;
4
5      % Create the model object and set the advection speed to 1
6      conf.model = Model.LinAdv;
7      conf.model.a = 1;
8
9      conf.solver = Flux.Rusanov;
10     conf.timeInt = @TimeIntegration.FE;
11     conf.tMax = 2;
12     conf.CFL = 0.4;
13     conf.mesh = Mesh.Cartesian([-1,1], 200);
14     conf.bc = Mesh.BC.Periodic;
15     conf.initial = @(x) sin(2*pi*x);
16
17
18     %%% Run solver
19     soln = runSolver(conf);
20
21
22     %%% Display data, etc.
23     Plot.plotSolution(soln, [-1,1]);
24 end
```

Let's go through the code line-by-line.

```
3      conf = Configuration;
```

The first thing you do is create a `Configuration` object. The `Configuration` object contains all information about the problem you want to solve, such as which model

to solve, which numerical flux to use, information on the computational domain, etc.

```
6       conf.model = Model.LinAdv;
7       conf.model.a = 1;
```

Next, we create and configure the model object. Since we wish to solve the linear advection equation, we use the `LinAdv` class. Other model classes that are currently implemented are `Burgers` for the Burgers equation, `Wave` for the wave equation and `SW` for the shallow water system. The `LinAdv` class has a parameter `a` specifying the advection speed. Above we set this speed to 1.

```
9       conf.solver = Flux.Rusanov;
```

The `Flux` package contains all numerical fluxes. Implementing additional fluxes is simply a matter of adding a new MATLAB class file to this package. Above we set the `solver` property of the `Configuration` object to an instance of the `Rusanov` class.

```
10      conf.timeInt = @TimeIntegration.FE;
11      conf.tMax = 2;
12      conf.CFL = 0.4;
```

Next, we specify parameters relating to the time integration. The `FE()` function in the `TimeIntegration` package implements a forward-Euler discretization of the time derivative u_t. Other functions in this package are `RK2`, `RK3` and `RK4`, which implement 2nd, 3rd and 4th order Runge-Kutta integration methods. Note the usage of the at symbol @ before the name of the function. This returns a *function handle* to the function, instead of calling the function. For more information see the MATLAB help page on function handles.

In line 11 we specify that we want to compute up to $t = 2$, and in line 12 we set the CFL number to 0.4.

```
13      conf.mesh = Mesh.Cartesian([-1,1], 200);
14      conf.bc = Mesh.BC.Periodic;
```

Here we set properties relating to the computational domain. First, we specify the computational domain by creating a `Cartesian` object. The constructor takes a 2-vector specifying the computational domain (in this case $x \in [-1, 1]$) and the number of grid points to discretize with.

On line 14 we set the boundary condition to periodic, meaning

$$u(-1) = u(1).$$

Also implemented is the `Neumann` class, which gives a homogeneous Neumann boundary condition

$$u_x(-1) = 0, \qquad u_x(1) = 0.$$

```
15      conf.initial = @(x) sin(2*pi*x);
```

Last, we specify the initial data, in this case a sine wave. If you find the above syntax strange, read the MATLAB help page on anonymous functions.

```
19      soln = runSolver(conf);
```

Now that the `Configuration` object has been initialized, we can run the solver through the `runSolver()` function. This might take some time, depending on the speed of your computer. The return value is an object of the `Solution` class, the second important class in Compack. This object contains all information about the computed solution – most importantly, the solution itself. In this simple example, the `Solution` object is used to plot the computed solution in the following function call:

```
23      Plot.plotSolution(soln, [-1,1]);
```

The `plotSolution()` function, part of the `Plot` package, plots the computed solution over time. To start the animation, press Enter or click inside the figure that appears. The `plotSolution()` function takes the `Solution` object and two additional, optional parameters: the axis scaling and the name of the variable to plot. The latter is only relevant for systems of conservation laws, where you have several different variables, and has been left out in this example. The axis scaling parameter forces MATLAB to use the specified range for the y-axis (or z-axis for two-dimensional conservation laws) in the figure. Replacing this parameter by `0` or simply leaving it out will prompt MATLAB to use its default axis scaling.

2.2. Further examples

The `Examples` package contains several example scripts. As with `linAdv()`, these examples are run by simply typing `Example.nameOfExampleFile` while in the Compack base directory. A short description of some of the example files follow.

- `linAdv2D`: This function computes the two-dimensional linear advection equation
$$u_t + a_1 u_x + a_2 u_y = 0$$
in the periodic domain $(x, y) \in [-1, 1] \times [-1, 1]$. Note in particular the minor changes from the `linAdv` example: the advection speed is now a vector $[a_1, a_2]$; the computational domain is a 2×2 array; and the initial data is a function of two variables x and y.
- `linAdvOOC`: This function sets up the same problem as in `linAdv()`, with two important differences. First, a second-order time integration method (`RK2`) is used instead of the first-order `FE` forward-Euler method, and second, a the second-order minmod slope limiter `Lim_MM` is set in the reconstruction phase of the algorithm. This will result in an overall second-order accurate method. To check that this is indeed the case, we use the `calcOOC` function of the `Error` package to calculate an approximate rate of convergence:

