Boost.Numeric.Odeint

Karsten Ahnert

Mario Mulansky

Table of Contents

Getting started ... 3
- Overview ... 3
- Usage, Compilation, Headers ... 7
- Short Example ... 8

Tutorial ... 10
- Harmonic oscillator ... 10
- Solar system ... 13
- Chaotic systems and Lyapunov exponents ... 17
- Stiff systems ... 20
- Complex state types ... 22
- Lattice systems ... 23
- Ensembles of oscillators ... 25
- Using boost::units ... 28
- Using matrices as state types ... 30
- Using arbitrary precision floating point types ... 32
- Self expanding lattices ... 33
- Using CUDA (or OpenMP, TBB, ...) via Thrust ... 36
- Using OpenCL via VexCL ... 45
- All examples ... 47

odeint in detail ... 50
- Steppers ... 50
- Generation functions ... 67
- Integrate functions ... 69
- State types, algebras and operations ... 71
- Using boost::ref ... 81
- Using boost::range ... 81
- Binding member functions ... 83

Concepts ... 85
- System ... 85
- Symplectic System ... 85
- Simple Symplectic System ... 86
- Implicit System ... 87
- Stepper ... 88
- Error Stepper ... 89
- Controlled Stepper ... 91
- Dense Output Stepper ... 93
- State Algebra Operations ... 94
- State Wrapper ... 98

Literature ... 99

Acknowledgments ... 100

odeint Reference ... 101
- Header <boost/numeric/odeint/integrate/integrate.hpp> ... 101

Header <boost/numeric/odeint/integrate/integrate_adaptive.hpp> 102
Header <boost/numeric/odeint/integrate/integrate_const.hpp> 104
Header <boost/numeric/odeint/integrate/integrate_n_steps.hpp> 105
Header <boost/numeric/odeint/integrate/integrate_times.hpp> 106
Header <boost/numeric/odeint/integrate/null_observer.hpp> 107
Header <boost/numeric/odeint/integrate/observer_collection.hpp> 108
Header <boost/numeric/odeint/stepper/adams_bashforth.hpp> 109
Header <boost/numeric/odeint/stepper/adams_bashforth_moulton.hpp> 115
Header <boost/numeric/odeint/stepper/adams_moulton.hpp> 119
Header <boost/numeric/odeint/stepper/base/algebra_stepper_base.hpp> 122
Header <boost/numeric/odeint/stepper/base/explicit_error_stepper_base.hpp> 123
Header <boost/numeric/odeint/stepper/base/explicit_error_stepper_fsal_base.hpp> 131
Header <boost/numeric/odeint/stepper/base/explicit_stepper_base.hpp> 141
Header <boost/numeric/odeint/stepper/base/symplectic_rkn_stepper_base.hpp> 147
Header <boost/numeric/odeint/stepper/bulirsch_stoer.hpp> 153
Header <boost/numeric/odeint/stepper/bulirsch_stoer_dense_out.hpp> 158
Header <boost/numeric/odeint/stepper/controlled_runge_kutta.hpp> 164
Header <boost/numeric/odeint/stepper/controlled_step_result.hpp> 175
Header <boost/numeric/odeint/stepper/dense_output_runge_kutta.hpp> 175
Header <boost/numeric/odeint/stepper/euler.hpp> 182
Header <boost/numeric/odeint/stepper/explicit_error_generic_rk.hpp> 187
Header <boost/numeric/odeint/stepper/explicit_generic_rk.hpp> 195
Header <boost/numeric/odeint/stepper/implicit_euler.hpp> 200
Header <boost/numeric/odeint/stepper/modified_midpoint.hpp> 202
Header <boost/numeric/odeint/stepper/rosenbrock4.hpp> 208
Header <boost/numeric/odeint/stepper/rosenbrock4_controller.hpp> 212
Header <boost/numeric/odeint/stepper/rosenbrock4_dense_output.hpp> 214
Header <boost/numeric/odeint/stepper/runge_kutta4.hpp> 217
Header <boost/numeric/odeint/stepper/runge_kutta4_classic.hpp> 222
Header <boost/numeric/odeint/stepper/runge_kutta_cash_karp54.hpp> 227
Header <boost/numeric/odeint/stepper/runge_kutta_cash_karp54_classic.hpp> 234
Header <boost/numeric/odeint/stepper/runge_kutta_dopri5.hpp> 241
Header <boost/numeric/odeint/stepper/runge_kutta_fehlberg78.hpp> 250
Header <boost/numeric/odeint/stepper/stepper_categories.hpp> 257
Header <boost/numeric/odeint/stepper/symplectic_euler.hpp> 262
Header <boost/numeric/odeint/stepper/symplectic_rkn_sb3a_m4_mclachlan.hpp> 266
Header <boost/numeric/odeint/stepper/symplectic_rkn_sb3a_mclachlan.hpp> 271
Indexes 276

Getting started

Overview

Caution

Boost.Numeric.Odeint is not an official boost library!

odeint is a library for solving initial value problems (IVP) of ordinary differential equations. Mathematically, these problems are formulated as follows:

x'(t) = f(x,t), x(0) = x0.

x and *f* can be vectors and the solution is some function *x(t)* fulfilling both equations above. In the following we will refer to *x'(t)* also `dxdt` which is also our notation for the derivative in the source code.

Ordinary differential equations occur nearly everywhere in natural sciences. For example, the whole Newtonian mechanics are described by second order differential equations. Be sure, you will find them in every discipline. They also occur if partial differential equations (PDEs) are discretized. Then, a system of coupled ordinary differential occurs, sometimes also referred as lattices ODEs.

Numerical approximations for the solution *x(t)* are calculated iteratively. The easiest algorithm is the Euler scheme, where starting at *x(0)* one finds *x(dt) = x(0) + dt f(x(0),0)*. Now one can use *x(dt)* and obtain *x(2dt)* in a similar way and so on. The Euler method is of order 1, that means the error at each step is ~ dt^2. This is, of course, not very satisfying, which is why the Euler method is rarely used for real life problems and serves just as illustrative example.

The main focus of odeint is to provide numerical methods implemented in a way where the algorithm is completely independent on the data structure used to represent the state *x*. In doing so, odeint is applicable for a broad variety of situations and it can be used with many other libraries. Besides the usual case where the state is defined as a `std::vector` or a `boost::array`, we provide native support for the following libraries:

- Boost.uBLAS
- Thrust, making odeint naturally running on CUDA devices
- gsl_vector for compatibility with the many numerical function in the GSL
- Boost.Range
- Boost.Fusion (the state type can be a fusion vector)
- Boost.Units
- Intel Math Kernel Library for maximum performance
- VexCL for OpenCL
- Boost.Graph (still experimentally)

In odeint, the following algorithms are implemented:

Table 1. Stepper Algorithms

Algorithm	Class	Concept	System Concept	Order	Error Estimation	Dense Output	Internal state	Remarks
Explicit Euler	`euler`	Dense Output Stepper	System	1	No	Yes	No	Very simple, only for demonstrating purpose
Modified Midpoint	`modified_midpoint`	Stepper	System	configurable (2)	No	No	No	Used in Bulirsch-Stoer implementation
Runge-Kutta 4	`runge_kutta4`	Stepper	System	4	No	No	No	The classical Runge-Kutta scheme, good general scheme without error control
Cash-Karp	`runge_kutta_cash_karp54`	Error Stepper	System	5	Yes (4)	No	No	Good general scheme with error estimation, to be used in controlled_error_stepper
Dormand-Prince 5	`runge_kutta_dopri5`	Error Stepper	System	5	Yes (4)	Yes	Yes	Standard method with error control and dense output, to be used in controlled_error_stepper and in dense_output_controlled_explicit_fsal.
Fehlberg 78	`runge_kutta_fehlberg78`	Error Stepper	System	8	Yes (7)	No	No	Good high order method with error estimation, to be used in controlled_error_stepper.

Algorithm	Class	Concept	System Concept	Order	Error Estimation	Dense Output	Internal state	Remarks
Adams Bashforth	adams_bashforth	Stepper	System	configurable	No	No	Yes	Multistep method
Adams Moulton	adams_moulton	Stepper	System	configurable	No	No	Yes	Multistep method
Adams Bashforth Moulton	adams_bashforth_moulton	Stepper	System	configurable	No	No	Yes	Combined multistep method
Controlled Runge-Kutta	controlled_runge_kutta	Controlled Stepper	System	depends	Yes	No	depends	Error control for Error Stepper. Requires an Error Stepper from above. Order depends on the given Error-Stepper
Dense Output Runge-Kutta	dense_output_runge_kutta	Dense Output Stepper	System	depends	No	Yes	Yes	Dense output for Stepper and Error Stepper from above if they provide dense output functionality (like euler and runge_kutta_dopri5) Order depends on the given stepper.
Bulirsch-Stoer	bulirsch_stoer	Controlled Stepper	System	variable	Yes	No	No	Stepper with step size and order control. Very good if high precision is required.

Algorithm	Class	Concept	System Concept	Order	Error Estimation	Dense Output	Internal state	Remarks
Bulirsch-Stoer Dense Output	bulirsch_stoer_dense_out	Dense Output Stepper	System	variable	Yes	Yes	No	Stepper with step size and order control as well as dense output. Very good if high precision and dense output is required.
Implicit Euler	implicit_euler	Stepper	Implicit System	1	No	No	No	Basic implicit routine. Requires the Jacobian. Works only with BoostuBLAS vectors as state types.
Rosenbrock 4	rosenbrock4	Error Stepper	Implicit System	4	Yes	Yes	No	Good for stiff systems. Works only with BoostuBLAS vectors as state types.
Controlled Rosenbrock 4	rosenbrock4_controller	Controlled Stepper	Implicit System	4	Yes	Yes	No	Rosenbrock 4 with error control. Works only with BoostuBLAS vectors as state types.
Dense Output Rosenbrock 4	rosenbrock4_dense_output	Dense Output Stepper	Implicit System	4	Yes	Yes	No	Controlled Rosenbrock 4 with dense output. Works only with BoostuBLAS vectors as state types.

Algorithm	Class	Concept	System Concept	Order	Error Estimation	Dense Output	Internal state	Remarks
Symplectic Euler	`symplectic_euler`	Stepper	Symplectic System Simple Symplectic System	1	No	No	No	Basic symplectic solver for separable Hamiltonian system
Symplectic RKN McLachlan	`symplectic_rkn_sb3a_mclachlan`	Stepper	Symplectic System Simple Symplectic System	4	No	No	No	Symplectic solver for separable Hamiltonian system with 6 stages and order 4.
Symplectic RKN McLachlan	`symplectic_rkn_sb3a_m4_mclachlan`	Stepper	Symplectic System Simple Symplectic System	4	No	No	No	Symplectic solver with 5 stages and order 4, can be used with arbitrary precision types.

Usage, Compilation, Headers

odeint is a header-only library, no linking against pre-compiled code is required. It can be included by

```
#include <boost/numeric/odeint.hpp>
```

which includes all headers of the library. All functions and classes from odeint live in the namespace

```
using namespace boost::numeric::odeint;
```

It is also possible to include only parts of the library. This is the recommended way since it saves a lot of compilation time.

- `#include <boost/numeric/odeint/stepper/XYZ.hpp>` - the include path for all steppers, XYZ is a placeholder for a stepper.
- `#include <boost/numeric/odeint/algebra/XYZ.hpp>` - all algebras.
- `#include <boost/numeric/odeint/util/XYZ.hpp>` - the utility functions like `is_resizeable`, `same_size`, or `resize`.
- `#include <boost/numeric/odeint/integrate/XYZ.hpp>` - the integrate routines.
- `#include <boost/numeric/odeint/iterator/XYZ.hpp>` - the range and iterator functions.
- `#include <boost/numeric/odeint/external/XYZ.hpp>` - any binders to external libraries.

Short Example

Imaging, you want to numerically integrate a harmonic oscillator with friction. The equations of motion are given by $x'' = -x + \gamma x'$. Odeint only deals with first order ODEs that have no higher derivatives than x' involved. However, any higher order ODE can be transformed to a system of first order ODEs by introducing the new variables $q=x$ and $p=x'$ such that $w=(q,p)$. To apply numerical integration one first has to design the right hand side of the equation $w' = f(w) = (p,-q+\gamma p)$:

```
/* The type of container used to hold the state vector */
typedef std::vector< double > state_type;

const double gam = 0.15;

/* The rhs of x' = f(x) */
void harmonic_oscillator( const state_type &x , state_type &dxdt , const double /* t */ )
{
    dxdt[0] = x[1];
    dxdt[1] = -x[0] - gam*x[1];
}
```

Here we chose `vector<double>` as the state type, but others are also possible, for example `boost::array<double,2>`. odeint is designed in such a way that you can easily use your own state types. Next, the ODE is defined which is in this case a simple function calculating $f(x)$. The parameter signature of this function is crucial: the integration methods will always call them in the form `f(x, dxdt, t)` (there are exceptions for some special routines). So, even if there is no explicit time dependence, one has to define `t` as a function parameter.

Now, we have to define the initial state from which the integration should start:

```
state_type x(2);
x[0] = 1.0; // start at x=1.0, p=0.0
x[1] = 0.0;
```

For the integration itself we'll use the `integrate` function, which is a convenient way to get quick results. It is based on the error-controlled `runge_kutta54_cash_karp` stepper (5th order) and uses adaptive step-size.

```
size_t steps = integrate( harmonic_oscillator ,
        x , 0.0 , 10.0 , 0.1 );
```

The integrate function expects as parameters the rhs of the ode as defined above, the initial state `x`, the start-and end-time of the integration as well as the initial time step=size. Note, that `integrate` uses an adaptive step-size during the integration steps so the time points will not be equally spaced. The integration returns the number of steps that were applied and updates x which is set to the approximate solution of the ODE at the end of integration.

It is also possible to represent the ode system as a class. The rhs must then be implemented as a functor - a class with an overloaded function call operator:

```
/* The rhs of x' = f(x) defined as a class */
class harm_osc {

    double m_gam;

public:
    harm_osc( double gam ) : m_gam(gam) { }

    void operator() ( const state_type &x , state_type &dxdt , const double /* t */ )
    {
        dxdt[0] = x[1];
        dxdt[1] = -x[0] - m_gam*x[1];
    }
};
```

which can be used via

```
harm_osc ho(0.15);
steps = integrate( ho ,
        x , 0.0 , 10.0 , 0.1 );
```

In order to observe the solution during the integration steps all you have to do is to provide a reasonable observer. An example is

```
struct push_back_state_and_time
{
    std::vector< state_type >& m_states;
    std::vector< double >& m_times;

   push_back_state_and_time( std::vector< state_type > &states , std::vector< double > &times )
    : m_states( states ) , m_times( times ) { }

    void operator()( const state_type &x , double t )
    {
        m_states.push_back( x );
        m_times.push_back( t );
    }
};
```

which stores the intermediate steps in a container. Note, the argument structure of the ()-operator: odeint calls the observer exactly in this way, providing the current state and time. Now, you only have to pass this container to the integration function:

```
vector<state_type> x_vec;
vector<double> times;

steps = integrate( harmonic_oscillator ,
        x , 0.0 , 10.0 , 0.1 ,
        push_back_state_and_time( x_vec , times ) );

/* output */
for( size_t i=0; i<=steps; i++ )
{
    cout << times[i] << '\t' << x_vec[i][0] << '\t' << x_vec[i][1] << '\n';
}
```

That is all. You can use functional libraries like Boost.Lambda or Boost.Phoenix to ease the creation of observer functions.

The full cpp file for this example can be found here: harmonic_oscillator.cpp

Tutorial

Harmonic oscillator

Define the ODE

First of all, you have to specify the data type that represents a state *x* of your system. Mathematically, this usually is an n-dimensional vector with real numbers or complex numbers as scalar objects. For odeint the most natural way is to use `vector< double >` or `vector< complex< double > >` to represent the system state. However, odeint can deal with other container types as well, e.g. `boost::array< double , N >`, as long as it fulfills some requirements defined below.

To integrate a differential equation numerically, one also has to define the rhs of the equation *x' = f(x)*. In odeint you supply this function in terms of an object that implements the ()-operator with a certain parameter structure. Hence, the straightforward way would be to just define a function, e.g:

```
/* The type of container used to hold the state vector */
typedef std::vector< double > state_type;

const double gam = 0.15;

/* The rhs of x' = f(x) */
void harmonic_oscillator( const state_type &x , state_type &dxdt , const double /* t */ )
{
    dxdt[0] = x[1];
    dxdt[1] = -x[0] - gam*x[1];
}
```

The parameters of the function must follow the example above where `x` is the current state, here a two-component vector containing position *q* and momentum *p* of the oscillator, `dxdt` is the derivative *x'* and should be filled by the function with *f(x)*, and `t` is the current time. Note that in this example *t* is not required to calculate *f*, however odeint expects the function signature to have exactly three parameters (there are exception, discussed later).

A more sophisticated approach is to implement the system as a class where the rhs function is defined as the ()-operator of the class with the same parameter structure as above:

```
/* The rhs of x' = f(x) defined as a class */
class harm_osc {

    double m_gam;

public:
    harm_osc( double gam ) : m_gam(gam) { }

    void operator() ( const state_type &x , state_type &dxdt , const double /* t */ )
    {
        dxdt[0] = x[1];
        dxdt[1] = -x[0] - m_gam*x[1];
    }
};
```

odeint can deal with instances of such classes instead of pure functions which allows for cleaner code.

Stepper Types

Numerical integration works iteratively, that means you start at a state *x(t)* and perform a time-step of length *dt* to obtain the approximate state *x(t+dt)*. There exist many different methods to perform such a time-step each of which has a certain order *q*. If the order

of a method is *q* than it is accurate up to term $\sim dt^q$ that means the error in *x* made by such a step is $\sim dt^{q+1}$. odeint provides several steppers of different orders, see Stepper overview.

Some of steppers in the table above are special: Some need the Jacobian of the ODE, others are constructed for special ODE-systems like Hamiltonian systems. We will show typical examples and use-cases in this tutorial and which kind of steppers should be applied.

