Andrea Landella
POLITECNICO DI MILANO

Implementation of an explicit 4-th order Runge-Kutta solver for ODE systems in C++

As an addendum for the practical sessions of Chemical Plants II, a custom solver was developed to solve many of the governing equation systems for the exercises at hand.

This solver is implemented in C++ and it is backward compatible in C, meaning that it compiles in any of the two languages interpreted by the compiler. The solver inputs are almost the same as MATLAB, as the initial and final time, initial condition, and the function handle. Some extras have been added as, the separation h or the integration step and the dimensionality of the problem:

```
//RK4 solver for a single equation:
rungekutta4(y0, t0, tf, h, &odesysfun);

//RK4 solver for an equation system with dimension DIM_sys:
rungekutta4_vec(y0, t0, tf, h, &odesysfun, DIM_sys);
```

and the ODE system or function `odesysfun` is supplied as a common function in C or C++, defined in a separate file recalled by a header file or declared before the main and defined after the main. The second solver degenerates to the first one by letting the dimension equal to 1. It is remarked that both solvers are *void functions*, meaning that they do not return any quantity or solution array on the main, but they only generate the solution txt file (solution arrays are printed there).

Since the internals of each solver implements dynamic allocation, it may be possible to address the result array of times and solutions by letting:

```
//RK4 solver for a single equation (NOT IMPLEMENTED):
rungekutta4(y0, t0, tf, h, &odesysfun, &y, &t);

//RK4 solver for an equation system with dimension DIM_sys (NOT IMPLEMENTED):
rungekutta4_vec(y0, t0, tf, h, &odesysfun, DIM_sys, &y, &t);
```

and retain the void nature of the function. However, this feature *is not implemented*, and the functions return nothing, only the solution txt file.

The 4-th order Runge-Kutta method for solving ODE systems

The solver's internal architecture is trivial, as it executes in the most explicit way the Runge-Kutta algorithm both for scalar ($DIM = 1$) and vector ($DIM > 1$) valued ODEs:

$$\frac{d\mathbf{y}(t)}{dt} = F(\mathbf{y}(t), t) \quad with \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad \mathbf{y} \in \mathbb{R}^{DIM}$$

with a set of operations with four slope estimations. Each estimation improves the guess on the next point, but each one is explicit, as explained[1].

The latter are functions of the integration step h, as follows:

$$\mathbf{k}_1 = F(\mathbf{y}(t_n), t_n)$$

$$\mathbf{k}_2 = F(\mathbf{y}(t_n) + \mathbf{k}_1 h/2, t_n + h/2)$$

$$\mathbf{k}_3 = F(\mathbf{y}(t_n) + \mathbf{k}_2 h/2, t_n + h/2)$$

[1] http://lpsa.swarthmore.edu/NumInt/NumIntIntro.html

$$\mathbf{k}_4 = F(\mathbf{y}(t_n) + \mathbf{k}_3 h, t_n + h)$$

Then, the solution at the next iteration is expressed as:

$$\mathbf{y}(t_{n+1}) = \mathbf{y}(t_n) + \frac{\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4}{6} h \quad with \quad t_{n+1} = t_n + h$$

and the results are printed in a separate text file.

Solving higher order ODEs and IDEs with `rungekutta4_vec`

Using the ODE system solver, it is also possible to solve higher order ODEs and IDEs, for instance consider this IDE IVP on a single variable:

$$y'' - 4y' + y\sin(t) - \int_{t_0}^{t} y(\tau)d\tau = t^2 + 1 \quad with \quad y(t_0) = y_0 \;\wedge\; y'(t_0) = y_0'$$

This can be seen as a second order nonlinear system with an integral control action, or generally a 2 + 1 order ODE system, as the presence of the integral increases the order by one unit.

We can use substitution and variable change, letting 2 + 1 = 3 auxiliary functions, starting from:

$$Y_1 = \int_{t_0}^{t} y(\tau)d\tau$$

and by sequentially applying derivatives of Y_1 we have a new set of ODEs:

$$Y_2 = \frac{dY_1}{dt} = \frac{d}{dt}\int_{t_0}^{t} y(\tau)d\tau = y$$

$$Y_3 = \frac{dY_2}{dt} = \frac{dy}{dt} = y'$$

Substituting into the auxiliary functions into the original equation we have transformed an IDE into an ODE, and the solver can process constant and nonconstant coefficients:

$$\frac{dY_3}{dt} - 4Y_3 + Y_2\sin(t) - Y_1 = t^2 + 1$$

The resulting system is 3 × 3, expressed as follows:

$$\begin{cases} \dfrac{dY_3}{dt} = 4Y_3 - Y_2\sin(t) + Y_1 + t^2 + 1 \\ \dfrac{dY_2}{dt} = Y_3 \\ \dfrac{dY_1}{dt} = Y_2 \end{cases}$$

with modified initial conditions, that are obtained from the original initial conditions with the auxiliary functions declared above:

$$Y_1(t_0) = \int_{t_0}^{t_0} y(\tau)d\tau = 0 \quad \wedge \quad Y_2(t_0) = y_0 \quad \wedge \quad Y_3(t_0) = y_0'$$

