Andrea Landella
POLITECNICO DI MILANO

**Introducing the Explicit 4-th Order Runge-Kutta C++ Solver**

Practical sessions of Chemical Plants II involve the solution of complex ODEs and NLEs to model a required equipment or process. The required calculation framework is Visual Studio (Visual C++), with custom built external libraries such as `BzzMath`. However, oftentimes the latter could not be loaded into new versions of Visual Studio, and thus some time had to be devolved to troubleshooting such issue. The root cause in most cases is user dependent, for example the current OS, installed programs or kernel settings could generate conflict. These causes are thus difficult to discover and repair. In order to save time, if the above problems arise, it is now possible to solve the practical's exercises using the same syntax as MATLAB odexx commands.

A new, custom ODE solver was developed to solve many of the governing equation systems at hand. It is implemented in C++ and it is backward compatible in C, meaning that it can be compiled in any of the two languages. The solver consists of a source `rk4solver.cpp` and a header `rk4solver.h`, and they have to be included in the respective source and header directories of any new Visual Studio project. To use all the functions created for the solver, the header must be included in the main file, as the sample `rk4main.cpp` file states. The package comprises 3 files overall, 2 for the solver and 1 for the sample main.

The solver's functions are two Runge-Kutta 4th order algorithms that solve 1-D equations and DIM-D systems of equations, and are called as `void` functions, meaning the solutions `tSOL` and `ySOL` are treated as inputs:

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

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

Each function structure is almost identical to MATLAB `odexx` functions, as first the function handle is supplied, then the time span, initial condition and lastly additional options. The function handles refer to common C++ functions that describe the ODE or ODE system that needs to be solved: they can be defined in a separate file recalled by a header or declared before the main and defined after the main. Each function produces a txt file where the vector `tSOL` and vector/matrix `ySOL` is printed.

**Mathematical basis of the 4-th order Runge-Kutta method**

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

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

The algorithm consists of finding the explicit solution value $\mathbf{y}(t_{n+1})$ at the next time value $t_{n+1}$ using four slope estimations $\mathbf{k}_j$, which improve the guess on the next point. Each $\mathbf{k}_j$ is a function of the integration step $h$:

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

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

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

$$\mathbf{k}_4 = F(\mathbf{y}(t_n) + \mathbf{k}_3 h, t_{n+1})$$

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

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

Each solution at each time step will be printed in the txt file. The above derivation holds for any dimension, and for vector ($D > 1$) valued ODEs each component must be pointwise summed.

**Solving generic ODEs and IDEs with `rungekutta4_vec`**

Using the ODE system solver, it is also possible to solve generic order ODEs and IDEs using the substitution method. IDE are of paramount importance in process control theory, since they describe a class of automatic controllers known as Integral-type. The proposed solver can tackle such problems if and only if the general problem is reformulated. It is possible to formally describe the consistency of the substitution method, but for simplicity an example will be provided. Considering the second order governing IDE, as IVP:

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

It is possible to translate this IDE into a 2 + 1 order ODE system, as the presence of the integral increases the order by one unit. Thus, a total of 2 + 1 = 3 auxiliary functions can be constructed, 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:

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

$$Y_3 = \frac{dY_2}{dt} \quad \rightarrow \quad Y_3 = \frac{d}{dt}\frac{dY_1}{dt} = \frac{dy}{dt} = y'$$

substituting into the auxiliary functions into the original equation we have the 3-D ODE system, as the first equation is the original one, when all lower order terms are shifted to the right-hand side:

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

The new initial conditions are obtained from the original ones with the auxiliary functions declared above:

$$Y_1(t_0) = 0 \quad \wedge \quad Y_2(t_0) = y_0 \quad \wedge \quad Y_3(t_0) = y_0'$$

and the reformulated problem is complete. The original solution is now translated to $Y_2(t) = y(t)$.

