Solving ordinary differential equations with multi-precision libraries

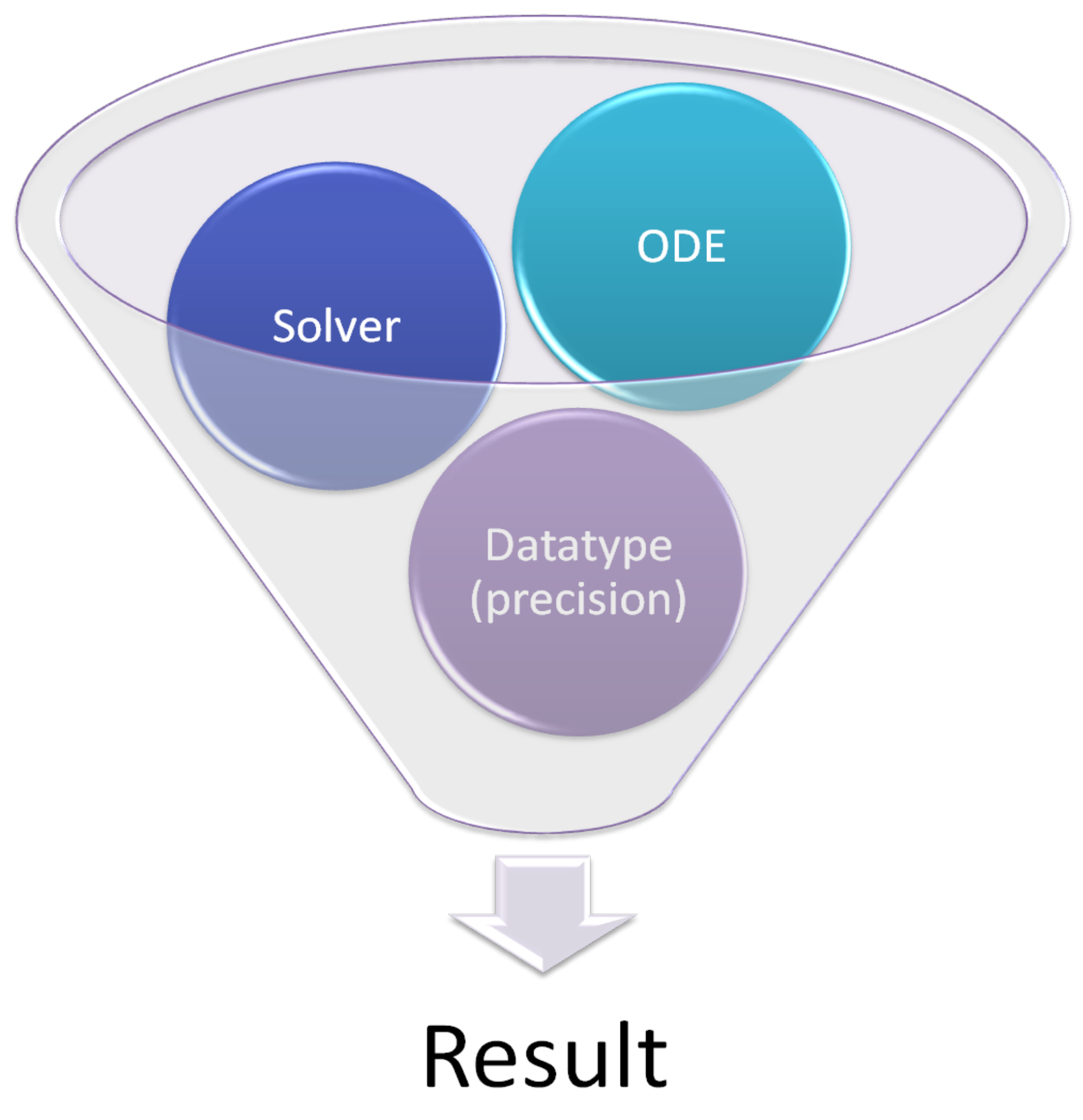
Martin Ettl[1], Manfred Schneider[1], Urs Hugentobler[1]
([1]FESG Wettzell)

Abstract:

Modern earth observation techniques require a precise knowledge about the position and velocity of observed satellites or other objects in space. Computing the position analytically does not provide the needed accuracy anymore, due to a missing analytical high accurate orbital-theory. In order to gain accuracy, it is common to compute an orbit by solving ordinary differential equations (ODEs). Solving this kind of mathematical equations leads to well tested standard-methods like Runge-Kutta-methods, Burlisch-Stoer, symplectic or power-series integrators. These solvers have been implemented using C++-templates allowing to change the floating-point data type at compile time. Therefore multi-precision data types with a free-to-choose decimal precision can be used. Based on this approach, each numerical solver can operate with variable internal precision. This, for instance, makes it possible to reveal round-off errors or missing accuracies by simply increase the precision of the underlying data type. It can be used to verify computed or measured results with, so far not available numerical accuracy. Solving an ODE with high accuracy using a multi-precision library requires more CPU-cycles. This is why the implemented algorithms has been profiled and highly optimized to avoid wasting CPU-cycles on our testing platforms.

Design and implementation

Numerically solving an Ordinary Differential Equation (ODE), seen from the programming point of view, is a task that can be split up into three parts. First of all an algorithm is needed, capable of solving an ODE. Each of this solvers has its own characteristic and therefore their specific pros and cons. Nevertheless, a standardized interface has been created using object-oriented inheritance, which allows to plug-in any ODE into the solver. This design makes it possible to extend the available ODEs without touching the already well tested implementation of the solving-algorithm. The ODE can be implemented by simply deriving from a base class containing pure-virtual functions that must be implemented by the derived class. This separation between algorithm and mathematical problem into separated and independent components massively improves the re-usability of the software components. In general, all algorithms are implemented with basic datatypes, having a fixed precision. This is why all the modules have been implemented as C++-template classes, offering a way to implement the classes without this datatype dependency. Based on this, the corresponding standard - or multi-precision datatype, used for the computation can be selected at compile-time. In case of multi-precision, most of the used datatypes are capable of changing their internal decimal precision during runtime.