Integration with Constant Step Size

The basic stepper just performs one time-step and doesn't give you any information about the error that was made (except that you know it is of order *q+1*). Such steppers are used with constant step size that should be chosen small enough to have reasonable small errors. However, you should apply some sort of validity check of your results (like observing conserved quantities) because you have no other control of the error. The following example defines a basic stepper based on the classical Runge-Kutta scheme of 4th order. The declaration of the stepper requires the state type as template parameter. The integration can now be done by using the `integrate_const( Stepper, System, state, start_time, end_time, step_size )` function from odeint:

```
runge_kutta4< state_type > stepper;
integrate_const( stepper , harmonic_oscillator , x , 0.0 , 10.0 , 0.01 );
```

This call integrates the system defined by `harmonic_oscillator` using the RK4 method from *t=0* to *10* with a step-size *dt=0.01* and the initial condition given in `x`. The result, *x(t=10)* is stored in `x` (in-place). Each stepper defines a `do_step` method which can also be used directly. So, you write down the above example as

```
const double dt = 0.01;
for( double t=0.0 ; t<10.0 ; t+= dt )
    stepper.do_step( harmonic_oscillator , x , t , dt );
```

Tip

If you have a C++11 enabled compiler you can easily use lambdas to create the system function :

```
runge_kutta4< state_type > stepper;
integrate_const( stepper , []( const state_type &x , state_type &dxdt , double t ) {
        dxdt[0] = x[1]; dxdt[1] = -x[0] - gam*x[1]; }
    , x , 0.0 , 10.0 , 0.01 );
```

Integration with Adaptive Step Size

To improve the numerical results and additionally minimize the computational effort, the application of a step size control is advisable. Step size control is realized via stepper algorithms that additionally provide an error estimation of the applied step. odeint provides a number of such **ErrorSteppers** and we will show their usage on the example of `explicit_error_rk54_ck` - a 5th order Runge-Kutta method with 4th order error estimation and coefficients introduced by Cash and Karp.

```
typedef runge_kutta_cash_karp54< state_type > error_stepper_type;
```

Given the error stepper, one still needs an instance that checks the error and adjusts the step size accordingly. In odeint, this is done by **ControlledSteppers**. For the `runge_kutta_cash_karp54` stepper a `controlled_runge_kutta` stepper exists which can be used via

```
typedef controlled_runge_kutta< error_stepper_type > controlled_stepper_type;
controlled_stepper_type controlled_stepper;
integrate_adaptive( controlled_stepper , harmonic_oscillator , x , 0.0 , 10.0 , 0.01 );
```

As above, this integrates the system defined by `harmonic_oscillator`, but now using an adaptive step size method based on the Runge-Kutta Cash-Karp 54 scheme from *t=0* to *10* with an initial step size of *dt=0.01* (will be adjusted) and the initial condition given in x. The result, *x(t=10)*, will also be stored in x (in-place).

In the above example an error stepper is nested in a controlled stepper. This is a nice technique; however one drawback is that one always needs to define both steppers. One could also write the instantiation of the controlled stepper into the call of the integrate function but a complete knowledge of the underlying stepper types is still necessary. Another point is, that the error tolerances for the step size control are not easily included into the controlled stepper. Both issues can be solved by using `make_controlled`:

```
integrate_adaptive( make_controlled< error_stepper_type >( 1.0e-10 , 1.0e-6 ) ,
                    harmonic_oscillator , x , 0.0 , 10.0 , 0.01 );
```

`make_controlled` can be used with many of the steppers of odeint. The first parameter is the absolute error tolerance *eps_abs* and the second is the relative error tolerance *eps_rel* which is used during the integration. The template parameter determines from which error stepper a controlled stepper should be instantiated. An alternative syntax of `make_controlled` is

```
integrate_adaptive( make_controlled( 1.0e-10 , 1.0e-6 , error_stepper_type() ) ,
                    harmonic_oscillator , x , 0.0 , 10.0 , 0.01 );
```

For the Runge-Kutta controller the error made during one step is compared with *eps_abs + eps_rel * (a_x * |x| + a_{dxdt} * dt * |dxdt|)*. If the error is smaller than this value the current step is accepted, otherwise it is rejected and the step size is decreased. Note, that the step size is also increased if the error gets too small compared to the rhs of the above relation. The full instantiation of the `controlled_runge_kutta` with all parameters is therefore

```
double abs_err = 1.0e-10 , rel_err = 1.0e-6 , a_x = 1.0 , a_dxdt = 1.0;
controlled_stepper_type controlled_stepper(
    default_error_checker< double >( abs_err , rel_err , a_x , a_dxdt ) );
integrate_adaptive( controlled_stepper , harmonic_oscillator , x , 0.0 , 10.0 , 0.01 );
```

When using `make_controlled` the parameter a_x and a_{dxdt} are used with their standard values of 1.

In the tables below, one can find all steppers which are working with `make_controlled` and `make_dense_output` which is the analog for the dense output steppers.

Table 2. Generation functions make_controlled(abs_error , rel_error , stepper)

Stepper	Result of make_controlled	Remarks
`runge_kutta_cash_karp54`	`controlled_runge_kutta< runge_kutta_cash_karp54 , default_error_checker<...> >`	a_x=1, a_{dxdt}=1
`runge_kutta_fehlberg78`	`controlled_runge_kutta< runge_kutta_fehlberg78 , default_error_checker<...> >`	a_x=1, a_{dxdt}=1
`runge_kutta_dopri5`	`controlled_runge_kutta< runge_kutta_dopri5 , default_error_checker<...> >`	a_x=1, a_{dxdt}=1
`rosenbrock4`	`rosenbrock4_controlled< rosenbrock4 >`	-

Table 3. Generation functions make_dense_output(abs_error , rel_error , stepper)

Stepper	Result of make_dense_output	Remarks
`runge_kutta_dopri5`	`dense_output_runge_kutta< controlled_runge_kutta< runge_kutta_dopri5 , default_error_checker<...> > >`	$a_x=1$, $a_{dxdt}=1$
`rosenbrock4`	`rosenbrock4_dense_output< rosenbrock4_controller< rosenbrock4 > >`	-

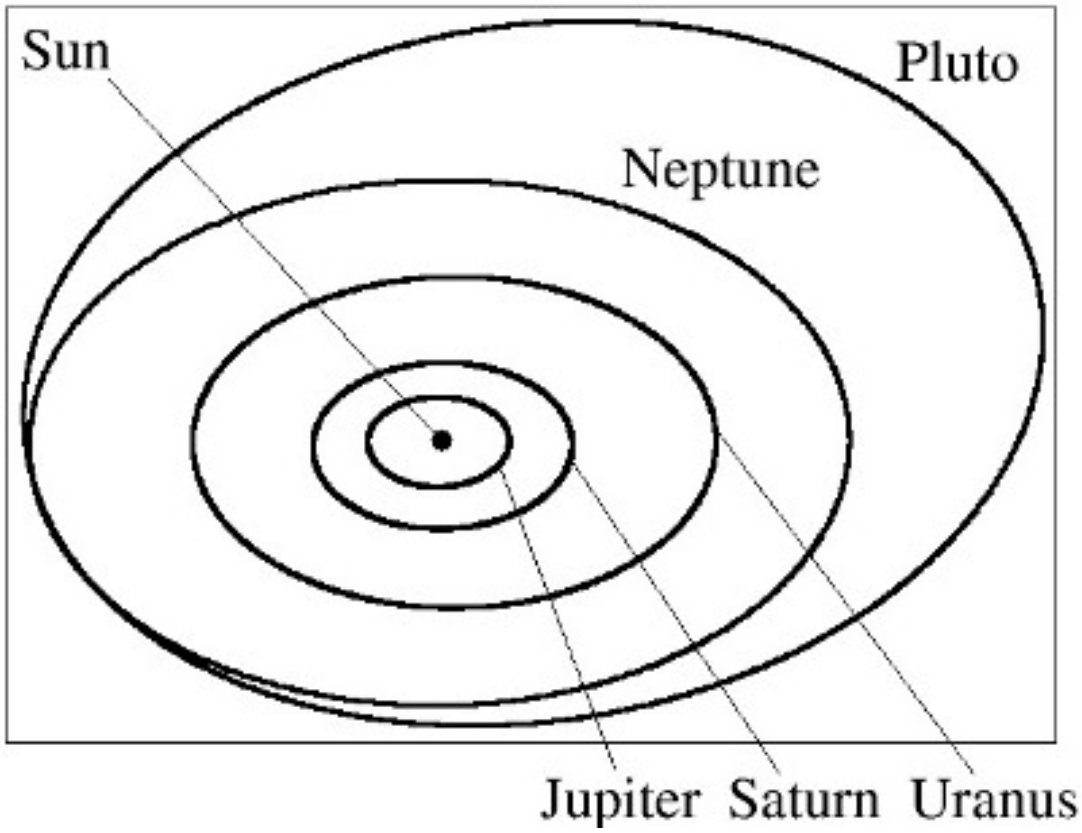
When using `make_controlled` or `make_dense_output` one should be aware which exact type is used and how the step size control works.

The full source file for this example can be found here: harmonic_oscillator.cpp

Solar system

Gravitation and energy conservation

The next example in this tutorial is a simulation of the outer solar system, consisting of the sun, Jupiter, Saturn, Uranus, Neptune and Pluto.



Each planet and of course the sun will be represented by mass points. The interaction force between each object is the gravitational force which can be written as

$$F_{ij} = -\gamma m_i m_j (q_i - q_j) / | q_i - q_j |^3$$

where γ is the gravitational constant, m_i and m_j are the masses and q_i and q_j are the locations of the two objects. The equations of motion are then

$dq_i / dt = p_i$

$dp_i / dt = 1 / m_i \Sigma_{ji} F_{ij}$

where p_i is the momenta of object i. The equations of motion can also be derived from the Hamiltonian

$H = \Sigma_i p_i^2 / (2 m_i) + \Sigma_j V(q_i , q_j)$

with the interaction potential $V(q_i,q_j)$. The Hamiltonian equations give the equations of motion

$dq_i / dt = dH / dp_i$

$dp_i / dt = -dH / dq_i$

In time independent Hamiltonian system the energy and the phase space volume are conserved and special integration methods have to be applied in order to ensure these conservation laws. The odeint library provides classes for separable Hamiltonian systems, which can be written in the form $H = \Sigma p_i^2 / (2m_i) + H_q(q)$, where $H_q(q)$ only depends on the coordinates. Although this functional form might look a bit arbitrary, it covers nearly all classical mechanical systems with inertia and without dissipation, or where the equations of motion can be written in the form $dq_i / dt = p_i / \mathrm{m_i}$, $dp_i / dt = f(q_i)$.

Note

A short physical note: While the two-body-problem is known to be integrable, that means it can be solved with purely analytic techniques, already the three-body-problem is not solvable. This was found in the end of the 19th century by H. Poincare which led to the whole new subject of Chaos Theory.

Define the system function

To implement this system we define a 3D point type which will represent the space as well as the velocity. Therefore, we use the operators from Boost.Operators:

```
/*the point type */
template< class T , size_t Dim >
class point :
    boost::additive1< point< T , Dim > ,
    boost::additive2< point< T , Dim  > , T ,
    boost::multiplicative2< point< T , Dim > , T
    > > >
    {
    public:

        const static size_t dim = Dim;
        typedef T value_type;
        typedef point< value_type , dim > point_type;

        // ...
        // constructors

        // ...
        // operators

    private:

        T m_val[dim];
    };

    //...
    // more operators
```

The next step is to define a container type storing the values of *q* and *p* and to define system functions. As container type we use `boost::array`

```
// we simulate 5 planets and the sun
const size_t n = 6;

typedef point< double , 3 > point_type;
typedef boost::array< point_type , n > container_type;
typedef boost::array< double , n > mass_type;
```

The `container_type` is different from the state type of the ODE. The state type of the ode is simply a `pair< container_type , container_type >` since it needs the information about the coordinates and the momenta.

Next we define the system's equations. As we will use a stepper that accounts for the Hamiltonian (energy-preserving) character of the system, we have to define the rhs different from the usual case where it is just a single function. The stepper will make use of the separable character, which means the system will be defined by two objects representing *f(p) = -dH/dq* and *g(q) = dH/dp*:

```
const double gravitational_constant = 2.95912208286e-4;

struct solar_system_coor
{
    const mass_type &m_masses;

    solar_system_coor( const mass_type &masses ) : m_masses( masses ) { }

    void operator()( const container_type &p , container_type &dqdt ) const
    {
        for( size_t i=0 ; i<n ; ++i )
            dqdt[i] = p[i] / m_masses[i];
    }
};
```

```
struct solar_system_momentum
{
    const mass_type &m_masses;

    solar_system_momentum( const mass_type &masses ) : m_masses( masses ) { }

    void operator()( const container_type &q , container_type &dpdt ) const
    {
        const size_t n = q.size();
        for( size_t i=0 ; i<n ; ++i )
        {
            dpdt[i] = 0.0;
            for( size_t j=0 ; j<i ; ++j )
            {
                point_type diff = q[j] - q[i];
                double d = abs( diff );
                diff *= ( gravitational_constant * m_masses[i] * m_masses[j] / d / d / d );
                dpdt[i] += diff;
                dpdt[j] -= diff;

            }
        }
    }
};
```

In general a three body-system is chaotic, hence we can not expect that arbitrary initial conditions of the system will lead to a solution comparable with the solar system dynamics. That is we have to define proper initial conditions, which are taken from the book of Hairer, Wannier, Lubich [4] .

As mentioned above, we need to use some special integrators in order to conserve phase space volume. There is a well known family of such integrators, the so-called Runge-Kutta-Nystroem solvers, which we apply here in terms of a `symplectic_rkn_sb3a_mclachlan` stepper:

```
typedef symplectic_rkn_sb3a_mclachlan< container_type > stepper_type;
const double dt = 100.0;

integrate_const(
        stepper_type() ,
        make_pair( solar_system_coor( masses ) , solar_system_momentum( masses ) ) ,
        make_pair( boost::ref( q ) , boost::ref( p ) ) ,
        0.0 , 200000.0 , dt , streaming_observer( cout ) );
```

These integration routine was used to produce the above sketch of the solar system. Note, that there are two particularities in this example. First, the state of the symplectic stepper is not `container_type` but a pair of `container_type`. Hence, we must pass such a pair to the integrate function. Since, we want to pass them as references we can simply pack them into Boost.Ref. The second point is the observer, which is called with a state type, hence a pair of `container_type`. The reference wrapper is also passed, but this is not a problem at all:

```
struct streaming_observer
{
    std::ostream& m_out;

    streaming_observer( std::ostream &out ) : m_out( out ) { }

    template< class State >
    void operator()( const State &x , double t ) const
    {
        container_type &q = x.first;
        m_out << t;
        for( size_t i=0 ; i<q.size() ; ++i ) m_out << "\t" << q[i];
        m_out << "\n";
    }
};
```

Tip

You can use C++11 lambda to create the observers

The full example can be found here: solar_system.cpp

Chaotic systems and Lyapunov exponents

In this example we present application of odeint to investigation of the properties of chaotic deterministic systems. In mathematical terms chaotic refers to an exponential growth of perturbations δx. In order to observe this exponential growth one usually solves the equations for the tangential dynamics which is again an ordinary differential equation. These equations are linear but time dependent and can be obtained via

$d \delta x / dt = J(x) \delta x$

where J is the Jacobian of the system under consideration. δx can also be interpreted as a perturbation of the original system. In principle n of these perturbations exist, they form a hypercube and evolve in the time. The Lyapunov exponents are then defined as logarithmic growth rates of the perturbations. If one Lyapunov exponent is larger then zero the nearby trajectories diverge exponentially hence they are chaotic. If the largest Lyapunov exponent is zero one is usually faced with periodic motion. In the case of a largest Lyapunov exponent smaller then zero convergence to a fixed point is expected. More information's about Lyapunov exponents and nonlinear dynamical systems can be found in many textbooks, see for example: E. Ott "Chaos is Dynamical Systems", Cambridge.

To calculate the Lyapunov exponents numerically one usually solves the equations of motion for n perturbations and orthonormalizes them every k steps. The Lyapunov exponent is the average of the logarithm of the stretching factor of each perturbation.

To demonstrate how one can use odeint to determine the Lyapunov exponents we choose the Lorenz system. It is one of the most studied dynamical systems in the nonlinear dynamics community. For the standard parameters it possesses a strange attractor with non-integer dimension. The Lyapunov exponents take values of approximately 0.9, 0 and -12.

The implementation of the Lorenz system is

```
const double sigma = 10.0;
const double R = 28.0;
const double b = 8.0 / 3.0;

typedef boost::array< double , 3 > lorenz_state_type;

void lorenz( const lorenz_state_type &x , lorenz_state_type &dxdt , double t )
{
    dxdt[0] = sigma * ( x[1] - x[0] );
    dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
    dxdt[2] = -b * x[2] + x[0] * x[1];
}
```

We need also to integrate the set of the perturbations. This is done in parallel to the original system, hence within one system function. Of course, we want to use the above definition of the Lorenz system, hence the definition of the system function including the Lorenz system itself and the perturbation could look like:

```
const size_t n = 3;
const size_t num_of_lyap = 3;
const size_t N = n + n*num_of_lyap;

typedef std::tr1::array< double , N > state_type;
typedef std::tr1::array< double , num_of_lyap > lyap_type;

void lorenz_with_lyap( const state_type &x , state_type &dxdt , double t )
{
    lorenz( x , dxdt , t );

    for( size_t l=0 ; l<num_of_lyap ; ++l )
    {
        const double *pert = x.begin() + 3 + l * 3;
        double *dpert = dxdt.begin() + 3 + l * 3;
        dpert[0] = - sigma * pert[0] + 10.0 * pert[1];
        dpert[1] = ( R - x[2] ) * pert[0] - pert[1] - x[0] * pert[2];
        dpert[2] = x[1] * pert[0] + x[0] * pert[1] - b * pert[2];
    }
}
```

The perturbations are stored linearly in the `state_type` behind the state of the Lorenz system. The problem of lorenz() and lorenz_with_lyap() having different state types may be solved putting the Lorenz system inside a functor with templatized arguments:

```
struct lorenz
{
    template< class StateIn , class StateOut , class Value >
    void operator()( const StateIn &x , StateOut &dxdt , Value t )
    {
        dxdt[0] = sigma * ( x[1] - x[0] );
        dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
        dxdt[2] = -b * x[2] + x[0] * x[1];
    }
};

void lorenz_with_lyap( const state_type &x , state_type &dxdt , double t )
{
    lorenz()( x , dxdt , t );
    ...
}
```

This works fine and `lorenz_with_lyap` can be used for example via

```
state_type x;
// initialize x
explicit_rk4< state_type > rk4;
integrate_n_steps( rk4 , lorenz_with_lyap , x , 0.0 , 0.01 , 1000 );
```

This code snippet performs 1000 steps with constant step size 0.01.