```
15      %%% Calculate rate of convergence
16      % Exact solution of the linear advection equation
17      exact = Error.exact_linAdv(conf.initial);
```

```
18      % Which mesh sizes to compute over
19      nx = 100:100:500;
20      % Which L^p norms to compute
21      p = [1, inf];
22      % Calculate the approximate rate of convergence.
23      Error.calcOOC(conf, exact, nx, p);
```

The `exact_linAdv` function returns a function (or more precisely, a function handle) that computes the exact solution at any point of time, given the initial data `conf.initial`. We set the `calcOOC` method to compute errors over mesh sizes $n_x = 100, 200, \ldots, 500$, and last we specify that we want to compute errors in both the L^1 and L^∞ norms.

If you run `linAdvOOC`, the computer will work for some time and then output the results to console, in addition to plotting error graphs. You will notice that the computed rate of convergence is not the expected one of 2, but something between 1 and 2. By experimenting with using a lower CFL number (to minimize the effect of time discretization) or a more accurate time integration method (such as `RK3` or `RK4`), or testing different limiters (such as the MC limiter `Lim_MC` or the superbee limiter `Lim_SB`), you may get convergence rates closer to 2.

- `burgers`: This function solves Burgers' equation

$$u_t + \left(\frac{u^2}{2}\right)_x = 0$$

using Roe's method. Since this method is different for different models (that is, for different fluxes f), the `Roe` class is found in the `Burgers` subpackage of the `Flux` package:

```
6       conf.solver = Flux.Burgers.Roe;
```

CHAPTER 3

Extending Compack

Compack has been written to facilitate quick and easy implementation of new numerical fluxes, boundary conditions, slope limiters etc. In this section we outline how to implement some of these yourself.

The package `User` has been created to hold your user scripts. A good starting point would be to copy one of the files found in the `Examples` package (for instance the `linAdv` function) to the `+User` folder and play around with that copy.

In the remainder of this section we explain by example how to extend different parts of the program.

3.1. Numerical fluxes

The library of numerical fluxes that comes with Compack is rather limited, so we will implement an additional method, the upwind method, for the linear advection equation. Recall that when the advection speed a is positive, this method takes the form

$$U_j^{n+1} = U_j^n - a\frac{\Delta t}{\Delta x}\left(U_j^n - U_{j-1}^n\right),$$

and when a is negative,

$$U_j^{n+1} = U_j^n - a\frac{\Delta t}{\Delta x}\left(U_{j+1}^n - U_j^n\right).$$

This can be written in the flux form

$$U_j^{n+1} = U_j^n - \frac{\Delta t}{\Delta x}\left(F_{j+1/2} - F_{j-1/2}\right)$$

by setting

$$F(U_L, U_R) = \begin{cases} aU_L & \text{if } a > 0 \\ aU_R & \text{if } a < 0. \end{cases}$$

To start, we create a new file `Upwind` in the `Flux.LinAdv` package, and start with the class skeleton

```
1  classdef Upwind < Flux.NumFlux
2      %UPWIND   Upwind scheme
```

Numerical fluxes in Compack are implemented as MATLAB classes. As all numerical fluxes share some common properties, such as the need to access the `Model` object, they derive from the base class `Flux.NumFlux`. Next, we specify the name of the solver by setting the `name` property:

```
4      properties
5          name = 'Upwind'
```

```
6       end
```

All classes deriving from the `Flux.NumFlux` must define this property. Last, we implement the flux method and close the class definition by adding