Andrea Landella
POLITECNICO DI MILANO

Performance test for ODEs with MATLAB `ode23`**,** `ode45`**,** `ode113`

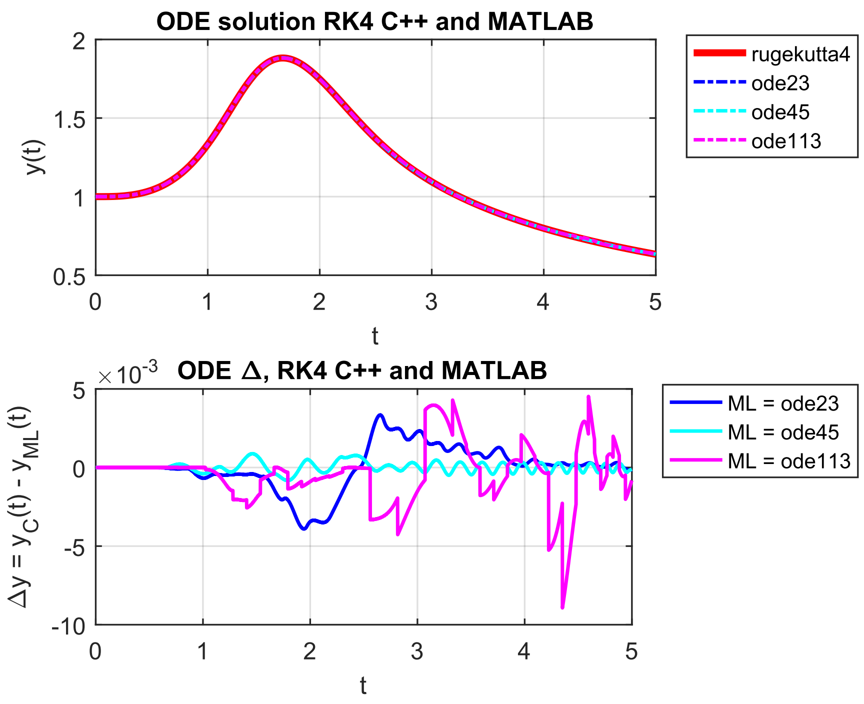
We test the solvers behaviour on test functions which are benchmarked with MATLAB non-stiff solvers, with the same integration step (same number of integration points) for each function or system. Such test functions are chosen to be *nonlinear* and are provided within the main file as normal C/C++ functions. The reason lies in the generality and applicability of the new C++ solver, as it is tested conservatively to a wider and harder class of problems than linear ones. Once the C++ code is compiled and executed in debug mode, the txt file is read and printed by MATLAB to be consistent with the results' format of odexx.

The test function for `rungekutta4` is:

$$\frac{dy}{dt} = t\sin(yt) \quad with \quad y(0) = 1, \quad h = 0.001, \quad t_1 = 5$$

the calculation times for C++ and MATLAB are almost equal, with C++ slightly faster than MATLAB. The total integration points are 5000, and the execution time is roughly 0.09 s on an 8 GB RAM, 4 i-5 CPU machine.

The results are expressed in the form of the explicit solution and the differences between the MATLAB and C++ solution for each solver:



In the second subplot, the line color indicates the type of MATLAB solver referenced to in the difference, so for instance, blue will indicate the difference from ode23. The accuracy may be acceptable when a rough estimate of the solution may be needed, however the method's correctness cannot be evaluated. We can list the statistical performance parameters of each difference:

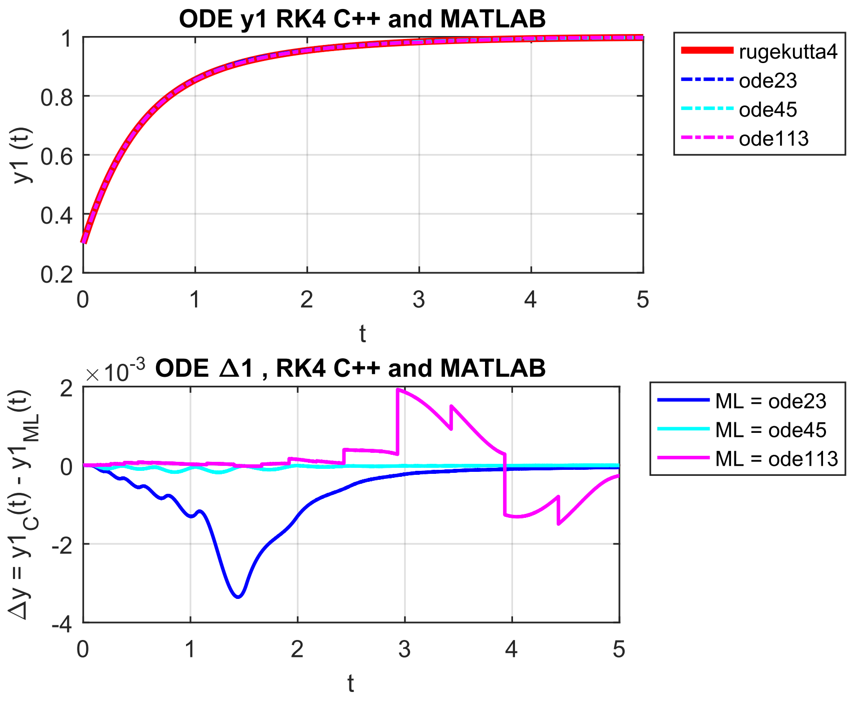
MATLAB ODE solver	**μ(Δ)**	**σ(Δ)**
ode113	-4.4673e-04	1.9257e-03
ode23	+7.7249e-06	1.4260e-03
ode45	+1.9744e-05	3.2066e-04

The test function for `rungekutta4_vec` is:

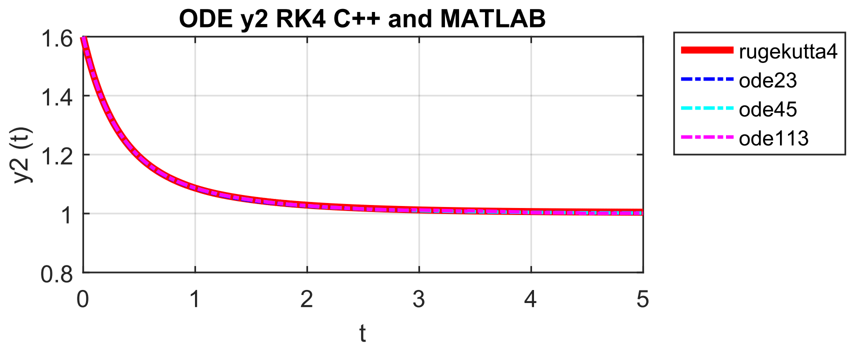
$$\begin{cases} \dfrac{dy_4}{dt} = y_1 - y_4^2 \\ \dfrac{dy_3}{dt} = y_2 - y_3^2 \\ \dfrac{dy_2}{dt} = y_3 - y_2^2 \\ \dfrac{dy_1}{dt} = y_4 - y_1^2 \end{cases} \quad with \quad \mathbf{y}(0) = \begin{pmatrix} 0.3 \\ 1.6 \\ 0.9 \\ 1.3 \end{pmatrix}, \quad h = 0.0005, \quad t_1 = 5$$

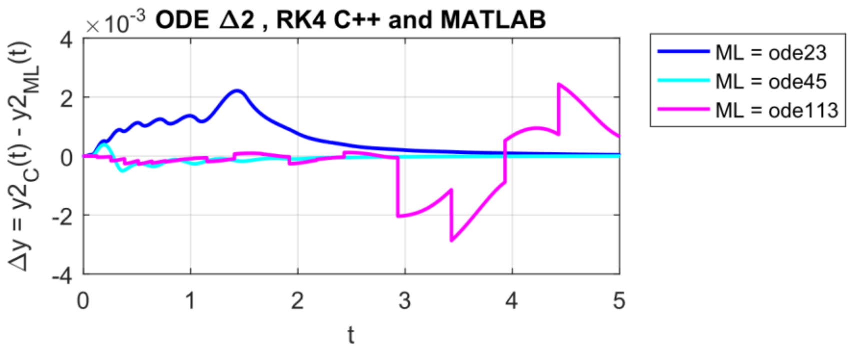
the calculation times for C++ and MATLAB are almost equal, with MATLAB slightly faster than C++. The total integration points are 10000, and the execution time is roughly 0.21 s on an 8 GB RAM, 4 i-5 CPU machine.