A real world use case for the calculation of the Lyapunov exponents of Lorenz system would always include some transient steps, just to ensure that the current state lies on the attractor, hence it would look like

```
state_type x;
// initialize x
explicit_rk4< state_type > rk4;
integrate_n_steps( rk4 , lorenz , x , 0.0 , 0.01 , 1000 );
```

The problem is now, that x is the full state containing also the perturbations and `integrate_n_steps` does not know that it should only use 3 elements. In detail, odeint and its steppers determine the length of the system under consideration by determining the length of the state. In the classical solvers, e.g. from Numerical Recipes, the problem was solved by pointer to the state and an appropriate length, something similar to

```
void lorenz( double* x , double *dxdt , double t, void* params )
{
    ...
}

int system_length = 3;
rk4( x , system_length , t , dt , lorenz );
```

But odeint supports a similar and much more sophisticated concept: Boost.Range. To make the steppers and the system ready to work with Boost.Range the system has to by changed:

```
struct lorenz
{
    template< class State , class Deriv >
    void operator()( const State &x_ , Deriv &dxdt_ , double t ) const
    {
        typename boost::range_iterator< const State >::type x = boost::begin( x_ );
        typename boost::range_iterator< Deriv >::type dxdt = boost::begin( dxdt_ );

        dxdt[0] = sigma * ( x[1] - x[0] );
        dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
        dxdt[2] = -b * x[2] + x[0] * x[1];
    }
};
```

This is in principle all. Now, we only have to call `integrate_n_steps` with a range including only the first 3 components of *x*:

```
// perform 10000 transient steps
integrate_n_steps( rk4 , lorenz() , std::make_pair( x.begin() , x.be↵
gin() + n ) , 0.0 , dt , 10000 );
```

Having integrated a sufficient number of transients steps we are now able to calculate the Lyapunov exponents:

1. Initialize the perturbations. They are stored linearly behind the state of the Lorenz system. The perturbations are initialized such that $p_{ij} = \delta_{ij}$, where p_{ij} is the j-component of the i.-th perturbation and δ_{ij} is the Kronecker symbol.

2. Integrate 100 steps of the full system with perturbations

3. Orthonormalize the perturbation using Gram-Schmidt orthonormalization algorithm.

4. Repeat step 2 and 3. Every 10000 steps write the current Lyapunov exponent.

```
fill( x.begin()+n , x.end() , 0.0 );
for( size_t i=0 ; i<num_of_lyap ; ++i ) x[n+n*i+i] = 1.0;
fill( lyap.begin() , lyap.end() , 0.0 );

double t = 0.0;
size_t count = 0;
while( true )
{

    t = integrate_n_steps( rk4 , lorenz_with_lyap , x , t , dt , 100 );
    gram_schmidt< num_of_lyap >( x , lyap , n );
    ++count;

    if( !(count % 100000) )
    {
        cout << t;
        for( size_t i=0 ; i<num_of_lyap ; ++i ) cout << "\t" << lyap[i] / t ;
        cout << endl;
    }
}
```

The full code can be found here: chaotic_system.cpp

Stiff systems

An important class of ordinary differential equations are so called stiff system which are characterized by two or more time scales of different order. Examples of such systems are found in chemical systems where reaction rates of individual sub-reaction might differ over large ranges, for example:

$d S_1 / dt = -101 S_2 - 100 S_1$

$d S_2 / dt = S_1$

In order to efficiently solve stiff systems numerically the Jacobian

$J = d f_i / d x_j$

is needed. Here is the definition of the above example

```
typedef boost::numeric::ublas::vector< double > vector_type;
typedef boost::numeric::ublas::matrix< double > matrix_type;

struct stiff_system
{
    void operator()( const vector_type &x , vector_type &dxdt , double /* t */ )
    {
        dxdt[ 0 ] = -101.0 * x[ 0 ] - 100.0 * x[ 1 ];
        dxdt[ 1 ] = x[ 0 ];
    }
};

struct stiff_system_jacobi
{
    void operator()( const vector_type & /* x */ , matrix_type &J , const double & /* t */ , vec↵
tor_type &dfdt )
    {
        J( 0 , 0 ) = -101.0;
        J( 0 , 1 ) = -100.0;
        J( 1 , 0 ) = 1.0;
        J( 1 , 1 ) = 0.0;
        dfdt[0] = 0.0;
        dfdt[1] = 0.0;
    }
};
```

The state type has to be a `ublas::vector` and the matrix type must by a `ublas::matrix` since the stiff integrator only accepts these types. However, you might want use non-stiff integrators on this system, too - we will do so later for demonstration. Therefore we want to use the same function also with other state_types, realized by templatizing the `operator()`:

```
typedef boost::numeric::ublas::vector< double > vector_type;
typedef boost::numeric::ublas::matrix< double > matrix_type;

struct stiff_system
{
    template< class State >
    void operator()( const State &x , State &dxdt , double t )
    {
        ...
    }
};

struct stiff_system_jacobi
{
    template< class State , class Matrix >
    void operator()( const State &x , Matrix &J , const double &t , State &dfdt )
    {
        ...
    }
};
```

Now you can use `stiff_system` in combination with `std::vector` or `boost::array`. In the example the explicit time derivative of *f(x,t)* is introduced separately in the Jacobian. If *df / dt = 0* simply fill `dfdt` with zeros.

A well know solver for stiff systems is the Rosenbrock method. It has a step size control and dense output facilities and can be used like all the other steppers:

```
vector_type x( 2 , 1.0 );

size_t num_of_steps = integrate_const( make_dense_output< rosenbrock4< double > >( 1.0e-6 , 1.0e-
6 ) ,
        make_pair( stiff_system() , stiff_system_jacobi() ) ,
        x , 0.0 , 50.0 , 0.01 ,
        cout << phoenix::arg_names::arg2 << " " << phoenix::arg_names::arg1[0] << "\n" );
```

During the integration 71 steps have been done. Comparing to a classical Runge-Kutta solver this is a very good result. For example the Dormand-Prince 5 method with step size control and dense output yields 1531 steps.

```
vector_type x2( 2 , 1.0 );

size_t num_of_steps2 = integrate_const( make_dense_output< runge_kutta_dopri5< vec↵
tor_type > >( 1.0e-6 , 1.0e-6 ) ,
        stiff_system() , x2 , 0.0 , 50.0 , 0.01 ,
        cout << phoenix::arg_names::arg2 << " " << phoenix::arg_names::arg1[0] << "\n" );
```

Note, that we have used Boost.Phoenix, a great functional programming library, to create and compose the observer.

The full example can be found here: stiff_system.cpp

Complex state types

Thus far we have seen several examples defined for real values. odeint can handle complex state types, hence ODEs which are defined on complex vector spaces, as well. An example is the Stuart-Landau oscillator

$$d\,\Psi / dt = (1 + i\,\eta)\,\Psi + (1 + i\,\alpha)\,|\,\Psi\,|^2\,\Psi$$

where Ψ and i is a complex variable. The definition of this ODE in C++ using complex< double > as a state type may look as follows

```
typedef complex< double > state_type;

struct stuart_landau
{
    double m_eta;
    double m_alpha;

    stuart_landau( double eta = 1.0 , double alpha = 1.0 )
    : m_eta( eta ) , m_alpha( alpha ) { }

    void operator()( const state_type &x , state_type &dxdt , double t ) const
    {
        const complex< double > I( 0.0 , 1.0 );
        dxdt = ( 1.0 + m_eta * I ) * x - ( 1.0 + m_alpha * I ) * norm( x ) * x;
    }
};
```

One can also use a function instead of a functor to implement it

```
double eta = 1.0;
double alpha = 1.0;

void stuart_landau( const state_type &x , state_type &dxdt , double t )
{
    const complex< double > I( 0.0 , 1.0 );
    dxdt[0] = ( 1.0 + m_eta * I ) * x[0] - ( 1.0 + m_alpha * I ) * norm( x[0] ) * x[0];
}
```

We strongly recommend to use the first ansatz. In this case you have explicit control over the parameters of the system and are not restricted to use global variables to parametrize the oscillator.

When choosing the stepper type one has to account for the "unusual" state type: it is a single `complex<double>` opposed to the vector types used in the previous examples. This means that no iterations over vector elements have to be performed inside the stepper algorithm. You can enforce this by supplying additional template arguments to the stepper including the `vector_space_algebra`. Details on the usage of algebras can be found in the section Adapt your own state types.

```
state_type x = complex< double >( 1.0 , 0.0 );

const double dt = 0.1;

typedef runge_kutta4< state_type , double , state_type , double ,
                      vector_space_algebra > stepper_type;

integrate_const( stepper_type() , stuart_landau( 2.0 , 1.0 ) , x , 0.0 , 10.0 , dt , streaming_ob↵
server( cout ) );
```

The full cpp file for the Stuart-Landau example can be found here stuart_landau.cpp

Note

The fact that we have to configure a different algebra is solely due to the fact that we use a non-vector state type and not to the usage of complex values. So for, e.g. `vector< complex<double> >`, this would not be required.

Lattice systems

odeint can also be used to solve ordinary differential equations defined on lattices. A prominent example is the Fermi-Pasta-Ulam system [8] . It is a Hamiltonian system of nonlinear coupled harmonic oscillators. The Hamiltonian is

$H = \Sigma_i p_i^2/2 + 1/2 (q_{i+1} - q_i)^2 + \beta / 4 (q_{i+1} - q_i)^4$

Remarkably, the Fermi-Pasta-Ulam system was the first numerical experiment to be implemented on a computer. It was studied at Los Alamos in 1953 on one of the first computers (a MANIAC I) and it triggered a whole new tree of mathematical and physical science.

Like the Solar System, the FPU is solved again by a symplectic solver, but in this case we can speed up the computation because the q components trivially reduce to $dq_i / dt = p_i$. odeint is capable of doing this performance improvement. All you have to do is to call the symplectic solver with an state function for the p components. Here is how this function looks like

```
typedef vector< double > container_type;

struct fpu
{
    const double m_beta;

    fpu( const double beta = 1.0 ) : m_beta( beta ) { }

    // system function defining the ODE
    void operator()( const container_type &q , container_type &dpdt ) const
    {
        size_t n = q.size();
        double tmp = q[0] - 0.0;
        double tmp2 = tmp + m_beta * tmp * tmp * tmp;
        dpdt[0] = -tmp2;
        for( size_t i=0 ; i<n-1 ; ++i )
        {
            tmp = q[i+1] - q[i];
            tmp2 = tmp + m_beta * tmp * tmp * tmp;
            dpdt[i] += tmp2;
            dpdt[i+1] = -tmp2;
        }
        tmp = - q[n-1];
        tmp2 = tmp + m_beta * tmp * tmp * tmp;
        dpdt[n-1] += tmp2;
    }

    // calculates the energy of the system
    double energy( const container_type &q , const container_type &p ) const
    {
        // ...
    }

    // calculates the local energy of the system
    void local_energy( const container_type &q , const container_type &p , contain↵
er_type &e ) const
    {
        // ...
    }
};
```

You can also use `boost::array< double , N >` for the state type.

Now, you have to define your initial values and perform the integration:

```
const size_t n = 64;
container_type q( n , 0.0 ) , p( n , 0.0 );

for( size_t i=0 ; i<n ; ++i )
{
    p[i] = 0.0;
    q[i] = 32.0 * sin( double( i + 1 ) / double( n + 1 ) * M_PI );
}


const double dt = 0.1;

typedef symplectic_rkn_sb3a_mclachlan< container_type > stepper_type;
fpu fpu_instance( 8.0 );

integrate_const( stepper_type() , fpu_instance ,
        make_pair( boost::ref( q ) , boost::ref( p ) ) ,
        0.0 , 1000.0 , dt , streaming_observer( cout , fpu_instance , 10 ) );
```

The observer uses a reference to the system object to calculate the local energies:

```
struct streaming_observer
{
    std::ostream& m_out;
    const fpu &m_fpu;
    size_t m_write_every;
    size_t m_count;

    streaming_observer( std::ostream &out , const fpu &f , size_t write_every = 100 )
    : m_out( out ) , m_fpu( f ) , m_write_every( write_every ) , m_count( 0 ) { }

    template< class State >
    void operator()( const State &x , double t )
    {
        if( ( m_count % m_write_every ) == 0 )
        {
            container_type &q = x.first;
            container_type &p = x.second;
            container_type energy( q.size() );
            m_fpu.local_energy( q , p , energy );
            for( size_t i=0 ; i<q.size() ; ++i )
            {
                m_out << t << "\t" << i << "\t" << q[i] << "\t" << p[i] << "\t" << en↵
ergy[i] << "\n";
            }
            m_out << "\n";
            clog << t << "\t" << accumulate( energy.begin() , energy.end() , 0.0 ) << "\n";
        }
        ++m_count;
    }
};
```

The full cpp file for this FPU example can be found here fpu.cpp

Ensembles of oscillators

Another important high dimensional system of coupled ordinary differential equations is an ensemble of N all-to-all coupled phase oscillators [9] . It is defined as

$$d\phi_k / dt = \omega_k + \varepsilon / N \, \Sigma_j \, sin(\phi_j - \phi_k)$$

The natural frequencies ω_i of each oscillator follow some distribution and ε is the coupling strength. We choose here a Lorentzian distribution for ω_i. Interestingly a phase transition can be observed if the coupling strength exceeds a critical value. Above this value synchronization sets in and some of the oscillators oscillate with the same frequency despite their different natural frequencies. The transition is also called Kuramoto transition. Its behavior can be analyzed by employing the mean field of the phase

$$Z = K e^{i \Theta} = 1 / N \Sigma_k e^{i \phi_k}$$

The definition of the system function is now a bit more complex since we also need to store the individual frequencies of each oscillator.

```
typedef vector< double > container_type;


pair< double , double > calc_mean_field( const container_type &x )
{
    size_t n = x.size();
    double cos_sum = 0.0 , sin_sum = 0.0;
    for( size_t i=0 ; i<n ; ++i )
    {
        cos_sum += cos( x[i] );
        sin_sum += sin( x[i] );
    }
    cos_sum /= double( n );
    sin_sum /= double( n );

    double K = sqrt( cos_sum * cos_sum + sin_sum * sin_sum );
    double Theta = atan2( sin_sum , cos_sum );

    return make_pair( K , Theta );
}


struct phase_ensemble
{
    container_type m_omega;
    double m_epsilon;

    phase_ensemble( const size_t n , double g = 1.0 , double epsilon = 1.0 )
    : m_omega( n , 0.0 ) , m_epsilon( epsilon )
    {
        create_frequencies( g );
    }

    void create_frequencies( double g )
    {
        boost::mt19937 rng;
        boost::cauchy_distribution<> cauchy( 0.0 , g );
        boost::variate_generator< boost::mt19937&, boost::cauchy_distribu↵
tion<> > gen( rng , cauchy );
        generate( m_omega.begin() , m_omega.end() , gen );
    }

    void set_epsilon( double epsilon ) { m_epsilon = epsilon; }

    double get_epsilon( void ) const { return m_epsilon; }

    void operator()( const container_type &x , container_type &dxdt , double /* t */ ) const
```

```
    {
        pair< double , double > mean = calc_mean_field( x );
        for( size_t i=0 ; i<x.size() ; ++i )
            dxdt[i] = m_omega[i] + m_epsilon * mean.first * sin( mean.second - x[i] );
    }
};
```

Note, that we have used Z to simplify the equations of motion. Next, we create an observer which computes the value of Z and we record Z for different values of ε.

```
struct statistics_observer
{
    double m_K_mean;
    size_t m_count;

    statistics_observer( void )
    : m_K_mean( 0.0 ) , m_count( 0 ) { }

    template< class State >
    void operator()( const State &x , double t )
    {
        pair< double , double > mean = calc_mean_field( x );
        m_K_mean += mean.first;
        ++m_count;
    }

    double get_K_mean( void ) const { re↵
turn ( m_count != 0 ) ? m_K_mean / double( m_count ) : 0.0 ; }

    void reset( void ) { m_K_mean = 0.0; m_count = 0; }
};
```

Now, we do several integrations for different values of ε and record Z. The result nicely confirms the analytical result of the phase transition, i.e. in our example the standard deviation of the Lorentzian is 1 such that the transition will be observed at $\varepsilon = 2$.

```
const size_t n = 16384;
const double dt = 0.1;

container_type x( n );

boost::mt19937 rng;
boost::uniform_real<> unif( 0.0 , 2.0 * M_PI );
boost::variate_generator< boost::mt19937&, boost::uniform_real<> > gen( rng , unif );

// gamma = 1, the phase transition occurs at epsilon = 2
phase_ensemble ensemble( n , 1.0 );
statistics_observer obs;

for( double epsilon = 0.0 ; epsilon < 5.0 ; epsilon += 0.1 )
{
    ensemble.set_epsilon( epsilon );
    obs.reset();

    // start with random initial conditions
    generate( x.begin() , x.end() , gen );

    // calculate some transients steps
    integrate_const( runge_kutta4< container_type >() , boost::ref( en↵
semble ) , x , 0.0 , 10.0 , dt );

    // integrate and compute the statistics
    integrate_const( runge_kutta4< container_type >() , boost::ref( en↵
semble ) , x , 0.0 , 100.0 , dt , boost::ref( obs ) );
    cout << epsilon << "\t" << obs.get_K_mean() << endl;
}
```

The full cpp file for this example can be found here phase_oscillator_ensemble.cpp

Using boost::units

odeint also works well with Boost.Units - a library for compile time unit and dimension analysis. It works by decoding unit information into the types of values. For a one-dimensional unit you can just use the Boost.Unit types as state type, deriv type and time type and hand the `vector_space_algebra` to the stepper definition and everything works just fine:

```
typedef units::quantity< si::time , double > time_type;
typedef units::quantity< si::length , double > length_type;
typedef units::quantity< si::velocity , double > velocity_type;

typedef runge_kutta4< length_type , double , velocity_type , time_type ,
                      vector_space_algebra > stepper_type;
```

If you want to solve more-dimensional problems the individual entries typically have different units. That means that the `state_type` is now possibly heterogeneous, meaning that every entry might have a different type. To solve this problem, compile-time sequences from Boost.Fusion can be used.