```
8       methods
9           function ret = F(obj, Ul, Ur, UlR, UrR, t, dt)
10              a = obj.model.a(1);
11              if a > 0
12                  ret = a*UlR;
13              else
14                  ret = a*UrR;
15              end
16          end
17      end
18  end
```

Let's go through this line-by-line. In line 8 we start a class method block with the `methods` directive. Line 9 contains the function declaration of our numerical flux function `F`. The `obj` variable is a pointer to the `Upwind` object, akin to the `this` pointer in C++ and Java or the `self` pointer in Python. The next two parameters `Ul` and `Ur` are the left and right cell averages. Next come the left and right reconstructed values `UlR` and `UrR`. When no reconstruction procedure is used, we have `UlR = Ul` and `UrR = Ur`. To use the MUSCL-type approach of higher-order accurate numerical fluxes, one should use `UlR`, `UrR` instead of their cell average counterparts `Ul`, `Ur`. Last come the current time `t` and the time step `dt` – none of which are used in the present function.

In line 11 we check if the advection speed in the x-direction – the first component of the `a` property – is positive. Here, `model` is a property of the `NumFlux` base class, and is simply a pointer to the `Model` object that we created earlier. If the advection speed turns out to be positive, we set the flux to be the flux coming from the left state, `a*Ul`; if not, we set it to be `a*Ur`. The full listing of the `Upwind` class then reads

```
1   classdef Upwind < Flux.NumFlux
2       %UPWIND   Upwind scheme for the linear advection equation
3
4       properties
5           name = 'Upwind'
6       end
7
8       methods
9           function ret = F(obj, Ul, Ur, UlR, UrR, t, dt)
10              a = obj.model.a(1);
11              if a > 0
12                  ret = a*UlR;
13              else
14                  ret = a*UrR;
15              end
16          end
17      end
18  end
```

Using the `Upwind` class is now simply a matter of setting

```
conf.solver = Flux.LinAdv.Upwind;
```

in your user script.

3.2. Models

By "model" we mean the equation that we are trying to solve,

$$u_t + f(u)_x = 0.$$

The model is uniquely determined by the flux function f. To implement a new model in Compack, the flux function must be supplied, along with some helper functions. This is done by extending the `Model.ModelBase` base function.

Note that all non-static member functions have a parameter `o`, which is a handle to the current model object (similar to `this` in C++ or `self` in Python).

3.2.1. Model.ModelBase methods.

Public methods with a default implementation.

`calcTimestep(o, U, mesh)`

Calculates the maximum currently allowable timestep using the `maxEig` function. Called by `runSolver`.

`breaksPositivity(o, U)`

Some models have one or more variables that must be in a certain range to make sense physically. For instance, pressure and density cannot be negative. Model classes can overload this function to check whether a solution is in the required range, returning `true` if it is and `false` if it isn't. The solver will immediately stop running if this happens.

`maxEig(o, U, d)`

Computes the maximum eigenvalue of the flux Jacobian in direction `d` in each cell. `d` is either an integer $\in \{1, 2\}$, or an array of normal vectors. If `d`$\in \{1, 2\}$, then the return value is maximum eigenvalue of $\frac{df^{(d)}}{dU}(U)$ (1 and 2 indicate x- or y-direction, respectively). If `d` is an array of normal vectors, then the return value is the maximum eigenvalue of $\frac{d(\mathbf{f}\cdot n)}{dU}(U)$. The work of computing these is delegated to `maxEigRect` and `maxEigDir`, respectively.

`getVariable(soln, U, varname)`

Extracts or computes a specific component from the solution `U`. `varname` is either an integer index into the solution vector, or a string describing the variable. If `varname` is a string, then the function `doGetVariable(soln, U, varname)` is used to compute the variable. This function is static, so it can be called as `ModelName.getVariable(...)`, where `ModelName` is the model class.
As an example, if `soln` is the Solution object for a computation for the Euler equations, use

```
[t,U] = soln.get(soln.end());
rho = model.getVariable(soln, U, 'rho');
```

to retrieve the density component at the last timestep.

`doGetVariable(o, soln, U, varname)` *(protected)*
Called by `getVariable`. Returns the variable with name `varname` from the solution vector `U`. This computation is model-specific, and hence this function must be overloaded in each model class. The default implementation returns the first component of `U`, equivalent to `getVariable(soln, U, 1)`.

`maxEigRect(o, U, dir)` *(protected, abstract)*
Called by `maxEig`. Computes the maximum eigenvalue of $\frac{df^{(d)}}{dU}(U)$. This function must be implemented in all model classes.

`maxEigDir(o, U, n)` *(protected)*
Called by `maxEig`. Computes the maximum eigenvalue of $\frac{d(\mathbf{f}\cdot n)}{dU}(U)$. The default implementation computes this as $|\lambda \cdot n|$, where λ is a vector containing the maximum eigenvalue in each direction, computed with `maxEigRect`.