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

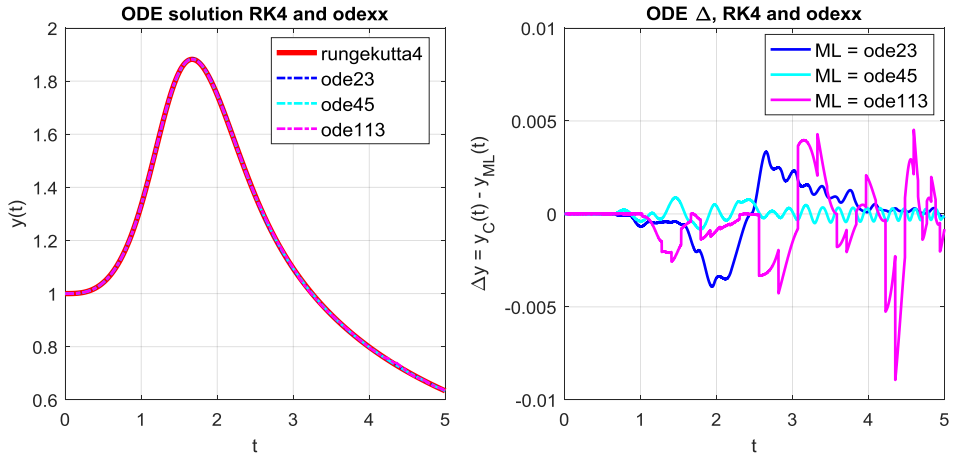
We test the solvers' behaviour on test functions which are benchmarked with MATLAB non-stiff solvers, with the same integration step (same number of integration points) for each function or system. Such test functions are chosen to be nonlinear and are provided within the `rk4main.cpp` file as common C++ functions. The

reason lies in the generality and applicability of the 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 the 1D ODE solver, `rungekutta4` is expressed as follows:

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

with 5000 integration points. The calculation times for `rungekutta4` and `odexx` are almost equal, with the former being slightly faster than MATLAB as the execution time is roughly 0.09 s on an 8 GB RAM, 4 i-5 CPU machine. The results are expressed as explicit solutions and the absolute differences between the `odexx` and `rungekutta4` solution for each solver:




In the second subplot, the line color indicates the `odexx` solver referenced to in the difference. For example, 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 |

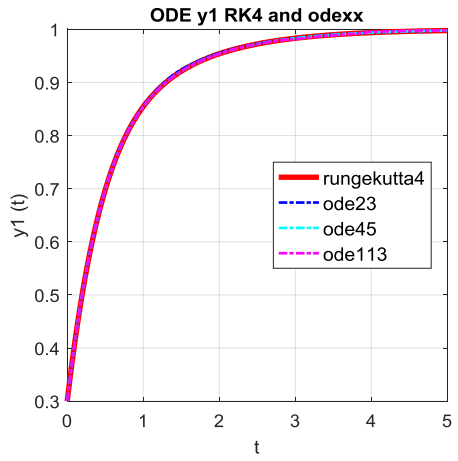
and conclude that the performance of `rungekutta4` and `ode45` are similar, as the absolute mean error and deviations are sufficiently small.

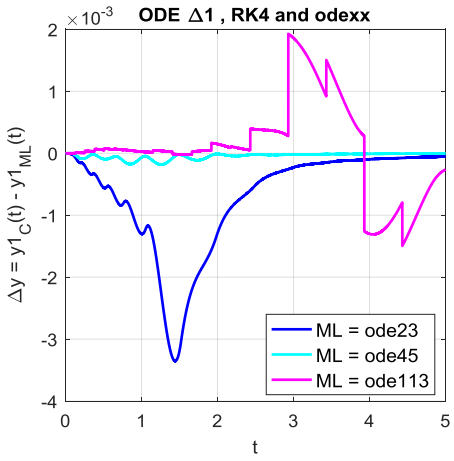
- The test function for the DIM-D ODE solver, `rungekutta4_vec` is expressed as follows:

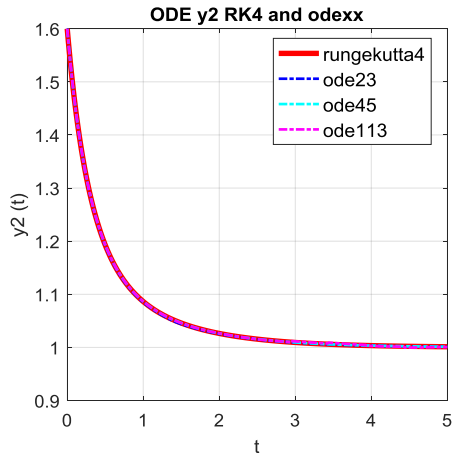
$$\begin{Bmatrix} \frac{dy_1}{dt} = y_4 - y_1^2 & \frac{dy_3}{dt} = y_2 - y_3^2 \\ \frac{dy_2}{dt} = y_3 - y_2^2 & \frac{dy_4}{dt} = y_1 - y_4^2 \end{Bmatrix} \quad with \quad \mathbf{y}(0) = \begin{pmatrix} 0.3 \\ 1.6 \\ 0.9 \\ 1.3 \end{pmatrix}, \quad h = 0.0005, \quad t_1 = 5$$