To illustrate how odeint works with Boost.Units we use the harmonic oscillator as primary example. We start with defining all quantities

```
#include <boost/numeric/odeint.hpp>
#include <boost/numeric/odeint/algebra/fusion_algebra.hpp>

#include <boost/units/systems/si/length.hpp>
#include <boost/units/systems/si/time.hpp>
#include <boost/units/systems/si/velocity.hpp>
#include <boost/units/systems/si/acceleration.hpp>
#include <boost/units/systems/si/io.hpp>

#include <boost/fusion/container.hpp>

using namespace std;
using namespace boost::numeric::odeint;
namespace fusion = boost::fusion;
namespace units = boost::units;
namespace si = boost::units::si;

typedef units::quantity< si::time , double > time_type;
typedef units::quantity< si::length , double > length_type;
typedef units::quantity< si::velocity , double > velocity_type;
typedef units::quantity< si::acceleration , double > acceleration_type;
typedef units::quantity< si::frequency , double > frequency_type;

typedef fusion::vector< length_type , velocity_type > state_type;
typedef fusion::vector< velocity_type , acceleration_type > deriv_type;
```

Note, that the `state_type` and the `deriv_type` are now a compile-time fusion sequences. `deriv_type` represents x' and is now different from the state type as it has different unit definitions. Next, we define the ordinary differential equation which is completely equivalent to the example in Harmonic Oscillator:

```
struct oscillator
{
    frequency_type m_omega;

    oscillator( const frequency_type &omega = 1.0 * si::hertz ) : m_omega( omega ) { }

    void operator()( const state_type &x , deriv_type &dxdt , time_type t ) const
    {
        fusion::at_c< 0 >( dxdt ) = fusion::at_c< 1 >( x );
        fusion::at_c< 1 >( dxdt ) = - m_omega * m_omega * fusion::at_c< 0 >( x );
    }
};
```

Next, we instantiate an appropriate stepper. We must explicitly parametrize the stepper with the `state_type`, `deriv_type`, `time_type`. Furthermore, the iteration over vector elements is now done by the `fusion_algebra` which must also be given. For more on the state types / algebras see chapter Adapt your own state types.

```
typedef runge_kutta_dopri5< state_type , double , deriv_type , time_type , fusion_algebra > step↵
per_type;

state_type x( 1.0 * si::meter , 0.0 * si::meter_per_second );

integrate_const( make_dense_output( 1.0e-6 , 1.0e-6 , stepper_type() ) , oscillat↵
or( 2.0 * si::hertz ) , x , 0.0 * si::second , 100.0 * si::second , 0.1 * si::second , stream↵
ing_observer( cout ) );
```

It is quite easy but the compilation time might take very long. Furthermore, the observer is defined a bit different

```
struct streaming_observer
{
    std::ostream& m_out;

    streaming_observer( std::ostream &out ) : m_out( out ) { }

    struct write_element
    {
        std::ostream &m_out;
        write_element( std::ostream &out ) : m_out( out ) { };

        template< class T >
        void operator()( const T &t ) const
        {
            m_out << "\t" << t;
        }
    };

    template< class State , class Time >
    void operator()( const State &x , const Time &t ) const
    {
        m_out << t;
        fusion::for_each( x , write_element( m_out ) );
        m_out << "\n";
    }
};
```

Caution

Using Boost.Units works nicely but compilation can be very time and memory consuming. For example the unit test for the usage of Boost.Units in odeint take up to 4 GB of memory at compilation.

The full cpp file for this example can be found here harmonic_oscillator_units.cpp.

Using matrices as state types

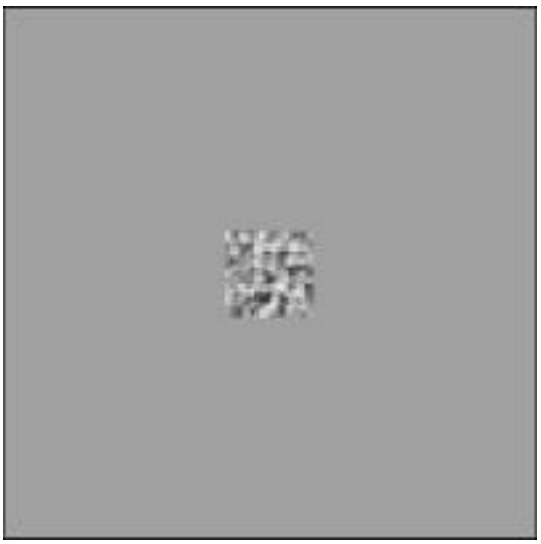
odeint works well with a variety of different state types. It is not restricted to pure vector-wise types, like `vector< double >`, `array< double , N >`, `fusion::vector< double , double >`, etc. but also works with types having a different topology then simple vectors. Here, we show how odeint can be used with matrices as states type, in the next section we will show how can be used to solve ODEs defined on complex networks.

By default, odeint can be used with `ublas::matrix< T >` as state type for matrices. A simple example is a two-dimensional lattice of coupled phase oscillators. Other matrix types like `mtl::dense_matrix` or blitz arrays and matrices can used as well but need some kind of activation in order to work with odeint. This activation is described in following sections,

The definition of the system is

```
typedef boost::numeric::ublas::matrix< double > state_type;

struct two_dimensional_phase_lattice
{
    two_dimensional_phase_lattice( double gamma = 0.5 )
    : m_gamma( gamma ) { }

    void operator()( const state_type &x , state_type &dxdt , double /* t */ ) const
    {
        size_t size1 = x.size1() , size2 = x.size2();

        for( size_t i=1 ; i<size1-1 ; ++i )
        {
            for( size_t j=1 ; j<size2-1 ; ++j )
            {
                dxdt( i , j ) =
                        coupling_func( x( i + 1 , j ) - x( i , j ) ) +
                        coupling_func( x( i - 1 , j ) - x( i , j ) ) +
                        coupling_func( x( i , j + 1 ) - x( i , j ) ) +
                        coupling_func( x( i , j - 1 ) - x( i , j ) );
            }
        }

        for( size_t i=0 ; i<x.size1() ; ++i ) dxdt( i , 0 ) = dxdt( i , x.size2() -1 ) = 0.0;
        for( size_t j=0 ; j<x.size2() ; ++j ) dxdt( 0 , j ) = dxdt( x.size1() -1 , j ) = 0.0;
    }

    double coupling_func( double x ) const
    {
        return sin( x ) - m_gamma * ( 1.0 - cos( x ) );
    }

    double m_gamma;
};
```

In principle this is all. Please note, that the above code is far from being optimal. Better performance can be achieved if every interaction is only calculated once and iterators for columns and rows are used. Below are some visualizations of the evolution of this lattice equation.



The full cpp for this example can be found here two_dimensional_phase_lattice.cpp.

Using arbitrary precision floating point types

Besides the classical floating point number like `float`, `double`, `complex< double >` you can also use arbitrary precision types, like the types from gmp and mpfr. But you have to be careful about instantiating any numbers.

For gmp types you have to set the default precision before any number is instantiated. This can be done by calling `mpf_set_default_prec( precision )` as the first function in your main program. Secondly, you can not use any global constant variables since they will not be set with the default precision you have already set.

Here is a simple example:

```
typedef mpf_class value_type;
typedef boost::array< value_type , 3 > state_type;

struct lorenz
{
    void operator()( const state_type &x , state_type &dxdt , value_type t ) const
    {
        const value_type sigma( 10.0 );
        const value_type R( 28.0 );
        const value_type b( value_type( 8.0 ) / value_type( 3.0 ) );

        dxdt[0] = sigma * ( x[1] - x[0] );
        dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
        dxdt[2] = -b * x[2] + x[0] * x[1];
    }
};
```

which can be integrated:

```
const int precision = 1024;
mpf_set_default_prec( precision );

state_type x = {{ value_type( 10.0 ) , value_type( 10.0 ) , value_type( 10.0 ) }};

cout.precision( 1000 );
integrate_const( runge_kutta4< state_type , value_type >() ,
       ↵
 lorenz() , x , value_type( 0.0 ) , value_type( 10.0 ) , value_type( value_type( 1.0 ) / value_type( 10.0 ) ) ,
        streaming_observer( cout ) );
```

Caution

The full support of arbitrary precision types depends on the functionality they provide. For example, the types from gmp are lacking of functions for calculating the power and arbitrary roots, hence they can not be used with the controlled steppers. In detail, for full support the `min( x , y )`, `max( x , y )`, `pow( x , y )` must be callable.

The full example can be found at lorenz_gmpxx.cpp.

Self expanding lattices

odeint supports changes of the state size during integration if a state_type is used which can be resized, like `std::vector`. The adjustment of the state's size has to be done from outside and the stepper has to be instantiated with `always_resizer` as the template argument for the `resizer_type`. In this configuration, the stepper checks for changes in the state size and adjust it's internal storage accordingly.

We show this for a Hamiltonian system of nonlinear, disordered oscillators with nonlinear nearest neighbor coupling.

The system function is implemented in terms of a class that also provides functions for calculating the energy. Note, that this class stores the random potential internally which is not resized, but rather a start index is kept which should be changed whenever the states' size change.

```
typedef vector< double > coord_type;
typedef pair< coord_type , coord_type > state_type;

struct compacton_lattice
{
    const int m_max_N;
    const double m_beta;
    int m_pot_start_index;
    vector< double > m_pot;

    compacton_lattice( int max_N , double beta , int pot_start_index )
       : m_max_N( max_N ) , m_beta( beta ) , m_pot_start_index( pot_start_index ) , m_pot( max_N )
    {
        srand( time( NULL ) );
        // fill random potential with iid values from [0,1]
        boost::mt19937 rng;
        boost::uniform_real<> unif( 0.0 , 1.0 );
        boost::variate_generator< boost::mt19937&, boost::uniform_real<> > gen( rng , unif );
        generate( m_pot.begin() , m_pot.end() , gen );
    }

    void operator()( const coord_type &q , coord_type &dpdt )
    {
        // calculate dpdt = -dH/dq of this hamiltonian system
        // dp_i/dt = - V_i * q_i^3 - beta*(q_i - q_{i-1})^3 + beta*(q_{i+1} - q_i)^3
        const int N = q.size();
        double diff = q[0] - q[N-1];
        for( int i=0 ; i<N ; ++i )
        {
            dpdt[i] = - m_pot[m_pot_start_index+i] * q[i]*q[i]*q[i] -
                    m_beta * diff*diff*diff;
            diff = q[(i+1) % N] - q[i];
            dpdt[i] += m_beta * diff*diff*diff;
        }
    }

    void energy_distribution( const coord_type &q , const coord_type &p , coord_type &energies )
    {
        // computes the energy per lattice site normalized by total energy
        const size_t N = q.size();
        double en = 0.0;
        for( size_t i=0 ; i<N ; i++ )
        {
            const double diff = q[(i+1) % N] - q[i];
            energies[i] = p[i]*p[i]/2.0
                + m_pot[m_pot_start_index+i]*q[i]*q[i]*q[i]*q[i]/4.0
                + m_beta/4.0 * diff*diff*diff*diff;
            en += energies[i];
        }
        en = 1.0/en;
        for( size_t i=0 ; i<N ; i++ )
        {
            energies[i] *= en;
        }
    }

    double energy( const coord_type &q , const coord_type &p )
    {
        // calculates the total energy of the excitation
        const size_t N = q.size();
        double en = 0.0;
        for( size_t i=0 ; i<N ; i++ )
        {
```

```
            const double diff = q[(i+1) % N] - q[i];
            en += p[i]*p[i]/2.0
                + m_pot[m_pot_start_index+i]*q[i]*q[i]*q[i]*q[i] / 4.0
                + m_beta/4.0 * diff*diff*diff*diff;
        }
        return en;
    }

    void change_pot_start( const int delta )
    {
        m_pot_start_index += delta;
    }
};
```

The total size we allow is 1024 and we start with an initial state size of 60.

```
//start with 60 sites
const int N_start = 60;
coord_type q( N_start , 0.0 );
q.reserve( max_N );
coord_type p( N_start , 0.0 );
p.reserve( max_N );
// start with uniform momentum distribution over 20 sites
fill( p.begin()+20 , p.end()-20 , 1.0/sqrt(20.0) );

coord_type distr( N_start , 0.0 );
distr.reserve( max_N );

// create the system
compacton_lattice lattice( max_N , beta , (max_N-N_start)/2 );

//create the stepper, note that we use an always_resizer because state size might change during ↵
steps
typedef symplectic_rkn_sb3a_mclachlan< coord_type , coord_type , double , coord_type , co↵
ord_type , double ,
        range_algebra , default_operations , always_resizer > hamiltonian_stepper;
hamiltonian_stepper stepper;
hamiltonian_stepper::state_type state = make_pair( q , p );
```

The lattice gets resized whenever the energy distribution comes close to the borders `distr[10] > 1E-150`, `distr[distr.size()-10] > 1E-150`. If we increase to the left, `q` and `p` have to be rotated because their resize function always appends at the end. Additionally, the start index of the potential changes in this case.

```
double t = 0.0;
const double dt = 0.1;
const int steps = 10000;
for( int step = 0 ; step < steps ; ++step )
{
    stepper.do_step( boost::ref(lattice) , state , t , dt );
    lattice.energy_distribution( state.first , state.second , distr );
    if( distr[10] > 1E-150 )
    {
        do_resize( state.first , state.second , distr , state.first.size()+20 );
        rotate( state.first.begin() , state.first.end()-20 , state.first.end() );
        rotate( state.second.begin() , state.second.end()-20 , state.second.end() );
        lattice.change_pot_start( -20 );
        cout << t << ": resized left to " << distr.size() << ", energy = " << lattice.en⌟
ergy( state.first , state.second ) << endl;
    }
    if( distr[distr.size()-10] > 1E-150 )
    {
        do_resize( state.first , state.second , distr , state.first.size()+20 );
        cout << t << ": resized right to " << distr.size() << ", energy = " << lattice.en⌟
ergy( state.first , state.second ) << endl;
    }
    t += dt;
}
```

The `do_resize` function simply calls `vector.resize` of `q` , `p` and `distr`.

```
void do_resize( coord_type &q , coord_type &p , coord_type &distr , const int N )
{
    q.resize( N );
    p.resize( N );
    distr.resize( N );
}
```

The full example can be found in resizing_lattice.cpp

Using CUDA (or OpenMP, TBB, ...) via Thrust

Modern graphic cards (graphic processing units - GPUs) can be used to speed up the performance of time consuming algorithms by means of massive parallelization. They are designed to execute many operations in parallel. odeint can utilize the power of GPUs by means of CUDA and Thrust, which is a STL-like interface for the native CUDA API.

Important

Thrust also supports parallelization using OpenMP and Intel Threading Building Blocks (TBB). You can switch between CUDA, OpenMP and TBB parallelizations by a simple compiler switch. Hence, this also provides an easy way to get basic OpenMP parallelization into odeint. The examples discussed below are focused on GPU parallelization, though.

To use odeint with CUDA a few points have to be taken into account. First of all, the problem has to be well chosen. It makes absolutely no sense to try to parallelize the code for a three dimensional system, it is simply too small and not worth the effort. One single function call (kernel execution) on the GPU is slow but you can do the operation on a huge set of data with only one call. We have experienced that the vector size over which is parallelized should be of the order of 10^6 to make full use of the GPU. Secondly, you have to use Thrust's algorithms and functors when implementing the rhs the ODE. This might be tricky since it involves some kind of functional programming knowledge.

Typical applications for CUDA and odeint are large systems, like lattices or discretizations of PDE, and parameter studies. We introduce now three examples which show how the power of GPUs can be used in combination with odeint.

Important

The full power of CUDA is only available for really large systems where the number of coupled ordinary differential equations is of order $N=10^6$ or larger. For smaller systems the CPU is usually much faster. You can also integrate an ensemble of different uncoupled ODEs in parallel as shown in the last example.

Phase oscillator ensemble

The first example is the phase oscillator ensemble from the previous section:

$d\phi_k / dt = \omega_k + \varepsilon / N \Sigma_j sin(\phi_j - \phi_k)$.

It has a phase transition at $\varepsilon = 2$ in the limit of infinite numbers of oscillators N. In the case of finite N this transition is smeared out but still clearly visible.

Thrust and CUDA are perfectly suited for such kinds of problems where one needs a large number of particles (oscillators). We start by defining the state type which is a `thrust::device_vector`. The content of this vector lives on the GPU. If you are not familiar with this we recommend reading the *Getting started* section on the Thrust website.

```
//change this to float if your device does not support double computation
typedef double value_type;

//change this to host_vector< ... > of you want to run on CPU
typedef thrust::device_vector< value_type > state_type;
// typedef thrust::host_vector< value_type > state_type;
```

Thrust follows a functional programming approach. If you want to perform a calculation on the GPU you usually have to call a global function like `thrust::for_each`, `thrust::reduce`, ... with an appropriate local functor which performs the basic operation. An example is

```
struct add_two
{
    template< class T >
    __host__ __device__
    void operator()( T &t ) const
    {
        t += T( 2 );
    }
};

// ...

thrust::for_each( x.begin() , x.end() , add_two() );
```

This code generically adds two to every element in the container `x`.