The results are indicated per component or function of the system, for y1:

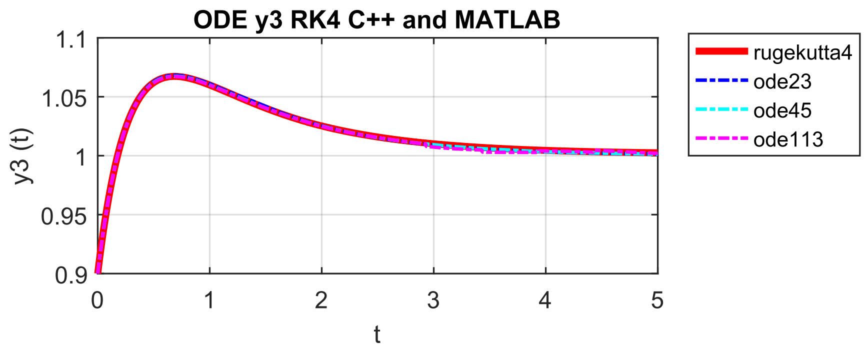


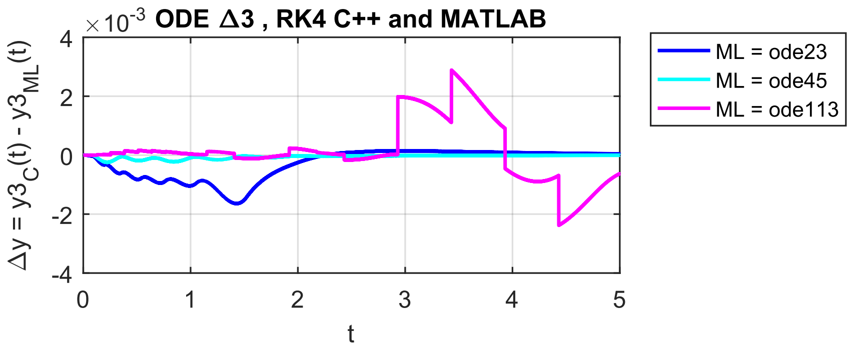
while for y2, we have:



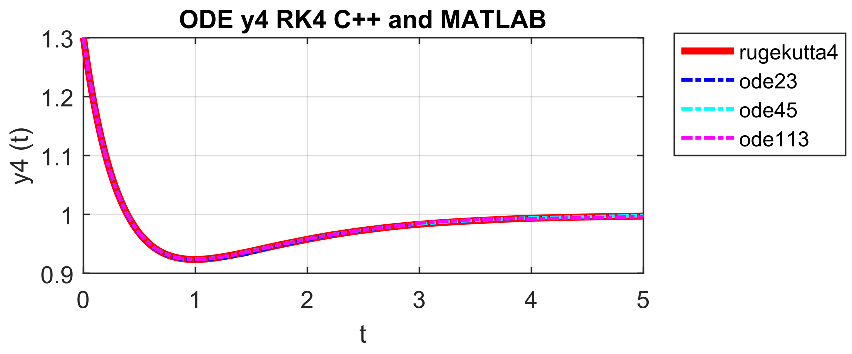


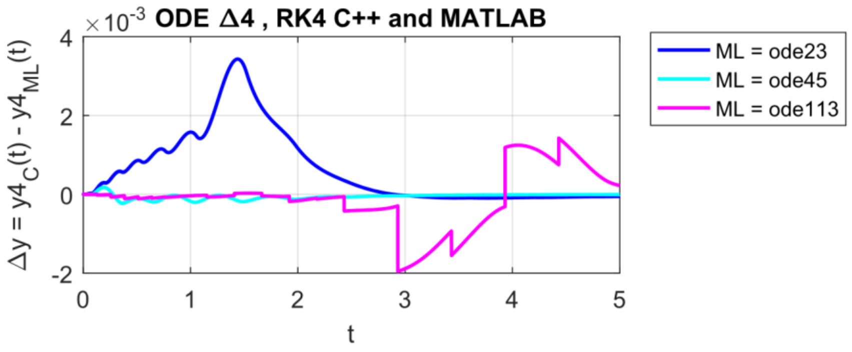
While for y3, we have again a specular behavior, both in the function values and in the errors relating to the MATLAB solver. Up to now, we notice a very low difference with respect to ode45:





While for y4 we have:





In the second subplot, the line color indicates the type of MATLAB solver referenced to in the difference, so for instance, blue will indicate the difference from ode23. The accuracy may be acceptable when a rough estimate of the solution may be needed, however the method's correctness cannot again be evaluated.

We can list the statistical performance parameters of each difference:

MATLAB ODE solver	**μ [μ(Δi)]**	**σ [σ(Δi)]**
ode113	-1.0211e-05	1.6117e-04
ode23	+6.2441e-05	1.9941e-04
ode45	-5.5410e-05	3.2669e-05

Using the two tables, we identify that the proposed solver has the closest performance and behavior to ode45, which is itself a Runge-Kutta 4/5 solver. This is proof of the solver equivalence to the MATLAB solver, and thus it is possible to translate its potentialities and limitations in C/C++ problems.