Introduction to Ordinary Differential Equations Università della Svizzera italiana
Giacomo Rosilho de Souza

PROJECT
NUMERICAL INTEGRATION OF DYNAMICAL SYSTEMS

Collocation Methods, Time Step Adaptivity and Nonlinear System Solvers

Goals. In this numerical project, we explore the implementation aspects of Runge–Kutta methods. The project is structured in three parts. Firstly, we delve into the implementation of collocation methods of arbitrary order, with a particular emphasis on Gauss and Radau methods. Secondly, we incorporate time step adaptivity into our code by employing the embedded methods strategy. Thirdly, we investigate various approaches for solving the Runge–Kutta nonlinear system, considering factors such as Newton and quasi-Newton algorithms, direct and iterative solvers, Jacobian construction, matrix-free methods, and the inclusion or exclusion of preconditioners. Throughout these explorations, we conduct numerical efficiency experiments on problems of diverse nature, evaluating the effectiveness of different approaches. By the end of the project, participants will gain proficiency in implementing and comprehending the aforementioned schemes, enabling them to judiciously select the appropriate nonlinear solver for specific problem contexts.

Have Fun 🤓!

November 24, 2023.

1. Guidelines

Let us set up the project framework and provide some directives.

1.1. Material

All the theoretical knowledge that you need to complete the project is in the course notes. Concerning the code, the implementation of all the aforementioned combinations of schemes has the potential to result in a tasty spaghetti code 🍝. To mitigate this challenge, I offer a template code as a starting point. You will find all the features of the code already provided at the beginning of this course, but the structure has been adapted to the project. In addition, new examples have been added.

The template code can be downloaded from here https://github.com/grosilho/ODE_Project_ImplementationRungeKutta

1.2. Directives

Here are some rules for the project.

- For each task in the project, you are required to fill in specific sections of the provided template code. Once you have completed your work, please organize the finalized code into a compressed archive named `Codes_your_name.zip`.
- The difficulty increases; hence the last points are harder to earn 🤯.

- To verify the correctness of your codes, we will run some scripts that you can find in the `Grading` folder. If the scripts run smoothly and produce the expected or reasonable results, you get full points for that part of the project. If not, we remove points for missing features or incorrect results.
- Different implementations might give rise to slightly different results, this is taken into account in the evaluation, don't worry 😉. However, hyperparameters such as Newton tolerances and iterative solver tolerances are already set, hence your results will likely match ours.
- We only grade pictures and console outputs. We do not examine the codes or engage in debugging activities 😳.
- The project must be carried out individually 🧐.
- The deadline for the project is on December 31 at 23h59' 🥳.
- The project will be awarded between 0 and 2 points, which will act as a bonus on your final grade.
- However, the total points accumulated from both the project and assignments cannot exceed 5 points in total.

2. Assignments

Finally, here are the assignments for the project. Oftentimes, you will find some hints in the code itself. Since each part is composed of many substeps, I suggest continuously testing your implementation by running `Convergence.py` experiments on simple problems. Yes, I know, "convergence experiments again 🥱", but they are the best tool for checking that your method is implemented correctly [1]. Alternatively, for quick checks (not so robust), use the `Integrate.py` script on a problem where the exact solution is known and check the errors and plots.

2.1. Collocation Methods

In this part we implement the $G(\boldsymbol{z})$ function, compute the Runge–Kutta coefficients of collocation methods from the collocation nodes c and finally compute the collocation nodes of Gauss and Radau methods.

i) (0.20 pts.) Implement the $G(\boldsymbol{z})$ function in the `ImplicitRK` class. Test your implementation by running convergence experiments with `LobattoIIIC` Runge–Kutta method, which is already implemented. Set the nonlinear solver to `scipy` and quasi-Newton to `False`.

ii) (0.10 pts.) Finish the implementation of the `Collocation` class. Hence, given c you should compute $\mathcal{A}$ and b. Test the code with the `RandomCollocation` class, try with different s values.

iii) (0.10 pts.) In the `Gauss` and `Radau` classes, implement the computation of the c collocation nodes. Verify order of convergence for different values of s.

iv) (0.05 pts.) Finally, a small code improvement. In `ImplicitRK.get_y1` we are doing additional f evaluations. However, in the considered cases, $\mathcal{A}$ is invertible, hence you can improve the code as explained in the lecture notes.

v) (0.05 pts.) For the Radau collocation method you can improve the code even further. Indeed, in this case, $a_{sj} = b_j$. Hence $g_s = y_1$ and thus $y_1 = y_0 + z_s$.