For the purpose of integrating the phase oscillator ensemble we need

- to calculate the system function, hence the r.h.s. of the ODE.
- this involves computing the mean field of the oscillator example, i.e. the values of R and θ

The mean field is calculated in a class `mean_field_calculator`

```
struct mean_field_calculator
{
    struct sin_functor : public thrust::unary_function< value_type , value_type >
    {
        __host__ __device__
        value_type operator()( value_type x) const
        {
            return sin( x );
        }
    };

    struct cos_functor : public thrust::unary_function< value_type , value_type >
    {
        __host__ __device__
        value_type operator()( value_type x) const
        {
            return cos( x );
        }
    };

    static std::pair< value_type , value_type > get_mean( const state_type &x )
    {
        value_type sin_sum = thrust::reduce(
                thrust::make_transform_iterator( x.begin() , sin_functor() ) ,
                thrust::make_transform_iterator( x.end() , sin_functor() ) );
        value_type cos_sum = thrust::reduce(
                thrust::make_transform_iterator( x.begin() , cos_functor() ) ,
                thrust::make_transform_iterator( x.end() , cos_functor() ) );

        cos_sum /= value_type( x.size() );
        sin_sum /= value_type( x.size() );

        value_type K = sqrt( cos_sum * cos_sum + sin_sum * sin_sum );
        value_type Theta = atan2( sin_sum , cos_sum );

        return std::make_pair( K , Theta );
    }
};
```

Inside this class two member structures `sin_functor` and `cos_functor` are defined. They compute the sine and the cosine of a value and they are used within a transform iterator to calculate the sum of $sin(\phi_k)$ and $cos(\phi_k)$. The classifiers `__host__` and `__device__` are CUDA specific and define a function or operator which can be executed on the GPU as well as on the CPU. The line

```
value_type sin_sum = thrust::reduce(
        thrust::make_transform_iterator( x.begin() , sin_functor() ) ,
        thrust::make_transform_iterator( x.end() , sin_functor() ) );
```

performs the calculation of this sine-sum on the GPU (or on the CPU, depending on your thrust configuration).

The system function is defined via

```
class phase_oscillator_ensemble
{

public:

    struct sys_functor
    {
        value_type m_K , m_Theta , m_epsilon;

        sys_functor( value_type K , value_type Theta , value_type epsilon )
        : m_K( K ) , m_Theta( Theta ) , m_epsilon( epsilon ) { }

        template< class Tuple >
        __host__ __device__
        void operator()( Tuple t )
        {
            thrust::get<2>(t) = thrust::get<1>(t) + m_epsi↵
lon * m_K * sin( m_Theta - thrust::get<0>(t) );
        }
    };

    // ...

    void operator() ( const state_type &x , state_type &dxdt , const value_type dt ) const
    {
        std::pair< value_type , value_type > mean_field = mean_field_calculator::get_mean( x );

        thrust::for_each(
                thrust::make_zip_iterator( thrust::make_tuple( x.begin() , m_omega.be↵
gin() , dxdt.begin() ) ),
                thrust::make_zip_iterat↵
or( thrust::make_tuple( x.end() , m_omega.end() , dxdt.end()) ) ,
                sys_functor( mean_field.first , mean_field.second , m_epsilon )
                );
    }

    // ...
};
```

This class is used within the `do_step` and `integrate` method. It defines a member structure `sys_functor` for the r.h.s. of each individual oscillator and the `operator()` for the use in the steppers and integrators of odeint. The functor computes first the mean field of ϕ_k and secondly calculates the whole r.h.s. of the ODE using this mean field. Note, how nicely `thrust::tuple` and `thrust::zip_iterator` play together.

Now, we are ready to put everything together. All we have to do for making odeint ready for using the GPU is to parametrize the stepper with the appropriate thrust algebra/operations:

```
typedef runge_kutta4< state_type , value_type , state_type , value_type , thrust_al↵
gebra , thrust_operations > stepper_type;
```

You can also use a controlled or dense output stepper, e.g.

```
typedef runge_kutta_dopri5< state_type , value_type , state_type , value_type , thrust_al↵
gebra , thrust_operations > stepper_type;
```

Then, it is straightforward to integrate the phase ensemble by creating an instance of the rhs class and using an integration function:

```
phase_oscillator_ensemble ensemble( N , 1.0 );
```

```
size_t steps1 = integrate_const( make_controlled( 1.0e-6 , 1.0e-6 , step↵
per_type() ) , boost::ref( ensemble ) , x , 0.0 , t_transients , dt );
```

We have to use `boost::ref` here in order to pass the rhs class as reference and not by value. This ensures that the natural frequencies of each oscillator are not copied when calling `integrate_const`. In the full example the performance and results of the Runge-Kutta-4 and the Dopri5 solver are compared.

The full example can be found at phase_oscillator_example.cu.

Large oscillator chains

The next example is a large, one-dimensional chain of nearest-neighbor coupled phase oscillators with the following equations of motion:

$d \phi_k / dt = \omega_k + sin(\phi_{k+1} - \phi_k) + sin(\phi_k - \phi_{k-1})$

In principle we can use all the techniques from the previous phase oscillator ensemble example, but we have to take special care about the coupling of the oscillators. To efficiently implement the coupling you can use a very elegant way employing Thrust's permutation iterator. A permutation iterator behaves like a normal iterator on a vector but it does not iterate along the usual order of the elements. It rather iterates along some permutation of the elements defined by some index map. To realize the nearest neighbor coupling we create one permutation iterator which travels one step behind a usual iterator and another permutation iterator which travels one step in front. The full system class is:

```
//change this to host_vector< ... > if you want to run on CPU
typedef thrust::device_vector< value_type > state_type;
typedef thrust::device_vector< size_t > index_vector_type;
//typedef thrust::host_vector< value_type > state_type;
//typedef thrust::host_vector< size_t > index_vector_type;

class phase_oscillators
{

public:

    struct sys_functor
    {
        template< class Tuple >
        __host__ __device__
        void operator()( Tuple t )  // this functor works on tuples of values
        {
            // first, unpack the tuple into value, neighbors and omega
            const value_type phi = thrust::get<0>(t);
            const value_type phi_left = thrust::get<1>(t);  // left neighbor
            const value_type phi_right = thrust::get<2>(t); // right neighbor
            const value_type omega = thrust::get<3>(t);
            // the dynamical equation
            thrust::get<4>(t) = omega + sin( phi_right - phi ) + sin( phi - phi_left );
        }
    };

    phase_oscillators( const state_type &omega )
       : m_omega( omega ) , m_N( omega.size() ) , m_prev( omega.size() ) , m_next( omega.size() )
    {
        // build indices pointing to left and right neighbours
        thrust::counting_iterator<size_t> c( 0 );
        thrust::copy( c , c+m_N-1 , m_prev.begin()+1 );
        m_prev[0] = 0; // m_prev = { 0 , 0 , 1 , 2 , 3 , ... , N-1 }

        thrust::copy( c+1 , c+m_N , m_next.begin() );
        m_next[m_N-1] = m_N-1; // m_next = { 1 , 2 , 3 , ... , N-1 , N-1 }
    }

    void operator() ( const state_type &x , state_type &dxdt , const value_type dt )
    {
        thrust::for_each(
                thrust::make_zip_iterator(
                        thrust::make_tuple(
                                x.begin() ,
                              thrust::make_permutation_iterator( x.begin() , m_prev.begin() ) ,
                              thrust::make_permutation_iterator( x.begin() , m_next.begin() ) ,
                                m_omega.begin() ,
                                dxdt.begin()
                                ) ),
                thrust::make_zip_iterator(
                        thrust::make_tuple(
                                x.end() ,
                               thrust::make_permutation_iterator( x.begin() , m_prev.end() ) ,
                               thrust::make_permutation_iterator( x.begin() , m_next.end() ) ,
                                m_omega.end() ,
                                dxdt.end()) ) ,
                sys_functor()
                );
    }

private:
```

```
    const state_type &m_omega;
    const size_t m_N;
    index_vector_type m_prev;
    index_vector_type m_next;
};
```

Note, how easy you can obtain the value for the left and right neighboring oscillator in the system functor using the permutation iterators. But, the call of the `thrust::for_each` function looks relatively complicated. Every term of the r.h.s. of the ODE is resembled by one iterator packed in exactly the same way as it is unpacked in the functor above.

Now we put everything together. We create random initial conditions and decreasing frequencies such that we should get synchronization. We copy the frequencies and the initial conditions onto the device and finally initialize and perform the integration. As result we simply write out the current state, hence the phase of each oscillator.

```
// create initial conditions and omegas on host:
vector< value_type > x_host( N );
vector< value_type > omega_host( N );
for( size_t i=0 ; i<N ; ++i )
{
    x_host[i] = 2.0 * pi * drand48();
    omega_host[i] = ( N - i ) * epsilon; // decreasing frequencies
}

// copy to device
state_type x = x_host;
state_type omega = omega_host;

// create stepper
runge_kutta4< state_type , value_type , state_type , value_type , thrust_algebra , thrust_opera↵
tions > stepper;

// create phase oscillator system function
phase_oscillators sys( omega );

// integrate
integrate_const( stepper , sys , x , 0.0 , 10.0 , dt );

thrust::copy( x.begin() , x.end() , std::ostream_iterator< value_type >( std::cout , "\n" ) );
std::cout << std::endl;
```

The full example can be found at phase_oscillator_chain.cu.

Parameter studies

Another important use case for Thrust and CUDA are parameter studies of relatively small systems. Consider for example the three-dimensional Lorenz system from the chaotic systems example in the previous section which has three parameters. If you want to study the behavior of this system for different parameters you usually have to integrate the system for many parameter values. Using thrust and odeint you can do this integration in parallel, hence you integrate a whole ensemble of Lorenz systems where each individual realization has a different parameter value.

In the following we will show how you can use Thrust to integrate the above mentioned ensemble of Lorenz systems. We will vary only the parameter β but it is straightforward to vary other parameters or even two or all three parameters. Furthermore, we will use the largest Lyapunov exponent to quantify the behavior of the system (chaoticity).

We start by defining the range of the parameters we want to study. The state_type is again a `thrust::device_vector< value_type >`.

```
vector< value_type > beta_host( N );
const value_type beta_min = 0.0 , beta_max = 56.0;
for( size_t i=0 ; i<N ; ++i )
    beta_host[i] = beta_min + value_type( i ) * ( beta_max - beta_min ) / value_type( N - 1 );

state_type beta = beta_host;
```

The next thing we have to implement is the Lorenz system without perturbations. Later, a system with perturbations is also implemented in order to calculate the Lyapunov exponent. We will use an ansatz where each device function calculates one particular realization of the Lorenz ensemble

```
struct lorenz_system
{
    struct lorenz_functor
    {
        template< class T >
        __host__ __device__
        void operator()( T t ) const
        {
            // unpack the parameter we want to vary and the Lorenz variables
            value_type R = thrust::get< 3 >( t );
            value_type x = thrust::get< 0 >( t );
            value_type y = thrust::get< 1 >( t );
            value_type z = thrust::get< 2 >( t );
            thrust::get< 4 >( t ) = sigma * ( y - x );
            thrust::get< 5 >( t ) = R * x - y - x * z;
            thrust::get< 6 >( t ) = -b * z + x * y ;

        }
    };

    lorenz_system( size_t N , const state_type &beta )
    : m_N( N ) , m_beta( beta ) { }

    template< class State , class Deriv >
    void operator()(  const State &x , Deriv &dxdt , value_type t ) const
    {
        thrust::for_each(
                thrust::make_zip_iterator( thrust::make_tuple(
                        boost::begin( x ) ,
                        boost::begin( x ) + m_N ,
                        boost::begin( x ) + 2 * m_N ,
                        m_beta.begin() ,
                        boost::begin( dxdt ) ,
                        boost::begin( dxdt ) + m_N ,
                        boost::begin( dxdt ) + 2 * m_N  ) ) ,
                thrust::make_zip_iterator( thrust::make_tuple(
                        boost::begin( x ) + m_N ,
                        boost::begin( x ) + 2 * m_N ,
                        boost::begin( x ) + 3 * m_N ,
                        m_beta.begin() ,
                        boost::begin( dxdt ) + m_N ,
                        boost::begin( dxdt ) + 2 * m_N ,
                        boost::begin( dxdt ) + 3 * m_N  ) ) ,
                lorenz_functor() );
    }
    size_t m_N;
    const state_type &m_beta;
};
```

As `state_type` a `thrust::device_vector` or a Boost.Range of a `device_vector` is used. The length of the state is *3N* where *N* is the number of systems. The system is encoded into this vector such that all *x* components come first, then every *y* components and finally every *z* components. Implementing the device function is then a simple task, you only have to decompose the tuple originating from the zip iterators.

Besides the system without perturbations we furthermore need to calculate the system including linearized equations governing the time evolution of small perturbations. Using the method from above this is straightforward, with a small difficulty that Thrust's tuples have a maximal arity of 10. But this is only a small problem since we can create a zip iterator packed with zip iterators. So the top level zip iterator contains one zip iterator for the state, one normal iterator for the parameter, and one zip iterator for the derivative. Accessing the elements of this tuple in the system function is then straightforward, you unpack the tuple with `thrust::get<>()`. We will not show the code here, it is to large. It can be found here and is easy to understand.

Furthermore, we need an observer which determines the norm of the perturbations, normalizes them and averages the logarithm of the norm. The device functor which is used within this observer is defined

```
struct lyap_functor
{
    template< class T >
    __host__ __device__
    void operator()( T t ) const
    {
        value_type &dx = thrust::get< 0 >( t );
        value_type &dy = thrust::get< 1 >( t );
        value_type &dz = thrust::get< 2 >( t );
        value_type norm = sqrt( dx * dx + dy * dy + dz * dz );
        dx /= norm;
        dy /= norm;
        dz /= norm;
        thrust::get< 3 >( t ) += log( norm );
    }
};
```

Note, that this functor manipulates the state, i.e. the perturbations.

Now we complete the whole code to calculate the Lyapunov exponents. First, we have to define a state vector. This vector contains *6N* entries, the state *x,y,z* and its perturbations *dx,dy,dz*. We initialize them such that *x=y=z=10*, *dx=1*, and *dy=dz=0*. We define a stepper type, a controlled Runge-Kutta Dormand-Prince 5 stepper. We start with some integration to overcome the transient behavior. For this, we do not involve the perturbation and run the algorithm only on the state *x,y,z* without any observer. Note, how Boost.Range is used for partial integration of the state vector without perturbations (the first half of the whole state). After the transient, the full system with perturbations is integrated and the Lyapunov exponents are calculated and written to `stdout`.

```
state_type x( 6 * N );

// initialize x,y,z
thrust::fill( x.begin() , x.begin() + 3 * N , 10.0 );

// initial dx
thrust::fill( x.begin() + 3 * N , x.begin() + 4 * N , 1.0 );

// initialize dy,dz
thrust::fill( x.begin() + 4 * N , x.end() , 0.0 );


// create error stepper, can be used with make_controlled or make_dense_output
typedef runge_kutta_dopri5< state_type , value_type , state_type , value_type , thrust_al↵
gebra , thrust_operations > stepper_type;


lorenz_system lorenz( N , beta );
lorenz_perturbation_system lorenz_perturbation( N , beta );
lyap_observer obs( N , 1 );

// calculate transients
integrate_adaptive( make_controlled( 1.0e-6 , 1.0e-6 , step↵
per_type() ) , lorenz , std::make_pair( x.begin() , x.begin() + 3 * N ) , 0.0 , 10.0 , dt );

// calculate the Lyapunov exponents -- the main loop
double t = 0.0;
while( t < 10000.0 )
{
    integrate_adaptive( make_controlled( 1.0e-6 , 1.0e-6 , stepper_type() ) , lorenz_perturba↵
tion , x , t , t + 1.0 , 0.1 );
    t += 1.0;
    obs( x , t );
}

vector< value_type > lyap( N );
obs.fill_lyap( lyap );

for( size_t i=0 ; i<N ; ++i )
    cout << beta_host[i] << "\t" << lyap[i] << "\n";
```

The full example can be found at lorenz_parameters.cu.

Using OpenCL via VexCL

In the previous section the usage of odeint in combination with Thrust was shown. In this section we show how one can use OpenCL with odeint. The point of odeint is not to implement its own low-level data structures and algorithms, but to use high level libraries doing this task. Here, we will use the VexCL framework to use OpenCL. VexCL is a nice library for general computations and it uses heavily expression templates. With the help of VexCL it is possible to write very compact and expressive application.

Note

vexcl needs C++11 features! So you have to compile with C++11 support enabled.

To use VexCL one needs to include one additional header which includes the data-types and algorithms from vexcl and the adaption to odeint. Adaption to odeint means here only to adapt the resizing functionality of VexCL to odeint.

```
#include <boost/numeric/odeint/external/vexcl/vexcl_resize.hpp>
```

To demonstrate the use of VexCL we integrate an ensemble of Lorenz system. The example is very similar to the parameter study of the Lorenz system in the previous section except that we do not compute the Lyapunov exponents. Again, we vary the parameter R of the Lorenz system an solve a whole ensemble of Lorenz systems in parallel (each with a different parameter R). First, we define the state type and a vector type

```
typedef vex::vector< double >    vector_type;
typedef vex::multivector< double, 3 > state_type;
```

The `vector_type` is used to represent the parameter R. The `state_type` is a multi-vector of three sub vectors and is used to represent. The first component of this multi-vector represent all `x` components of the Lorenz system, while the second all `y` components and the third all `z` components. The components of this vector can be obtained via

```
auto &x = X(0);
auto &y = X(1);
auto &z = X(2);
```

As already mentioned VexCL supports expression templates and we will use them to implement the system function for the Lorenz ensemble:

```
const double sigma = 10.0;
const double b = 8.0 / 3.0;

struct sys_func
{
    const vector_type &R;

    sys_func( const vector_type &_R ) : R( _R ) { }

    void operator()( const state_type &x , state_type &dxdt , double t ) const
    {
        dxdt(0) = -sigma * ( x(0) - x(1) );
        dxdt(1) = R * x(0) - x(1) - x(0) * x(2);
        dxdt(2) = - b * x(2) + x(0) * x(1);
    }
};
```

It's very easy, isn't it? These three little lines do all the computations for you. There is no need to write your own OpenCL kernels. VexCL does everything for you. Next we have to write the main application. We initialize the vector of parameters (R) and the initial state. Since VexCL supports odeint we can already use the `vector_space_algebra` in combination with the `default_operations` for the stepper and we are done:

```
// setup the opencl context
vex::Context ctx( vex::Filter::Type(CL_DEVICE_TYPE_GPU) );
std::cout << ctx << std::endl;

// set up number of system, time step and integration time
const size_t n = 1024 * 1024;
const double dt = 0.01;
const double t_max = 100.0;

// initialize R
double Rmin = 0.1 , Rmax = 50.0 , dR = ( Rmax - Rmin ) / double( n - 1 );
std::vector<double> x( n * 3 ) , r( n );
for( size_t i=0 ; i<n ; ++i ) r[i] = Rmin + dR * double( i );
vector_type R( ctx.queue() , r );

// initialize the state of the lorenz ensemble
state_type X(ctx.queue(), n);
X(0) = 10.0;
X(1) = 10.0;
X(2) = 10.0;

// create a stepper
runge_kutta4<
    state_type , double , state_type , double ,
    odeint::vector_space_algebra , odeint::default_operations
    > stepper;

// solve the system
integrate_const( stepper , sys_func( R ) , X , 0.0 , t_max , dt );
```

All examples

The following table gives an overview over all examples.

