Introduction to ECO

André Marchildon, Alex Bercik

October 29, 2020

This is a brief introduction to ECO, Efficient Chaotic Optimizer for the numerical solution of PDEs and ODEs in Python.

1 Code Organization

The code is structured into 3 main sections: `Driver`, `Source`, and `Tests`.

1.1 `Driver`

This directory contains the files required to run the code. These files define parameters, then call the classes defined in `Source` to initialize problems with the parameters provided. Each instance of the class defines a new problem, and so if multiple problems are to be run at once, one only need to define several instances in the driver file. One can also perform analysis here in the driver file.

1.2 `Tests`

TO DO

1.3 `Source`

Holds the source code for the solver. `DiffEq` holds classes for each type of equation to be solved, as well as base classes `DiffEqBase` and `SatBase`. Here is where information on how to solve the equation is stored, such as how to calculate the residual. `Disc` stores information for spatial discretization, such as definition of operators and the mesh. `Grad` calculates the gradient of the objective with respect to design variables using the adjoint, tangent and other modified methods applicable to chaotic systems. `MakeFig` defines some plotting things. `Optz` has some routines for Bayesian optimization. `Results` contains text files exported from runs either completed locally or on SciNet and also classes to plot these results. `Solvers` holds information specific to the method used, such as Finite Difference, SBP, or DG. This calls `Disc` classes and modifies the methods defined in `DiffEq`, in a sense completing the problem initialization. `TimeMarch` defines methods to time march the resulting discretized ODE and is called by the `solve` function in `Solvers/OdeSolver` once the differential equation is properly initialized.

2 Example Problem: SBP Linear Convection

We begin in `Driver/RunLCE` by defining parameters. We choose one of the 4 SBP finite difference choices for `disc_type`, noting that since we are solving the problem in 1 dimension, `'Rdn1'` and `'R0'` will result in the same operators.

We initialize a class instance `diffeq` for our problem with

```
from Source.DiffEq.LinearConv import LinearConvSbp
diffeq = LinearConvSbp(para, obj_name, q0_type)
```

where `para` is the wave speed, `obj_name=None` means we have no objective function to calculate (see 3 for the case where calculate an objective function), and `q0_type` is a string that will automatically create an initial condition under `PdeBase.set_q0` (unless a specific initial condition is given to the solver as variable `q0`).

The initialization of `diffeq` will first trigger the initialization defined in `LinearConv`, which stores the wave speed `para` as an attribute, and then inherits the initialization from `PdeBase`. This similarly stores the initial condition setting as an attribute, then once again inherits the initialization of `DiffEqBase`. Aside from a few more parameters being redundantly stored, no further initialization occurs for this case. We notice however important differences for cases with varying parameters. Since these cases differ in treatment in several aspects of the code, they are described later in 4.

`diffeq` is now a class instance with access to all the functions responsible for calculating the residual in any given physical element. These functions include various derivatives of the residual used for implicit time marching, and includes functions to calculate the contributions from the SATs at any particular interface. This is because `diffeq` inherited the functions from `PdeBaseCons` (specific to conservative PDEs in the form $q_t + E_x(q) = G(q,t)$) as well as its parent classes `PdeBase` and `DiffEqBase`, but also from `SatBaseCons` and `SatBase`. These functions however are not complete, as they require information about the spatial discretization. For this we must initialize the solver class.

```
from Source.Solvers.PdeSolver import PdeSolver
solver = PdeSolver(diffeq,
                   tm_method, dt, tf, t_init,
                   q0, n_q0,
                   p, spat_disc_type, nn, nelem, nen,
                   isperiodic, xmin, xmax,
                   bool_plot_sol = bool_plot_sol,
                   print_sol_norm = print_sol_norm)
```