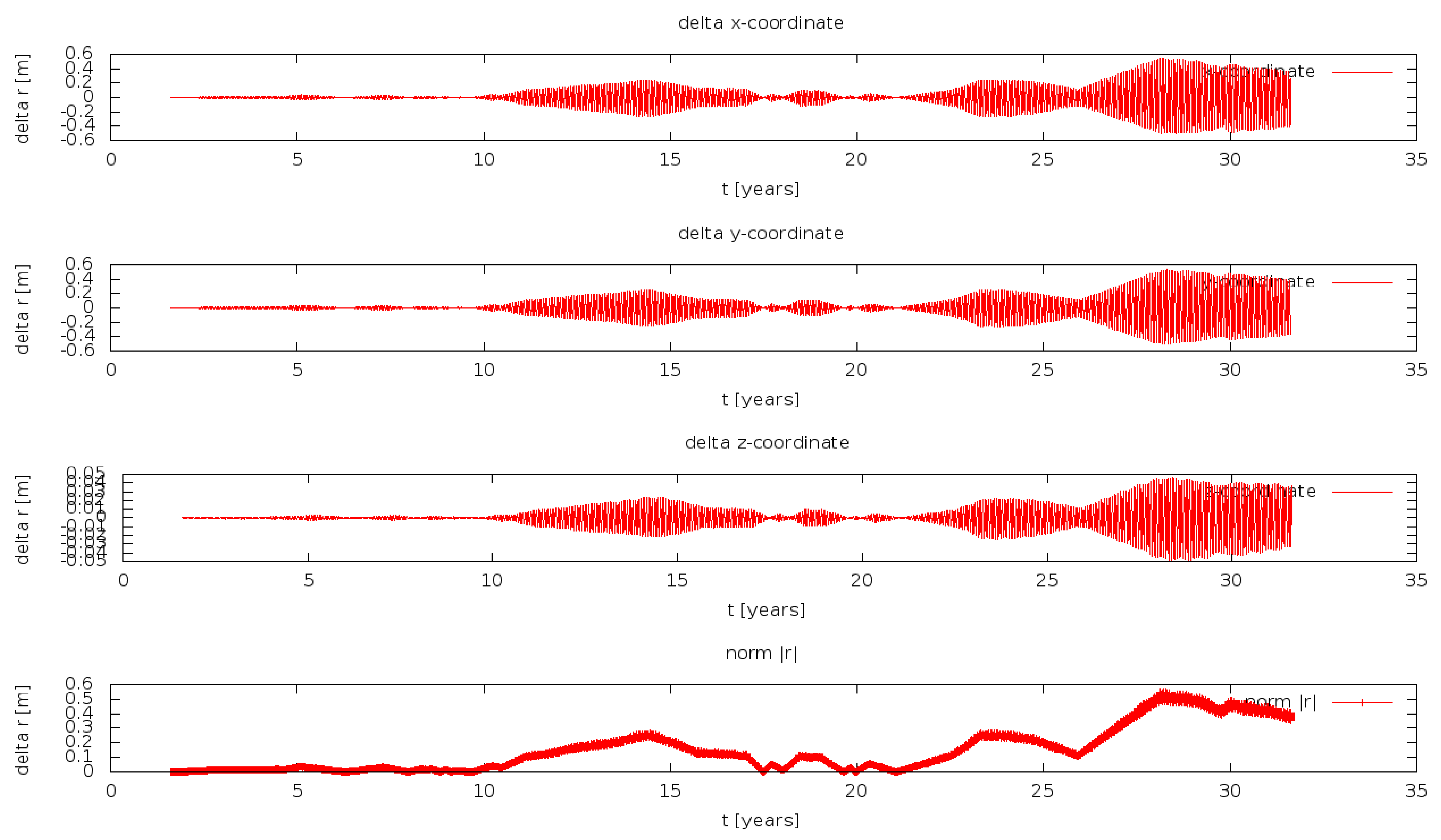
solver	categorization						
	adaptive stepsize control	ODE order		single step	multistep	power series	symplectic
		1	2				
Shampine Gordon (SG)	✓	✓			✓		
Burlisch Stoer (BS)	✓	✓			✓		
Gauss-Jackson (GJ4)			✓	✓			
Runge Kutta 4.order (RK4)		✓		✓			
Runge Kutta 10.order (RK10)		✓		✓			
Dormand-Prince (DOPRI)	✓	✓		✓			
Leap-Frog (LF)		✓		✓			✓
Symplectic (SYMP) 4.-, 6.-and 8.order		✓		✓			✓
Power series (POWSER)		✓		✓		✓	

Categorization of solvers

In the table (left), the implemented solvers are listed and grouped by their specific features. Each solving-algorithm has its own characteristics. Some of the solvers have a built-in adaptive step-size control mechanism. This means, before the integration starts, the user can set an absolute and relative error-bound. According to this information, the integration-solver chooses the step-size automatically. Most of the algorithms are designed to solve ODEs of first order, because a ODE of higher order can be transformed into a system of first order ODEs. Nevertheless the GJ4 algorithm has been implemented, capable of solving second order ODEs. Computing the next time-step (solving an ODE) can be done using different approaches. The single-step methods use only data from the last step, whereas the multi-step methods take into account former time-steps. The power series methods create and solve at each integration step a power-series, according to recursive laws of power-series composition. In this case, the order of the series is not fixed and can be set by the user. The symplectic solvers are designed to be more energy conserving than others.

Results of long time evolution of Earth-Moon distance

In the plot (right) the results of a comparison of two different numerical simulations is shown. Both simulations where computed using the BS-solver with adaptive step-size control. Furthermore identical settings and initial values where used to compute the orbit based on the LiDIA - multi-precision datatype. The first solution was computed with a decimal precision of 16 significant digits and the second with 38 significant digits. The plot shows the absolute errors for each coordinates and the distance along the integration time of 32 years. The plot shows the absolute error in each coordinate starting to oscillate after approximate ten years integration time. The absolute error in range increases after approx. 12 years, decreases and then accumulates to 0.5 m at approx. 28 years. During the integration the ODE-function was called ~300000 times. This possibly explains the increase of round-off errors, due to millions of floating point operations over the whole integration. This possibility of checking for round-off errors using multi-precision methods is a very helpful technique to verify results.


delta x-coordinate
delta r [m]
t [years]
x-coordinate
delta y-coordinate
y-coordinate
delta z-coordinate
z-coordinate
norm |r|
0
5
10
15
20
25
30
35