Table 4. Examples Overview

File	Brief Description
bind_member_functions.cpp	This examples shows how member functions can be used as system functions in odeint.
bind_member_functions_cpp11.cpp	This examples shows how member functions can be used as system functions in odeint with `std::bind` in C++11.
bulirsch_stoer.cpp	Shows the usage of the Bulirsch-Stoer method.
chaotic_system.cpp	The chaotic system examples integrates the Lorenz system and calculates the Lyapunov exponents.
elliptic_functions.cpp	Example calculating the elliptic functions using Bulirsch-Stoer and Runge-Kutta-Dopri5 Steppers with dense output.
fpu.cpp	The Fermi-Pasta-Ulam (FPU) example shows how odeint can be used to integrate lattice systems.
generation_functions.cpp	Shows skeletal code on how to implement own factory functions.
harmonic_oscillator.cpp	The harmonic oscillator examples gives a brief introduction to odeint and shows the usage of the classical Runge-Kutta-solvers.
harmonic_oscillator_units.cpp	This examples shows how Boost.Units can be used with odeint.
heun.cpp	The Heun example shows how an custom Runge-Kutta stepper can be created with odeint generic Runge-Kutta method.
list_lattice.cpp	Example of a phase lattice integration using `std::list` as state type.
lorenz_point.cpp	Alternative way of integrating lorenz by using a self defined point3d data type as state type.
my_vector.cpp	Simple example showing how to get odeint to work with a self-defined vector type.
phase_oscillator_ensemble.cpp	The phase oscillator ensemble example shows how globally coupled oscillators can be analyzed and how statistical measures can be computed during integration.
resizing_lattice.cpp	Shows the strength of odeint's memory management by simulating a Hamiltonian system on an expanding lattice.
simple1d.cpp	Integrating a simple, one-dimensional ODE showing the usage of integrate- and generate-functions.
solar_system.cpp	The solar system example shows the usage of the symplectic solvers.
stepper_details.cpp	Trivial example showing the usability of the several stepper classes.
stiff_system.cpp	The stiff system example shows the usage of the stiff solvers using the Jacobian of the system function.

File	Brief Description
stochastic_euler.cpp	Implementation of a custom stepper - the stochastic euler - for solving stochastic differential equations.
stuart_landau.cpp	The Stuart-Landau example shows how odeint can be used with complex state types.
two_dimensional_phase_lattice.cpp	The 2D phase oscillator example shows how a two-dimensional lattice works with odeint and how matrix types can be used as state types in odeint.
van_der_pol_stiff.cpp	This stiff system example again shows the usage of the stiff solvers by integrating the van der Pol oscillator.
gmpxx/lorenz_gmpxx.cpp	This examples integrates the Lorenz system by means of an arbitrary precision type.
mtl/gauss_packet.cpp	The MTL-Gauss-packet example shows how the MTL can be easily used with odeint.
mtl/implicit_euler_mtl.cpp	This examples shows the usage of the MTL implicit Euler method with a sparse matrix type.
thrust/phase_oscillator_ensemble.cu	The Thrust phase oscillator ensemble example shows how globally coupled oscillators can be analyzed with Thrust and CUDA, employing the power of modern graphic devices.
thrust/phase_oscillator_chain.cu	The Thrust phase oscillator chain example shows how chains of nearest neighbor coupled oscillators can be integrated with Thrust and odeint.
thrust/lorenz_parameters.cu	The Lorenz parameters examples show how ensembles of ordinary differential equations can be solved by means of Thrust to study the dependence of an ODE on some parameters.
thrust/relaxation.cu	Another examples for the usage of Thrust.
ublas/lorenz_ublas.cpp	This example shows how the ublas vector types can be used with odeint.
vexcl/lorenz_ensemble.cpp	This example shows how the VexCL - a framework for OpenCL computation - can be used with odeint.
2d_lattice/spreading.cpp	This examples shows how a `vector< vector< T > >` can be used a state type for odeint and how a resizing mechanism of this state can be implemented.
quadmath/black_hole.cpp	This examples shows how gcc libquadmath can be used with odeint. It provides a high precision floating point type which is adapted to odeint in this example.

odeint in detail

Steppers

Solving ordinary differential equation numerically is usually done iteratively, that is a given state of an ordinary differential equation is iterated forward *x(t) -> x(t+dt) -> x(t+2dt)*. The steppers in odeint perform one single step. The most general stepper type is described by the Stepper concept. The stepper concepts of odeint are described in detail in section Concepts, here we briefly present the mathematical and numerical details of the steppers. The Stepper has two versions of the `do_step` method, one with an in-place transform of the current state and one with an out-of-place transform:

```
do_step( sys , inout , t , dt )

do_step( sys , in , t , out , dt )
```

The first parameter is always the system function - a function describing the ODE. In the first version the second parameter is the step which is here updated in-place and the third and the fourth parameters are the time and step size (the time step). After a call to `do_step` the state `inout` is updated and now represents an approximate solution of the ODE at time *t+dt*. In the second version the second argument is the state of the ODE at time *t*, the third argument is t, the fourth argument is the approximate solution at time *t+dt* which is filled by `do_step` and the fifth argument is the time step. Note that these functions do not change the time `t`.

System functions

Up to now, we have nothing said about the system function. This function depends on the stepper. For the explicit Runge-Kutta steppers this function can be a simple callable object hence a simple (global) C-function or a functor. The parameter syntax is `sys( x , dxdt , t )` and it is assumed that it calculates *dx/dt = f(x,t)*. The function structure in most cases looks like:

```
void sys( const state_type & /*x*/ , state_type & /*dxdt*/ , const double /*t*/ )
{
    // ...
}
```

Other types of system functions might represent Hamiltonian systems or systems which also compute the Jacobian needed in implicit steppers. For information which stepper uses which system function see the stepper table below. It might be possible that odeint will introduce new system types in near future. Since the system function is strongly related to the stepper type, such an introduction of a new stepper might result in a new type of system function.

Explicit steppers

A first specialization are the explicit steppers. Explicit means that the new state of the ode can be computed explicitly from the current state without solving implicit equations. Such steppers have in common that they evaluate the system at time *t* such that the result of *f(x,t)* can be passed to the stepper. In odeint, the explicit stepper have two additional methods

```
do_step( sys , inout , dxdtin , t , dt )

do_step( sys , in , dxdtin , t , out , dt )
```

Here, the additional parameter is the value of the function *f* at state *x* and time *t*. An example is the Runge-Kutta stepper of fourth order:

```
runge_kutta4< state_type > rk;
rk.do_step( sys1 , inout , t , dt );              // In-place transformation of inout
rk.do_step( sys2 , inout , t , dt );              // call with different system: Ok
rk.do_step( sys1 , in , t , out , dt );           // Out-of-place transformation
rk.do_step( sys1 , inout , dxdtin , t , dt );     // In-place tranformation of inout
rk.do_step( sys1 , in , dxdtin , t , out , dt );  // Out-of-place transformation
```

In fact, you do not need to call these two methods. You can always use the simpler `do_step( sys , inout , t , dt )`, but sometimes the derivative of the state is needed externally to do some external computations or to perform some statistical analysis.

A special class of the explicit steppers are the FSAL (first-same-as-last) steppers, where the last evaluation of the system function is also the first evaluation of the following step. For such steppers the `do_step` method are slightly different:

```
do_step( sys , inout , dxdtinout , t , dt )

do_step( sys , in , dxdtin , out , dxdtout , t , dt )
```

This method takes the derivative at time `t` and also stores the derivative at time *t+dt*. Calling these functions subsequently iterating along the solution one saves one function call by passing the result for dxdt into the next function call. However, when using FSAL steppers without supplying derivatives:

```
do_step( sys , inout , t , dt )
```

the stepper internally satisfies the FSAL property which means it remembers the last `dxdt` and uses it for the next step. An example for a FSAL stepper is the Runge-Kutta-Dopri5 stepper. The FSAL trick is sometimes also referred as the Fehlberg trick. An example how the FSAL steppers can be used is

```
runge_kutta_dopri5< state_type > rk;
rk.do_step( sys1 , in , t , out , dt );
rk.do_step( sys2 , in , t , out , dt );         // DONT do this, sys1 is assumed

rk.do_step( sys2 , in2 , t , out , dt );
rk.do_step( sys2 , in3 , t , out , dt );        // DONT do this, in2 is assumed

rk.do_step( sys1 , inout , dxdtinout , t , dt );
rk.do_step( sys2 , inout , dxdtinout , t , dt );           // Ok, internal derivative is not ↵
used, dxdtinout is updated

rk.do_step( sys1 , in , dxdtin , t , out , dxdtout , dt );
rk.do_step( sys2 , in , dxdtin , t , out , dxdtout , dt ); // Ok, internal derivative is not used
```

Caution

The FSAL-steppers save the derivative at time *t+dt* internally if they are called via `do_step( sys , in , out , t , dt )`. The first call of `do_step` will initialize `dxdt` and for all following calls it is assumed that the same system and the same state are used. If you use the FSAL stepper within the integrate functions this is taken care of automatically. See the Using steppers section for more details or look into the table below to see which stepper have an internal state.

Symplectic solvers

As mentioned above symplectic solvers are used for Hamiltonian systems. Symplectic solvers conserve the phase space volume exactly and if the Hamiltonian system is energy conservative they also conserve the energy approximately. A special class of symplectic systems are separable systems which can be written in the form *dqdt/dt = f1(p)*, *dpdt/dt = f2(q)*, where *(q,p)* are the state of system. The space of *(q,p)* is sometimes referred as the phase space and *q* and *p* are said the be the phase space variables. Symplectic systems in this special form occur widely in nature. For example the complete classical mechanics as written down by Newton, Lagrange and Hamilton can be formulated in this framework. The separability of the system depends on the specific choice of coordinates.

Symplectic systems can be solved by odeint by means of the symplectic_euler stepper and a symplectic Runge-Kutta-Nystrom method of fourth order. These steppers assume that the system is autonomous, hence the time will not explicitly occur. Further they fulfill in principle the default Stepper concept, but they expect the system to be a pair of callable objects. The first entry of this pair calculates *f1(p)* while the second calculates *f2(q)*. The syntax is `sys.first(p,dqdt)` and `sys.second(q,dpdt)`, where the first and second part can be again simple C-functions of functors. An example is the harmonic oscillator:

```
typedef boost::array< double , 1 > vector_type;


struct harm_osc_f1
{
    void operator()( const vector_type &p , vector_type &dqdt )
    {
        dqdt[0] = p[0];
    }
};

struct harm_osc_f2
{
    void operator()( const vector_type &q , vector_type &dpdt )
    {
        dpdt[0] = -q[0];
    }
};
```

The state of such an ODE consist now also of two parts, the part for q (also called the coordinates) and the part for p (the momenta). The full example for the harmonic oscillator is now:

```
pair< vector_type , vector_type > x;
x.first[0] = 1.0; x.second[0] = 0.0;
symplectic_rkn_sb3a_mclachlan< vector_type > rkn;
rkn.do_step( make_pair( harm_osc_f1() , harm_osc_f2() ) , x , t , dt );
```

If you like to represent the system with one class you can easily bind two public method:

```
struct harm_osc
{
    void f1( const vector_type &p , vector_type &dqdt ) const
    {
        dqdt[0] = p[0];
    }

    void f2( const vector_type &q , vector_type &dpdt ) const
    {
        dpdt[0] = -q[0];
    }
};
```

```
harm_osc h;
rkn.do_step( make_pair( boost::bind( &harm_osc::f1 , h , _1 , _2 ) , boost::bind( &harm_osc::f2 , h , _1 , _2 ) ) ,
        x , t , dt );
```

Many Hamiltonian system can be written as *dq/dt=p*, *dp/dt=f(q)* which is computationally much easier than the full separable system. Very often, it is also possible to transform the original equations of motion to bring the system in this simplified form. This kind of system can be used in the symplectic solvers, by simply passing *f(p)* to the `do_step` method, again *f(p)* will be represented by a simple C-function or a functor. Here, the above example of the harmonic oscillator can be written as

```
pair< vector_type , vector_type > x;
x.first[0] = 1.0; x.second[0] = 0.0;
symplectic_rkn_sb3a_mclachlan< vector_type > rkn;
rkn.do_step( harm_osc_f1() , x , t , dt );
```

In this example the function `harm_osc_f1` is exactly the same function as in the above examples.

Note, that the state of the ODE must not be constructed explicitly via `pair< vector_type , vector_type > x`. One can also use a combination of `make_pair` and `ref`. Furthermore, a convenience version of `do_step` exists which takes q and p without combining them into a pair:

```
rkn.do_step( harm_osc_f1() , make_pair( boost::ref( q ) , boost::ref( p ) ) , t , dt );
rkn.do_step( harm_osc_f1() , q , p , t , dt );
rkn.do_step( make_pair( harm_osc_f1() , harm_osc_f2() ) , q , p , t , dt );
```

Implicit solvers

Caution

This section is not up-to-date.

For some kind of systems the stability properties of the classical Runge-Kutta are not sufficient, especially if the system is said to be stiff. A stiff system possesses two or more time scales of very different order. Solvers for stiff systems are usually implicit, meaning that they solve equations like *x(t+dt) = x(t) + dt * f(x(t+1))*. This particular scheme is the implicit Euler method. Implicit methods usually solve the system of equations by a root finding algorithm like the Newton method and therefore need to know the Jacobian of the system $J_{ij} = df_i / dx_j$.

For implicit solvers the system is again a pair, where the first component computes *f(x,t)* and the second the Jacobian. The syntax is `sys.first( x , dxdt , t )` and `sys.second( x , J , t )`. For the implicit solver the `state_type` is `ublas::vector` and the Jacobian is represented by `ublas::matrix`.

Important

Implicit solvers only work with ublas::vector as state type. At the moment, no other state types are supported.

Multistep methods

Another large class of solvers are multi-step method. They save a small part of the history of the solution and compute the next step with the help of this history. Since multi-step methods know a part of their history they do not need to compute the system function very often, usually it is only computed once. This makes multi-step methods preferable if a call of the system function is expensive. Examples are ODEs defined on networks, where the computation of the interaction is usually where expensive (and might be of order O(N^2)).

Multi-step methods differ from the normal steppers. They save a part of their history and this part has to be explicitly calculated and initialized. In the following example an Adams-Bashforth-stepper with a history of 5 steps is instantiated and initialized;

```
adams_bashforth_moulton< 5 , state_type > abm;
abm.initialize( sys , inout , t , dt );
abm.do_step( sys , inout , t , dt );
```

The initialization uses a fourth-order Runge-Kutta stepper and after the call of `initialize` the state of `inout` has changed to the current state, such that it can be immediately used by passing it to following calls of `do_step`. You can also use you own steppers to initialize the internal state of the Adams-Bashforth-Stepper:

```
abm.initialize( runge_kutta_fehlberg78< state_type >() , sys , inout , t , dt );
```

Many multi-step methods are also explicit steppers, hence the parameter of `do_step` method do not differ from the explicit steppers.

Caution

The multi-step methods have some internal variables which depend on the explicit solution. Hence after any external changes of your state (e.g. size) or system the initialize function has to be called again to adjust the internal state of the stepper. If you use the integrate functions this will be taken into account. See the Using steppers section for more details.

Controlled steppers

Many of the above introduced steppers possess the possibility to use adaptive step-size control. Adaptive step size integration works in principle as follows:

1. The error of one step is calculated. This is usually done by performing two steps with different orders. The difference between these two steps is then used as a measure for the error. Stepper which can calculate the error are Error Stepper and they form an own class with an separate concept.

2. This error is compared against some predefined error tolerances. Are the tolerance violated the step is reject and the step-size is decreases. Otherwise the step is accepted and possibly the step-size is increased.

The class of controlled steppers has their own concept in odeint - the Controlled Stepper concept. They are usually constructed from the underlying error steppers. An example is the controller for the explicit Runge-Kutta steppers. The Runge-Kutta steppers enter the controller as a template argument. Additionally one can pass the Runge-Kutta stepper to the constructor, but this step is not necessary; the stepper is default-constructed if possible.

Different step size controlling mechanism exist. They all have in common that they somehow compare predefined error tolerance against the error and that they might reject or accept a step. If a step is rejected the step size is usually decreased and the step is made again with the reduced step size. This procedure is repeated until the step is accepted. This algorithm is implemented in the integration functions.

A classical way to decide whether a step is rejected or accepted is to calculate

$$val = || \, | \, err_i \, | \, / \, (\, \varepsilon_{abs} + \varepsilon_{rel} \, * \, (\, a_x \, | \, x_i \, | \, + \, a_{dxdt} \, | \, | \, dxdt_i \, | \,) ||$$

ε_{abs} and ε_{rel} are the absolute and the relative error tolerances, and $|| \, x \, ||$ is a norm, typically $||x|| = (\Sigma_i \, x_i^2)^{1/2}$ or the maximum norm. The step is rejected if *val* is greater then 1, otherwise it is accepted. For details of the used norms and error tolerance see the table below.

For the `controlled_runge_kutta` stepper the new step size is then calculated via

$$val > 1 : dt_{new} = dt_{current} \, max(\, 0.9 \, pow(\, val \, , \, -1 \, / \, (\, O_E \, - \, 1 \,) \,) \, , \, 0.2 \,)$$

$$val < 0.5 : dt_{new} = dt_{current} \, min(\, 0.9 \, pow(\, val \, , \, -1 \, / \, O_S \,) \, , \, 5 \,)$$

$$else : dt_{new} = dt_{current}$$

Here, O_S and O_E are the order of the stepper and the error stepper. These formulas also contain some safety factors, avoiding that the step size is reduced or increased to much. For details of the implementations of the controlled steppers in odeint see the table below.