Try to run the `Grading/Collocation.py` script and verify that you get the same plots as in Fig. 1. Some methods saturate, this is normal behavior when very small errors are reached. The reasons can be multiple: 1) the round off error starts to dominate over the discretization error, 2) the method reached the same accuracy as the reference solution, 3) the tolerance on the nonlinear solver is too high and the Newton error starts to dominate, 4) same for iterative linear solvers, if used.

[1] Remember that convergence results are meaningful if you have a reference solution that is more accurate than the methods that you are testing. When you run a convergence experiment, the template code will look for an available reference solution, otherwise it computes one on the fly and writes it to file for future use. If you change any parameter in the problem or employ considerably more accurate solvers, it becomes necessary to recalculate the reference solution. This can be achieved by deleting the corresponding file in the `ReferenceSolutions` folder, prompting the code to generate a new one based on the changed parameters. I already provided some reference solutions, so you don't need to stress about it too much.

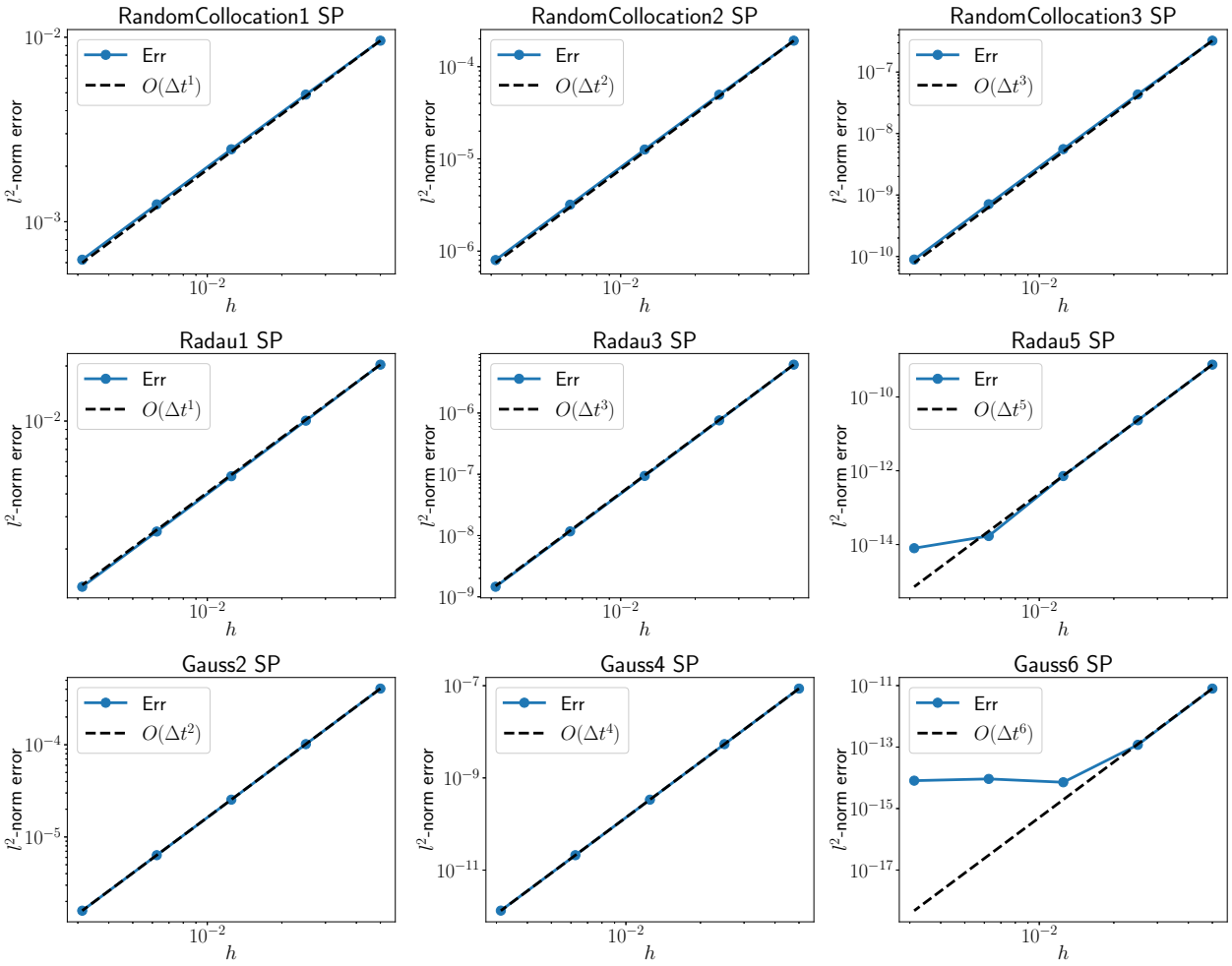


FIGURE 1. Results of `Collocation.py`

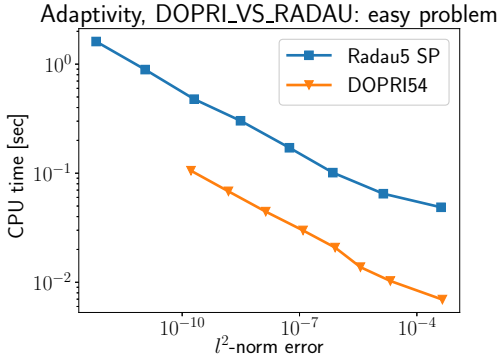
2.2. **Time Step Adaptivity**

Here we implement time step adaptivity and the error estimators via embedded methods. Since embedded methods require additional coefficients $\hat{b}$, which are specific to a Runge–Kutta method, here we restrain to the Radau method with $s = 3$ (Radau5 in the literature) and the DOPRI5(4) method.

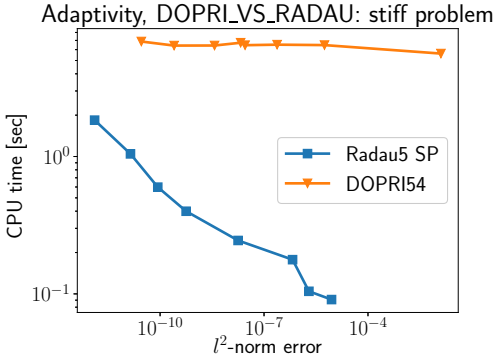