The function `PdeSolver` simply returns an initialization of the `PdeSolverSbp` class with the relevant parameters. The first step of this initialization is to initialize an instance of the `MakeSbpOp` class under the attribute `self.sbp` within `solver`. This stores all information relevant for the spatial discretization on the reference element. Based on the degree `p`, and either the total number of nodes `nn` or number of elements `nelem` and number of nodes per element `nen`, the SBP operators are created and the remaining variables are assigned. Next the 1D mesh is created by an initialization of the `Disc/MakeMesh` class under `self.mesh` using the physical boundaries `xmin` and `xmax`, a flag `isperiodic` indicating periodicity, and the number of elements and nodal locations defined in the previous step. Because a simple linear mapping with equal size elements is employed, the mesh Jacobian is constant over the entire domain. This allows us to compute the mesh Jacobian, its inverse, and the determinant only once using the first element. Likewise, we then use these quantities to compute the physical element-wise operators only once by then calling `self.sbp.ref_2_phys_op`, as the physical operators are the same in each element. Note that this function as defined in `MakeSbpOp` has yet to

be generalized for higher dimensions, and would also need to be called individually for each element should a more general mesh mapping be employed. Finally, a Kroncker product is applied for cases where q is a vector (i.e. systems).

The final step is to incorporate these quantities with the methods defined in the class instance `diffeq`. To do this, first `diffeq` is stored as an attribute within `solver` under the name `self.diffeq_in` (the reason for this name will become apparent soon). The mesh attributes and physical operators currently defined in `solver` are then passed to `solver.diffeq_in` to be stored as attributes there with the functions `solver.diffeq_in.set_mesh(...)` and `solver.diffeq_in.set_sbp_op(...)`. Without this step, as with any call to an instance of the `DiffEq` class, functions in `solver.diffeq_in` will return errors as the instance is missing the required attributes. `solver.diffeq_in` now contains all the relevant functions to calculate the residual in a given physical element, however, crucially, it does not have the ability to compute the global residual. Therefore, an instance of the class `Diffeq4SbpSolver` (defined in file `PdeSolverSbp.py`) is now initiated as an attribute within `solver` under the name `self.diffeq`. Not to be confused with `solver.diffeq_in`, the initialization of `solver.diffeq` takes functions originally defined in `solver.diffeq_in`, such as `solver.diffeq_in.dqdt` and `solver.diffeq_in.calc_sat`, which calculate the residual for an individual element interior and SAT contributions at an individual element interface, and combines them into a global function `solver.diffeq.dqdt` that calculates the residual over the entire domain. The initialization of `solver.diffeq` takes global functions defined in `PdeSolverSbp` as arguments (ex. `dqdt_sbp(...)` for the example above) and sets them as methods that can be called directly in `solver.diffeq`. In this way, the class instance `solver` now contains all the information required to solve the PDE.

The now semi-discrete system of ODEs (in time) can now be solved by simply calling

```
solver.solve()
```

For cases with non-varying parameters, this reverts to the function `solve_main` defined in `Solvers/OdeSolver`. For problems that do not require first solving for a steady initial condition, this first sets the initial condition using `q0 = solver.diffeq.set_q0()`, defined in `PdeBase`. For other problems, see 7. The system of ODE's is then marched in time by

```
tm_class = TimeMarching(self.diffeq, self.tm_method, keep_all_ts,
                        bool_plot_sol = self.bool_plot_sol,
                        bool_calc_obj = self.bool_calc_obj,
                        print_sol_norm = self.print_sol_norm)
self.q_sol =  tm_class.solve(q0, self.dt, self.n_ts)
self.obj = tm_class.obj
self.obj_all_iter = tm_class.obj_all_iter
```