Table 5. Adaptive step size algorithms

Stepper	Tolerance formula	Norm	Step size adaption
`controlled_runge_kutta`	$val = \|\| \| err_i \| / (\varepsilon_{abs} + \varepsilon_{rel} * (a_x \| x_i \| + a_{dxdt} \| \| dxdt_i \|) \|\|$	$\|\|x\\|\| = max(x_i)$	$val > 1 : dt_{new} = dt_{current} max(0.9 pow(val , -1 / (O_E - 1)) , 0.2)$ $val < 0.5 : dt_{new} = dt_{current} min(0.9 pow(val , -1 / O_S) , 5)$ $else : dt_{new} = dt_{current}$
`rosenbrock4_controller`	$val = \|\| err_i / (\varepsilon_{abs} + \varepsilon_{rel} max(\| x_i \| , \| xold_i \|)) \|\|$	$\|\|x\\|\|=(\Sigma_i x_i^2)^{1/2}$	$fac = max(1 / 6 , min(5 , pow(val , 1 / 4) / 0.9)$ $fac2 = max(1 / 6 , min(5 , dt_{old} / dt_{current} pow(val^2 / val_{old} , 1 / 4) / 0.9)$ $val > 1 : dt_{new} = dt_{current} / fac$ $val < 1 : dt_{new} = dt_{current} / max(fac , fac2)$
bulirsch_stoer	$tol=1/2$	-	$dt_{new} = dt_{old}^{1/a}$

To ease to generation of the controlled stepper, generation functions exist which take the absolute and relative error tolerances and a predefined error stepper and construct from this knowledge an appropriate controlled stepper. The generation functions are explained in detail in Generation functions.

Dense output steppers

A fourth class of stepper exists which are the so called dense output steppers. Dense-output steppers might take larger steps and interpolate the solution between two consecutive points. This interpolated points have usually the same order as the order of the stepper. Dense-output steppers are often composite stepper which take the underlying method as a template parameter. An example is the `dense_output_runge_kutta` stepper which takes a Runge-Kutta stepper with dense-output facilities as argument. Not all Runge-Kutta steppers provide dense-output calculation; at the moment only the Dormand-Prince 5 stepper provides dense output. An example is

```
dense_output_runge_kutta< controlled_runge_kutta< runge_kutta_dopri5< state_type > > > dense;
dense.initialize( in , t , dt );
pair< double , double > times = dense.do_step( sys );
```

Dense output stepper have their own concept. The main difference to usual steppers is that they manage the state and time internally. If you call `do_step`, only the ODE is passed as argument. Furthermore `do_step` return the last time interval: `t` and `t+dt`, hence you can interpolate the solution between these two times points. Another difference is that they must be initialized with `initialize`, otherwise the internal state of the stepper is default constructed which might produce funny errors or bugs.

The construction of the dense output stepper looks a little bit nasty, since in the case of the `dense_output_runge_kutta` stepper a controlled stepper and an error stepper have to be nested. To simplify the generation of the dense output stepper generation functions exist:

```
typedef boost::numeric::odeint::result_of::make_dense_output<
    runge_kutta_dopri5< state_type > >::type dense_stepper_type;
dense_stepper_type dense2 = make_dense_output( 1.0e-6 , 1.0e-
6 , runge_kutta_dopri5< state_type >() );
```

This statement is also lengthy; it demonstrates how `make_dense_output` can be used with the `result_of` protocol. The parameters to `make_dense_output` are the absolute error tolerance, the relative error tolerance and the stepper. This explicitly assumes that the underlying stepper is a controlled stepper and that this stepper has an absolute and a relative error tolerance. For details about the generation functions see Generation functions. The generation functions have been designed for easy use with the integrate functions:

```
integrate_const( make_dense_output( 1.0e-6 , 1.0e-
6 , runge_kutta_dopri5< state_type >() ) , sys , inout , t_start , t_end , dt );
```

Using steppers

This section contains some general information about the usage of the steppers in odeint.

Steppers are copied by value

The stepper in odeint are always copied by values. They are copied for the creation of the controlled steppers or the dense output steppers as well as in the integrate functions.

Steppers might have a internal state

Caution

Some of the features described in this section are not yet implemented

Some steppers require to store some information about the state of the ODE between two steps. Examples are the multi-step methods which store a part of the solution during the evolution of the ODE, or the FSAL steppers which store the last derivative at time $t+dt$, to be used in the next step. In both cases the steppers expect that consecutive calls of `do_step` are from the same solution and the same ODE. In this case it is absolutely necessary that you call `do_step` with the same system function and the same state, see also the examples for the FSAL steppers above.

Stepper with an internal state support two additional methods: `reset` which resets the state and `initialize` which initializes the internal state. The parameters of `initialize` depend on the specific stepper. For example the Adams-Bashforth-Moulton stepper provides two initialize methods: `initialize( system , inout , t , dt )` which initializes the internal states with the help of the Runge-Kutta 4 stepper, and `initialize( stepper , system , inout , t , dt )` which initializes with the help of `stepper`. For the case of the FSAL steppers, `initialize` is `initialize( sys , in , t )` which simply calculates the r.h.s. of the ODE and assigns its value to the internal derivative.

All these steppers have in common, that they initially fill their internal state by themselves. Hence you are not required to call initialize. See how this works for the Adams-Bashforth-Moulton stepper: in the example we instantiate a fourth order Adams-Bashforth-Moulton stepper, meaning that it will store 4 internal derivatives of the solution at times `(t-dt,t-2*dt,t-3*dt,t-4*dt)`.

```
adams_bashforth_moulton< 4 , state_type > stepper;
stepper.do_step( sys , x , t , dt );   // make one step with the classical Runge-Kutta stepper ↵
and initialize the first internal state
                                       // the internal array is now [x(t-dt)]

stepper.do_step( sys , x , t , dt );   // make one step with the classical Runge-Kutta stepper ↵
and initialize the second internal state
                                       // the internal state array is now [x(t-dt), x(t-2*dt)]

stepper.do_step( sys , x , t , dt );   // make one step with the classical Runge-Kutta stepper ↵
and initialize the third internal state
                                       // the internal state array is now [x(t-dt), x(t-
2*dt), x(t-3*dt)]

stepper.do_step( sys , x , t , dt );   // make one step with the classical Runge-Kutta stepper ↵
and initialize the fourth internal state
                                       // the internal state array is now [x(t-dt), x(t-
2*dt), x(t-3*dt), x(t-4*dt)]

stepper.do_step( sys , x , t , dt );   // make one step with Adam-Bashforth-Moulton, the intern↵
al array of states is now rotated
```

In the stepper table at the bottom of this page one can see which stepper have an internal state and hence provide the `reset` and `initialize` methods.

Stepper might be resizable

Nearly all steppers in odeint need to store some intermediate results of the type `state_type` or `deriv_type`. To do so odeint need some memory management for the internal temporaries. As this memory management is typically related to adjusting the size of vector-like types, it is called resizing in odeint. So, most steppers in odeint provide an additional template parameter which controls the size adjustment of the internal variables - the resizer. In detail odeint provides three policy classes (resizers) `always_resizer`, `initially_resizer`, and `never_resizer`. Furthermore, all stepper have a method `adjust_size` which takes a parameter representing a state type and which manually adjusts the size of the internal variables matching the size of the given instance. Before performing the actual resizing odeint always checks if the sizes of the state and the internal variable differ and only resizes if they are different.

Note

You only have to worry about memory allocation when using dynamically sized vector types. If your state type is heap allocated, like `boost::array`, no memory allocation is required whatsoever.

By default the resizing parameter is `initially_resizer`, meaning that the first call to `do_step` performs the resizing, hence memory allocation. If you have changed the size of your system and your state you have to call `adjust_size` by hand in this case. The second resizer is the `always_resizer` which tries to resize the internal variables at every call of `do_step`. Typical use cases for this kind of resizer are self expanding lattices like shown in the tutorial (Self expanding lattices) or partial differential equations with an adaptive grid. Here, no calls of `adjust_size` are required, the steppers manage everything themselves. The third class of resizer is the `never_resizer` which means that the internal variables are never adjusted automatically and always have to be adjusted by hand .

There is a second mechanism which influences the resizing and which controls if a state type is at least resizeable - a meta-function `is_resizeable`. This meta-function returns a static Boolean value if any type is resizable. For example it will return `true` for `std::vector< T >` but `false` for `boost::array< T >`. By default and for unknown types `is_resizeable` returns `false`, so if you have your own type you need to specialize this meta-function. For more details on the resizing mechanism see the section Adapt your own state types.

Which steppers should be used in which situation

odeint provides a quite large number of different steppers such that the user is left with the question of which stepper fits his needs. Our personal recommendations are:

- `runge_kutta_dopri5` is maybe the best default stepper. It has step size control as well as dense-output functionality. Simple create a dense-output stepper by `make_dense_output( 1.0e-6 , 1.0e-5 , runge_kutta_dopri5< state_type >() )`.
- `runge_kutta4` is a good stepper for constant step sizes. It is widely used and very well known. If you need to create artificial time series this stepper should be the first choice.
- 'runge_kutta_fehlberg78' is similar to the 'runge_kutta4' with the advantage that it has higher precision. It can also be used with step size control.
- `adams_bashforth_moulton` is very well suited for ODEs where the r.h.s. is expensive (in terms of computation time). It will calculate the system function only once during each step.

Stepper overview

Table 6. Stepper Algorithms

Algorithm	Class	Concept	System Concept	Order	Error Estimation	Dense Output	Internal state	Remarks
Explicit Euler	`euler`	Dense Output Stepper	System	1	No	Yes	No	Very simple, only for demonstrating purpose
Modified Midpoint	`modified_midpoint`	Stepper	System	configurable (2)	No	No	No	Used in Bulirsch-Stoer implementation
Runge-Kutta 4	`runge_kutta4`	Stepper	System	4	No	No	No	The classical Runge-Kutta scheme, good general scheme without error control
Cash-Karp	`runge_kutta_cash_karp54`	Error Stepper	System	5	Yes (4)	No	No	Good general scheme with error estimation, to be used in controlled_error_stepper
Dormand-Prince 5	`runge_kutta_dopri5`	Error Stepper	System	5	Yes (4)	Yes	Yes	Standard method with error control and dense output, to be used in controlled_error_stepper and in dense_output_controlled_explicit_fsal.
Fehlberg 78	`runge_kutta_fehlberg78`	Error Stepper	System	8	Yes (7)	No	No	Good high order method with error estimation, to be used in controlled_error_stepper.

Algorithm	Class	Concept	System Concept	Order	Error Estimation	Dense Output	Internal state	Remarks
Adams Bashforth	adams_bashforth	Stepper	System	configurable	No	No	Yes	Multistep method
Adams Moulton	adams_moulton	Stepper	System	configurable	No	No	Yes	Multistep method
Adams Bashforth Moulton	adams_bashforth_moulton	Stepper	System	configurable	No	No	Yes	Combined multistep method
Controlled Runge-Kutta	controlled_runge_kutta	Controlled Stepper	System	depends	Yes	No	depends	Error control for Error Stepper. Requires an Error Stepper from above. Order depends on the given Error-Stepper
Dense Output Runge-Kutta	dense_output_runge_kutta	Dense Output Stepper	System	depends	No	Yes	Yes	Dense output for Stepper and Error Stepper from above if they provide dense output functionality (like euler and runge_kutta_dopri5) Order depends on the given stepper.
Bulirsch-Stoer	bulirsch_stoer	Controlled Stepper	System	variable	Yes	No	No	Stepper with step size and order control. Very good if high precision is required.

Algorithm	Class	Concept	System Concept	Order	Error Estimation	Dense Output	Internal state	Remarks
Bulirsch-Stoer Dense Output	bulirsch_stoer_dense_out	Dense Output Stepper	System	variable	Yes	Yes	No	Stepper with step size and order control as well as dense output. Very good if high precision and dense output is required.
Implicit Euler	implicit_euler	Stepper	Implicit System	1	No	No	No	Basic implicit routine. Requires the Jacobian. Works only with BoostuBLAS vectors as state types.
Rosenbrock 4	rosenbrock4	Error Stepper	Implicit System	4	Yes	Yes	No	Good for stiff systems. Works only with BoostuBLAS vectors as state types.
Controlled Rosenbrock 4	rosenbrock4_controller	Controlled Stepper	Implicit System	4	Yes	Yes	No	Rosenbrock 4 with error control. Works only with BoostuBLAS vectors as state types.
Dense Output Rosenbrock 4	rosenbrock4_dense_output	Dense Output Stepper	Implicit System	4	Yes	Yes	No	Controlled Rosenbrock 4 with dense output. Works only with BoostuBLAS vectors as state types.

Algorithm	Class	Concept	System Concept	Order	Error Estimation	Dense Output	Internal state	Remarks
Symplectic Euler	`symplectic_euler`	Stepper	Symplectic System Simple Symplectic System	1	No	No	No	Basic symplectic solver for separable Hamiltonian system
Symplectic RKN McLachlan	`symplectic_rkn_sb3a_mclachlan`	Stepper	Symplectic System Simple Symplectic System	4	No	No	No	Symplectic solver for separable Hamiltonian system with 6 stages and order 4.
Symplectic RKN McLachlan	`symplectic_rkn_sb3a_m4_mclachlan`	Stepper	Symplectic System Simple Symplectic System	4	No	No	No	Symplectic solver with 5 stages and order 4, can be used with arbitrary precision types.

Custom steppers

Finally, one can also write new steppers which are fully compatible with odeint. They only have to fulfill one or several of the stepper Concepts of odeint.

We will illustrate how to write your own stepper with the example of the stochastic Euler method. This method is suited to solve stochastic differential equations (SDEs). A SDE has the form

$dx/dt = f(x) + g(x)\ \xi(t)$

where ξ is Gaussian white noise with zero mean and a standard deviation $\sigma(t)$. $f(x)$ is said to be the deterministic part while $g(x)\ \xi$ is the noisy part. In case $g(x)$ is independent of x the SDE is said to have additive noise. It is not possible to solve SDE with the classical solvers for ODEs since the noisy part of the SDE has to be scaled differently then the deterministic part with respect to the time step. But there exist many solvers for SDEs. A classical and easy method is the stochastic Euler solver. It works by iterating

$x(t+\Delta t) = x(t) + \Delta t\ f(x(t)) + \Delta t^{1/2}\ g(x)\ \xi(t)$

where $\xi(t)$ is an independent normal distributed random variable.

Now we will implement this method. We will call the stepper `stochastic_euler`. It models the Stepper concept. For simplicity, we fix the state type to be an `array< double , N >` The class definition looks like

```
template< size_t N > class stochastic_euler
{
public:

    typedef boost::array< double , N > state_type;
    typedef boost::array< double , N > deriv_type;
    typedef double value_type;
    typedef double time_type;
    typedef unsigned short order_type;
    typedef boost::numeric::odeint::stepper_tag stepper_category;

    static order_type order( void ) { return 1; }

    // ...
};
```

The types are needed in order to fulfill the stepper concept. As internal state and deriv type we use simple arrays in the stochastic Euler, they are needed for the temporaries. The stepper has the order one which is returned from the `order()` function.

The system functions needs to calculate the deterministic and the stochastic part of our stochastic differential equation. So it might be suitable that the system function is a pair of functions. The first element of the pair computes the deterministic part and the second the stochastic one. Then, the second part also needs to calculate the random numbers in order to simulate the stochastic process. We can now implement the `do_step` method

```
template< size_t N > class stochastic_euler
{
public:

    // ...

    template< class System >
    void do_step( System system , state_type &x , time_type t , time_type dt ) const
    {
        deriv_type det , stoch ;
        system.first( x , det );
        system.second( x , stoch );
        for( size_t i=0 ; i<x.size() ; ++i )
            x[i] += dt * det[i] + sqrt( dt ) * stoch[i];
    }
};
```

This is all. It is quite simple and the stochastic Euler stepper implement here is quite general. Of course it can be enhanced, for example

- use of operations and algebras as well as the resizing mechanism for maximal flexibility and portability
- use of `boost::ref` for the system functions
- use of `boost::range` for the state type in the `do_step` method
- ...

Now, lets look how we use the new stepper. A nice example is the Ornstein-Uhlenbeck process. It consists of a simple Brownian motion overlapped with an relaxation process. Its SDE reads

$dx/dt = - x + \xi$

where ξ is Gaussian white noise with standard deviation σ. Implementing the Ornstein-Uhlenbeck process is quite simple. We need two functions or functors - one for the deterministic and one for the stochastic part:

```
const static size_t N = 1;
typedef boost::array< double , N > state_type;

struct ornstein_det
{
    void operator()( const state_type &x , state_type &dxdt ) const
    {
        dxdt[0] = -x[0];
    }
};

struct ornstein_stoch
{
    boost::mt19937 m_rng;
    boost::normal_distribution<> m_dist;

    ornstein_stoch( double sigma ) : m_rng() , m_dist( 0.0 , sigma ) { }

    void operator()( const state_type &x , state_type &dxdt )
    {
        dxdt[0] = m_dist( m_rng );
    }
};
```

In the stochastic part we have used the Mersenne twister for the random number generation and a Gaussian white noise generator `normal_distribution` with standard deviation σ. Now, we can use the stochastic Euler stepper with the integrate functions:

```
double dt = 0.1;
state_type x = {{ 1.0 }};
integrate_const( stochastic_euler< N >() , make_pair( ornstein_det() , ornstein_stoch( 1.0 ) ) ,
        x , 0.0 , 10.0 , dt , streaming_observer() );
```

Note, how we have used the `make_pair` function for the generation of the system function.

Custom Runge-Kutta steppers

odeint provides a C++ template meta-algorithm for constructing arbitrary Runge-Kutta schemes [1]. Some schemes are predefined in odeint, for example the classical Runge-Kutta of fourth order, or the Runge-Kutta-Cash-Karp 54 and the Runge-Kutta-Fehlberg 78 method. You can use this meta algorithm to construct you own solvers. This has the advantage that you can make full use of odeint's algebra and operation system.