i) (0.15 pts.) Finish the implementation of the DOPRI5(4) method in `ExplicitRK.py`. To verify correctness, check the order of convergence, you should get $p = 5$. Then, temporarily replace b with $\hat{b}$ and check again the order of convergence, you should get $\hat{p} = 4$. Do not forget to switch them back to the correct setting.

ii) (0.35 pts.) Implement time step adaptivity (Algorithm 2 in the lecture notes) in the `Integrator` class. Remember to take into account the possibility of deactivating adaptivity and keeping the step size constant. Check the correctness by `Integrate.py` a simple problem like `SIRV` with time step adaptivity on and off and compare the number of steps. Try with different error tolerances.

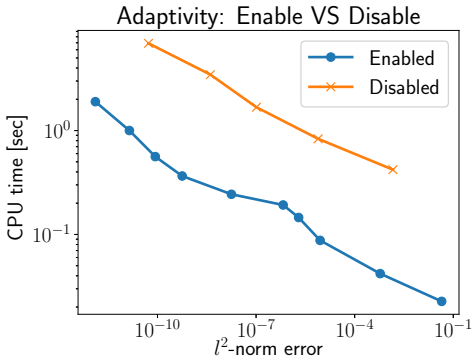
iii) (0.10 pts.) Finish the implementation of the `DOPRI54.estimate_error` and `Radau.estimate_error` functions. The error estimator for Radau with $s = 3$ is slightly different than standard ones. It has order $\hat{p} = 3$



(A) `Adaptivity_DOPRI_VS_RADAU_easy.py`



(B) `Adaptivity_DOPRI_VS_RADAU_stiff.py`



(C) `Adaptivity_ENABLE_VS_DISABLE_stiff.py`

```
Time: 100%|
Integrator: Radau5 SP
Solved in 1.18219899 seconds
Total steps: 374 steps
Accepted steps: 372 steps
Rejected steps: 2 steps
Average Newton iterations per RK step: 29.15
Average linear solver iterations per Newton step: 0.00
Relative error: 1.1247031641736758e-08
```

(D) `Adaptivity_number_of_steps.py`

FIGURE 2. Time step adaptivity efficiency results.

and is given by

$$\widehat{E}_{n+1}(h_n) = |\widehat{y}_{n+1} - y_{n+1}| = \left| \hat{b}_0 h_n f(t_n, y_n) + \sum_{i=1}^{s} e_i z_i \right|, \tag{1}$$

where

$$e_1 = -\hat{b}_0 \left(13 + 7\sqrt{6}\right)/3, \quad e_2 = -\hat{b}_0 \left(13 - 7\sqrt{6}\right)/3, \quad e_3 = -\hat{b}_0/3 \tag{2}$$

and $\hat{b}_0 = 0.274888829595677$. Test your implementation.