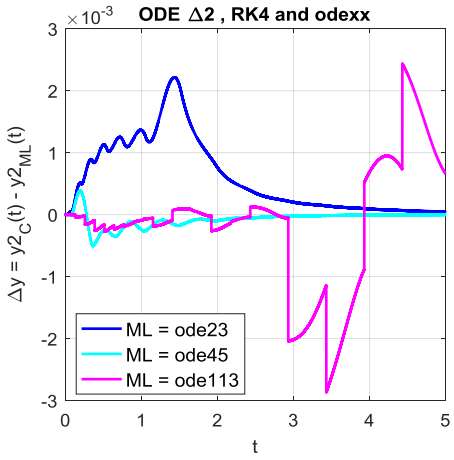
with 10000 integration points. The times for `rungekutta4_vec` and `odexx` are almost equal, with the

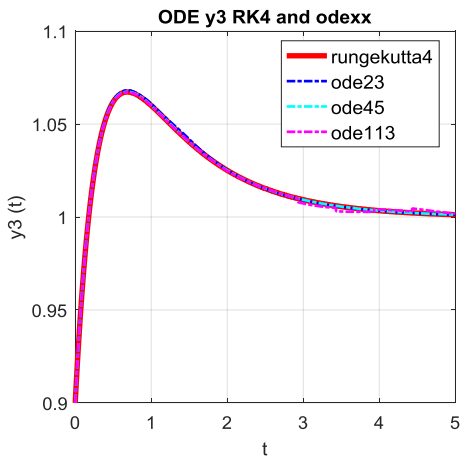
former being slightly faster than MATLAB as the execution time is roughly 0.21 s on an 8 GB RAM, 4 i-5 CPU machine. The explicit solutions and the absolute differences of `odexx` and `rungekutta4` are reported:

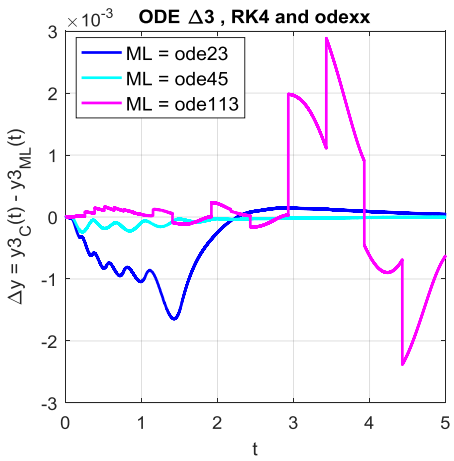


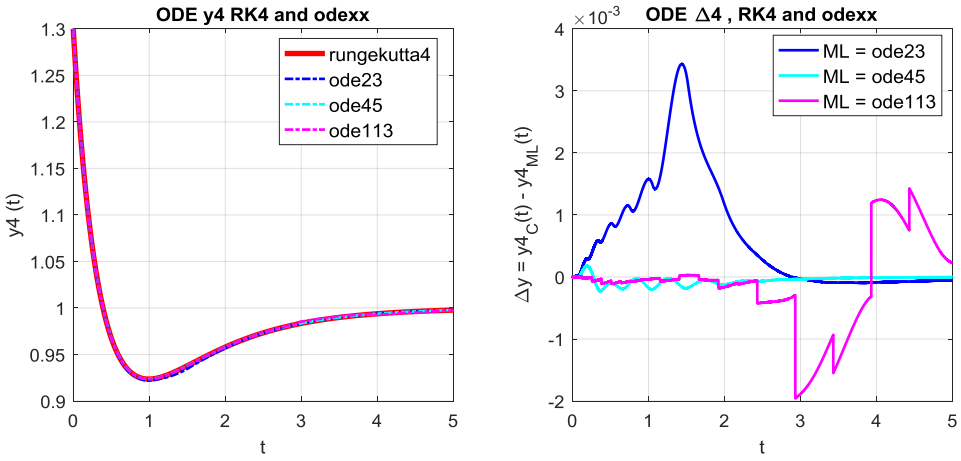



Andrea Landella
POLITECNICO DI MILANO




In the second subplot, the line color indicates the `odexx` solver referenced to in the difference. For example, 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** | -1.0211e-05 | 1.6117e-04 |
| **ode23** | 6.2441e-05 | 1.9941e-04 |
| **ode45** | 5.5410e-05 | 3.2669e-04 |

and conclude that the performance of `rungekutta4` and `ode45` are also similar, as the absolute mean error and deviations are sufficiently small.

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