Upon initialization of the class instance `tm_class`, the relevant functions from `solver.diffeq`, which are passed as argument, are extracted and the time marching method is set according to the string `tm_method` originally passed to `solver`. The boolean `keep_all_ts`, hard coded in `OdeSolver.solve_t_final` (a subroutine of `OdeSolver.solve_main`), determines whether to return the solution vector at each time step or to simply return the final solution vector when calling `tm_class.solve(...)`. Similarly, the boolean flags `bool_plot_sol` and `print_sol_norm` that were originally passed to `solver` now determine whether or not to plot the solution and print the L_2 solution norm at each time step. `bool_calc_obj` controls the calculation of objective functions. Back when `diffeq` was initialized, the string `obj_name=None` was set, indicating there is no objective function to calculate, and setting `solver.bool_calc_obj=False`. Finally, the time marching method is set according to the string `solver.tm_method`. These methods are defined in parent classes inherited by the main `TimeMarching` class.

With all the required information to solve the residual and its derivatives stored in the passed class `solver.diffeq`, the problem is now marched in time by calling `tm_class.solve(...)` along with the initial condition, step size, and number of steps. The solution vector (at all time steps), and two empty variables corresponding to the non-existent objective function are finally passed back to `solver`.

3 Objective Functions

Upon initialization of `diffeq`, `obj_name` can be set to either a string or a tuple of strings, each labelling a particular objective function. The length of this tuple determines the parameter `diffeq.n_obj`. When `diffeq.n_obj` > 0, the initialization now calls the function `DiffEqBase.init4obj()`. Once again we assume the case of nonvarying parameters, and set `self.all_para_set = True`. For the alternative case, see 4. If the objective has been previously evaluated and saved locally within a text file, the path to this file is now stored as an attribute. Note that these paths must be specified in the `__init__` functions of the problem-specific `DiffEq` files, otherwise an Attribute Error will be raised. In addition there must be functions defined in the problem-specific `DiffEq` file responsible for calculating each objective function, with a general function `calc_obj()` serving to call the individual functions and return a single vector with all desired objectives.

In the initialization of the solver class, the argument `bool_calc_obj` is passed as a boolean and stored as an attribute. This flag simply determines whether or not to evaluate the objective, and the default is set to `True` (unless `diffeq.n_obj` $= 0$). An additional method `solver.calc_obj` is also defined, which simply calls the main solve routine with an optional intial condition, but only returns the objective and standard deviation. This could be useful in post-analysis if only the objective is of interest. We stress the difference between the functions `diffeq.calc_obj` (and similarly `solver.diffeq.calc_obj`) and `solver.calc_obj`. The former defines the actual routine to calculate the objective function, wheras the latter calls the `solve()` routine.

When the time marching class is initiated, there is an option to manually overwrite the objective function by passing a new function as an argument. If using this option however, one must be careful that it take the three arguments (current solution vector, total number of time steps, time step size) and return `len{obj_name}` values. If this manual override option is not used, the objective function `self.fun_obj` is simply set as the default function from the given `diffeq` class instance, `solver.diffeq.calc_obj`. On each iteration of the time marching scheme, the function `self.fun_obj` is called within the `common(...)` routine of `solve(...)`, and the results are stored.

At the end of the time marching (return to `OdeSolver/solve_main()`), three quantities are returned. The first is the solution vector, the second is an array that contains the objective functions at each iteration, and the third is the sum of the objectives over all time steps. The second and third quantities are stored as `solver.obj_all_iter` and `solver.obj`, respectively. A running average of the objective functions is then performed, and quantities such as the standard deviation, `solver.std_obj`, are also stored.

4 Varying Parameters

5 Functions in DiffEq

6 Time Marching

7 Other Things I don't understand

The two arguments `q0_in=None` and `q0_idx=None` in `OdeSolver.solve_main(...)`. Does `q0_in` simply define an initial condition that overwrites the default initial condition? Then I have no idea what `q0_idx` does.

8 TO DO in code

For SBP, if a specific quadrature is desired that differs from the default choices (ex. Newton-Cotes), although the option exists to construct the operators in `Disc/MakeSbpOp`, one can not select this option from the Driver file.

DO DG

Generalize the DG part from collocated DGSEM

Get entropy stable schemes working