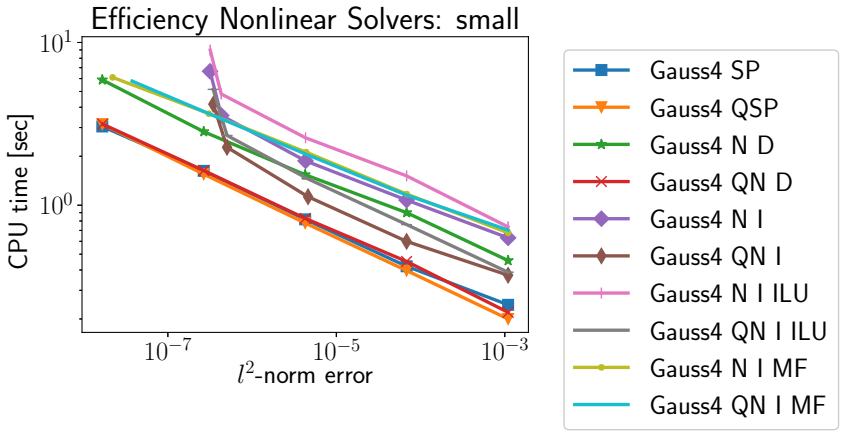
Run the `Grading/Adaptivity_*.py` scripts. You should get results similar to the ones of Fig. 2.

2.3. Nonlinear Solvers

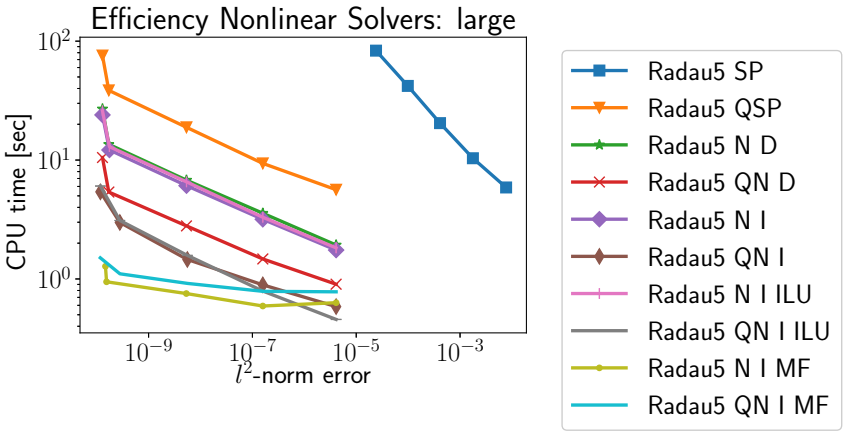
So far, we have used the black-box SciPy nonlinear solver. In fact, the `ImplicitRK.step` method calls the `ImplicitRK.scipy_solver` method (check the constructor), which in turn uses `scipy.optimize.fsolve`. Here we will implement the Newton algorithm by ourselves and explore the different ways of solving the nonlinear and linear systems.

For the next experiments, disable time step adaptivity.

i) (0.20 pts.) Finish the implementation of the Jacobian of Φ, hence the `ImplicitRK.dPhi` method. To test the method, you can enable the quasi-Newton option. In this case the `ImplicitRK.scipy_solver` method will use the just implemented `ImplicitRK.dPhi` method and evaluate the Jacobian only once.



(A) `NonlinearSolvers_small.py`



(B) `NonlinearSolvers_large.py`

FIGURE 3. Nonlinear solvers efficiency.

ii) (0.30 pts.) Here we finally implement the `ImplicitRK.Newton` method. To do so, you will have to complete the `ImplicitRK.call_linear_solver` method too. For the time being, consider only direct methods, hence also without preconditioners and with the matrix-free option disabled.
iii) (0.10 pts.) Complete `ImplicitRK.call_linear_solver`, hence add the iterative linear solver.
iv) (0.10 pts.) Complete `ImplicitRK.get_dPhi_preconditioner` and add the option for the ILU preconditioner. It will be used in the iterative solver when the Jacobian is available as a matrix (i.e. the matix-free option is disabled) and the `'preconditioner':'ILU'` option is passed.
v) (0.20 pts.) Finish the implementation of `ImplicitRK.dPhi_FD`, hence the approximate Jacobian operator $\tilde{\Phi}(\boldsymbol{z})\boldsymbol{v}$. Use it in `ImplicitRK.get_dPhi_info` to define `dPhi` for the matrix-free case.

Run the `Grading/NonlinearSolver_small.py` and `Grading/NonlinearSolver_large.py` scripts, you should get results similar to Fig. 3.

3. CONCLUSIONS

Project completed 🍾! Sit down and interpret the results ☕.