Consider for example the method of Heun, defined by the following Butcher tableau:

```
c1 = 0

c2 = 1/3, a21 = 1/3

c3 = 2/3, a31 =  0 , a32 = 2/3

          b1  = 1/4, b2  = 0  , b3 = 3/4
```

Implementing this method is very easy. First you have to define the constants:

[1] M. Mulansky, K. Ahnert, Template-Metaprogramming applied to numerical problems, arxiv:1110.3233

```
template< class Value = double >
struct heun_a1 : boost::array< Value , 1 > {
    heun_a1( void )
    {
        (*this)[0] = static_cast< Value >( 1 ) / static_cast< Value >( 3 );
    }
};

template< class Value = double >
struct heun_a2 : boost::array< Value , 2 >
{
    heun_a2( void )
    {
        (*this)[0] = static_cast< Value >( 0 );
        (*this)[1] = static_cast< Value >( 2 ) / static_cast< Value >( 3 );
    }
};


template< class Value = double >
struct heun_b : boost::array< Value , 3 >
{
    heun_b( void )
    {
        (*this)[0] = static_cast<Value>( 1 ) / static_cast<Value>( 4 );
        (*this)[1] = static_cast<Value>( 0 );
        (*this)[2] = static_cast<Value>( 3 ) / static_cast<Value>( 4 );
    }
};

template< class Value = double >
struct heun_c : boost::array< Value , 3 >
{
    heun_c( void )
    {
        (*this)[0] = static_cast< Value >( 0 );
        (*this)[1] = static_cast< Value >( 1 ) / static_cast< Value >( 3 );
        (*this)[2] = static_cast< Value >( 2 ) / static_cast< Value >( 3 );
    }
};
```

While this might look cumbersome, packing all parameters into a templatized class which is not immediately evaluated has the advantage that you can change the `value_type` of your stepper to any type you like - presumably arbitrary precision types. One could also instantiate the coefficients directly

```
const boost::array< double , 1 > heun_a1 = {{ 1.0 / 3.0 }};
const boost::array< double , 2 > heun_a2 = {{ 0.0 , 2.0 / 3.0 }};
const boost::array< double , 3 > heun_b = {{ 1.0 / 4.0 , 0.0 , 3.0 / 4.0 }};
const boost::array< double , 3 > heun_c = {{ 0.0 , 1.0 / 3.0 , 2.0 / 3.0 }};
```

But then you are nailed down to use doubles.

Next, you need to define your stepper, note that the Heun method has 3 stages and produces approximations of order 3:

```
template<
    class State ,
    class Value = double ,
    class Deriv = State ,
    class Time = Value ,
    class Algebra = boost::numeric::odeint::range_algebra ,
    class Operations = boost::numeric::odeint::default_operations ,
    class Resizer = boost::numeric::odeint::initially_resizer
>
class heun : public
boost::numeric::odeint::explicit_generic_rk< 3 , 3 , State , Value , Deriv , Time ,
                                             Algebra , Operations , Resizer >
{

public:

    typedef boost::numeric::odeint::explicit_generic_rk< 3 , 3 , State , Value , Deriv , Time ,
                                                         Algebra , Operations , Resizer > step↵
per_base_type;

    typedef typename stepper_base_type::state_type state_type;
    typedef typename stepper_base_type::wrapped_state_type wrapped_state_type;
    typedef typename stepper_base_type::value_type value_type;
    typedef typename stepper_base_type::deriv_type deriv_type;
    typedef typename stepper_base_type::wrapped_deriv_type wrapped_deriv_type;
    typedef typename stepper_base_type::time_type time_type;
    typedef typename stepper_base_type::algebra_type algebra_type;
    typedef typename stepper_base_type::operations_type operations_type;
    typedef typename stepper_base_type::resizer_type resizer_type;
    typedef typename stepper_base_type::stepper_type stepper_type;

    heun( const algebra_type &algebra = algebra_type() )
    : stepper_base_type(
            fusion::make_vector(
                heun_a1<Value>() ,
                heun_a2<Value>() ) ,
            heun_b<Value>() , heun_c<Value>() , algebra )
    { }
};
```

That's it. Now, we have a new stepper method and we can use it, for example with the Lorenz system:

```
typedef boost::array< double , 3 > state_type;
heun< state_type > h;
state_type x = {{ 10.0 , 10.0 , 10.0 }};

integrate_const( h , lorenz() , x , 0.0 , 100.0 , 0.01 ,
                 streaming_observer( std::cout ) );
```

Generation functions

In the Tutorial we have learned how we can use the generation functions `make_controlled` and `make_dense_output` to create controlled and dense output stepper from a simple stepper or an error stepper. The syntax of these two functions is very simple:

```
auto stepper1 = make_controlled( 1.0e-6 , 1.0e-6 , stepper_type() );
auto stepper2 = make_dense_output( 1.0e-6 , 1.0e-6 , stepper_type() );
```

The first two parameters are the absolute and the relative error tolerances and the third parameter is the stepper. In C++03 you can infer the type from the `result_of` mechanism:

```
boost::numeric::odeint::result_of::make_controlled< stepper_type >::type stepper3 = make_con↵
trolled( 1.0e-6 , 1.0e-6 , stepper_type() );
boost::numeric::odeint::result_of::make_dense_output< stepper_type >::type step↵
per4 = make_dense_output( 1.0e-6 , 1.0e-6 , stepper_type() );
```

To use your own steppers with the `make_controlled` or `make_dense_output` you need to specialize two class templates. Suppose your steppers are called `custom_stepper`, `custom_controller` and `custom_dense_output`. Then, the first class you need to specialize is `boost::numeric::get_controller`, a meta function returning the type of the controller:

```
namespace boost { namespace numeric { namespace odeint {

template<>
struct get_controller< custom_stepper >
{
    typedef custom_controller type;
};

} } }
```

The second one is a factory class `boost::numeric::odeint::controller_factory` which constructs the controller from the tolerances and the stepper. In our dummy implementation this class is

```
namespace boost { namespace numeric { namespace odeint {

template<>
struct controller_factory< custom_stepper , custom_controller >
{
    custom_controller operator()( double abs_tol , double rel_tol , const custom_stepper & ) const
    {
        return custom_controller();
    }
};

} } }
```

This is all to use the `make_controlled` mechanism. Now you can use your controller via

```
auto stepper5 = make_controlled( 1.0e-6 , 1.0e-6 , custom_stepper() );
```

For the dense_output_stepper everything works similar. Here you have to specialize `boost::numeric::odeint::get_dense_output` and `boost::numeric::odeint::dense_output_factory`. These two classes have the same syntax as their relatives `get_controller` and `controller_factory`.

All controllers and dense-output steppers in odeint can be used with these mechanisms. In the table below you will find, which steppers is constructed from `make_controlled` or `make_dense_output` if applied on a stepper from odeint:

Table 7. Generation functions make_controlled(abs_error , rel_error , stepper)

Stepper	Result of make_controlled	Remarks
`runge_kutta_cash_karp54`	`controlled_runge_kutta< runge_kutta_cash_karp54 , default_error_checker<...> >`	$a_x=1$, $a_{dxdt}=1$
`runge_kutta_fehlberg78`	`controlled_runge_kutta< runge_kutta_fehlberg78 , default_error_checker<...> >`	$a_x=1$, $a_{dxdt}=1$
`runge_kutta_dopri5`	`controlled_runge_kutta< runge_kutta_dopri5 , default_error_checker<...> >`	$a_x=1$, $a_{dxdt}=1$
`rosenbrock4`	`rosenbrock4_controlled< rosenbrock4 >`	-

Table 8. Generation functions make_dense_output(abs_error , rel_error , stepper)

Stepper	Result of make_dense_output	Remarks
`runge_kutta_dopri5`	`dense_output_runge_kutta< controlled_runge_kutta< runge_kutta_dopri5 , default_error_checker<...> > >`	$a_x=1$, $a_{dxdt}=1$
`rosenbrock4`	`rosenbrock4_dense_output< rosenbrock4_controller< rosenbrock4 > >`	-

Integrate functions

Integrate functions perform the time evolution of a given ODE from some starting time t_0 to a given end time t_1 and starting at state x_0 by subsequent calls of a given stepper's `do_step` function. Additionally, the user can provide an __observer to analyze the state during time evolution. There are five different integrate functions which have different strategies on when to call the observer function during integration. All of the integrate functions except `integrate_n_steps` can be called with any stepper following one of the stepper concepts: Stepper , Error Stepper , Controlled Stepper , Dense Output Stepper. Depending on the abilities of the stepper, the integrate functions make use of step-size control or dense output.

Equidistant observer calls

If observer calls at equidistant time intervals *dt* are needed, the `integrate_const` or `integrate_n_steps` function should be used. We start with explaining `integrate_const`:

```
integrate_const( stepper , system , x0 , t0 , t1 , dt )

integrate_const( stepper , system , x0 , t0 , t1 , dt , observer )
```

These integrate the ODE given by `system` with subsequent steps from `stepper`. Integration start at `t0` and `x0` and ends at some $t' = t_0 + n\ dt$ with n such that $t_1 - dt < t' <= t_1$. `x0` is changed to the approximative solution $x(t')$ at the end of integration. If provided, the `observer` is invoked at times t_0, $t_0 + dt$, $t_0 + 2dt$, ... ,t'. `integrate_const` returns the number of steps performed during the integration. Note that if you are using a simple Stepper or Error Stepper and want to make exactly `n` steps you should prefer the `integrate_n_steps` function below.

- If `stepper` is a Stepper or Error Stepper then `dt` is also the step size used for integration and the observer is called just after every step.
- If `stepper` is a Controlled Stepper then `dt` is the initial step size. The actual step size will change due to error control during time evolution. However, if an observer is provided the step size will be adjusted such that the algorithm always calculates *x(t)* at $t = t_0 + n\,dt$ and calls the observer at that point. Note that the use of Controlled Stepper is reasonable here only if `dt` is considerably larger than typical step sizes used by the stepper.
- If `stepper` is a Dense Output Stepper then `dt` is the initial step size. The actual step size will be adjusted during integration due to error control. If an observer is provided dense output is used to calculate *x(t)* at $t = t_0 + n\,dt$.

Integrate a given number of steps

This function is very similar to `integrate_const` above. The only difference is that it does not take the end time as parameter, but rather the number of steps. The integration is then performed until the time `t0+n*dt`.

```
integrate_n_steps( stepper , system , x0 , t0 , dt , n )

integrate_n_steps( stepper , system , x0 , t0 , dt , n , observer )
```

Integrates the ODE given by `system` with subsequent steps from `stepper` starting at x_0 and t_0. If provided, `observer` is called after every step and at the beginning with `t0`, similar as above. The approximate result for $x(t_0 + n\,dt)$ is stored in `x0`. This function returns the end time `t0 + n*dt`.

Observer calls at each step

If the observer should be called at each time step then the `integrate_adaptive` function should be used. Note that in the case of Controlled Stepper or Dense Output Stepper this leads to non-equidistant observer calls as the step size changes.

```
integrate_adaptive( stepper , system , x0 , t0 , t1 , dt )

integrate_adaptive( stepper , system , x0 , t0 , t1 , dt , observer )
```

Integrates the ODE given by `system` with subsequent steps from `stepper`. Integration start at `t0` and `x0` and ends at t_1. `x0` is changed to the approximative solution $x(t_1)$ at the end of integration. If provided, the `observer` is called after each step (and before the first step at `t0`). `integrate_adaptive` returns the number of steps performed during the integration.

- If `stepper` is a Stepper or Error Stepper then `dt` is the step size used for integration and `integrate_adaptive` behaves like `integrate_const` except that for the last step the step size is reduced to ensure we end exactly at `t1`. If provided, the observer is called at each step.
- If `stepper` is a Controlled Stepper then `dt` is the initial step size. The actual step size is changed according to error control of the stepper. For the last step, the step size will be reduced to ensure we end exactly at `t1`. If provided, the observer is called after each time step (and before the first step at `t0`).
- If stepper is a Dense Output Stepper then `dt` is the initial step size and `integrate_adaptive` behaves just like for Controlled Stepper above. No dense output is used.

Observer calls at given time points

If the observer should be called at some user given time points the `integrate_times` function should be used. The times for observer calls are provided as a sequence of time values. The sequence is either defined via two iterators pointing to begin and end of the sequence or in terms of a Boost.Range object.

```
integrate_times( stepper , system , x0 , times_start , times_end , dt , observer )

integrate_times( stepper , system , x0 , time_range , dt , observer )
```

Integrates the ODE given by `system` with subsequent steps from `stepper`. Integration starts at `*times_start` and ends exactly at `*(times_end-1)`. `x0` contains the approximate solution at the end point of integration. This function requires an observer which

is invoked at the subsequent times `*times_start++` until `times_start == times_end`. If called with a Boost.Range `time_range` the function behaves the same with `times_start = boost::begin( time_range )` and `times_end = boost::end( time_range )`. `integrate_times` returns the number of steps performed during the integration.

- If `stepper` is a Stepper or Error Stepper `dt` is the step size used for integration. However, whenever a time point from the sequence is approached the step size `dt` will be reduced to obtain the state *x(t)* exactly at the time point.
- If `stepper` is a Controlled Stepper then `dt` is the initial step size. The actual step size is adjusted during integration according to error control. However, if a time point from the sequence is approached the step size is reduced to obtain the state *x(t)* exactly at the time point.
- If `stepper` is a Dense Output Stepper then `dt` is the initial step size. The actual step size is adjusted during integration according to error control. Dense output is used to obtain the states *x(t)* at the time points from the sequence.

Convenience integrate function

Additionally to the sophisticated integrate function above odeint also provides a simple `integrate` routine which uses a dense output stepper based on `runge_kutta_dopri5` with standard error bounds 10^{-6} for the steps.

```
integrate( system , x0 , t0 , t1 , dt )

integrate( system , x0 , t0 , t1 , dt , observer )
```

This function behaves exactly like `integrate_adaptive` above but no stepper has to be provided. It also returns the number of steps performed during the integration.

State types, algebras and operations

In odeint the stepper algorithms are implemented independently of the underlying fundamental mathematical operations. This is realized by giving the user full control over the state type and the mathematical operations for this state type. Technically, this is done by introducing three concepts: StateType, Algebra, Operations. Most of the steppers in odeint expect three class types fulfilling these concepts as template parameters. Note that these concepts are not fully independent of each other but rather a valid combination must be provided in order to make the steppers work. In the following we will give some examples on reasonable state_type-algebra-operations combinations. For the most common state types, like `vector<double>` or `array<double,N>` the default values range_algebra and default_operations are perfectly fine and odeint can be used as is without worrying about algebra/operations at all.

Important

state_type, algebra and operations are not independent, a valid combination must be provided to make odeint work properly

Moreover, as odeint handles the memory required for intermediate temporary objects itself, it also needs knowledge about how to create state_type objects and maybe how to allocate memory (resizing). All in all, the following things have to be taken care of when odeint is used with non-standard state types:

- construction/destruction
- resizing (if possible/required)
- algebraic operations

Again, odeint already provides basic interfaces for most of the usual state types. So if you use a `std::vector`, or a `boost::array` as state type no additional work is required, they just work out of the box.

Construction/Resizing

We distinguish between two basic state types: fixed sized and dynamically sized. For fixed size state types the default constructor `state_type()` already allocates the required memory, prominent example is `boost::array<T,N>`. Dynamically sized types have to be resized to make sure enough memory is allocated, the standard constructor does not take care of the resizing. Examples for this are the STL containers like `vector<double>`.

The most easy way of getting your own state type to work with odeint is to use a fixed size state, base calculations on the range_algebra and provide the following functionality:

Name	Expression	Type	Semantics
Construct State	`State x()`	`void`	Creates an instance of `State` and allocates memory.
Begin of the sequence	boost::begin(x)	Iterator	Returns an iterator pointing to the begin of the sequence
End of the sequence	boost::end(x)	Iterator	Returns an iterator pointing to the end of the sequence

Warning

If your state type does not allocate memory by default construction, you **must define it as resizeable** and provide resize functionality (see below). Otherwise segmentation faults will occur.

So fixed sized arrays supported by Boost.Range immediately work with odeint. For dynamically sized arrays one has to additionally supply the resize functionality. First, the state has to be tagged as resizeable by specializing the struct `is_resizeable` which consists of one typedef and one bool value:

Name	Expression	Type	Semantics
Resizability	`is_resizeable<State>::type`	`boost::true_type` or `boost::false_type`	Determines resizeability of the state type, returns `boost::true_type` if the state is resizeable.
Resizability	`is_resizeable<State>::value`	`bool`	Same as above, but with `bool` value.

Defining `type` to be `true_type` and `value` as `true` tells odeint that your state is resizeable. By default, odeint now expects the support of `boost::size(x)` and a `x.resize( boost::size(y) )` member function for resizing:

Name	Expression	Type	Semantics
Get size	`boost::size( x )`	`size_type`	Returns the current size of x.
Resize	`x.resize( boost::size( y ) )`	`void`	Resizes x to have the same size as y.

Using the container interface

As a first example we take the most simple case and implement our own vector `my_vector` which will provide a container interface. This makes Boost.Range working out-of-box. We add a little functionality to our vector which makes it allocate some default capacity by construction. This is helpful when using resizing as then a resize can be assured to not require a new allocation.

```
template< int MAX_N >
class my_vector
{
    typedef std::vector< double > vector;

public:
    typedef vector::iterator iterator;
    typedef vector::const_iterator const_iterator;

public:
    my_vector( const size_t N )
        : m_v( N )
    {
        m_v.reserve( MAX_N );
    }

    my_vector()
        : m_v()
    {
        m_v.reserve( MAX_N );
    }

// ... [ implement container interface ]
```

The only thing that has to be done other than defining is thus declaring my_vector as resizeable:

```
// define my_vector as resizeable

namespace boost { namespace numeric { namespace odeint {

template<size_t N>
struct is_resizeable< my_vector<N> >
{
    typedef boost::true_type type;
    static const bool value = type::value;
};

} } }
```

If we wouldn't specialize the `is_resizeable` template, the code would still compile but odeint would not adjust the size of temporary internal instances of my_vector and hence try to fill zero-sized vectors resulting in segmentation faults! The full example can be found in my_vector.cpp

std::list

If your state type does work with Boost.Range, but handles resizing differently you are required to specialize two implementations used by odeint to check a state's size and to resize:

Name	Expression	Type	Semantics
Check size	`same_size_impl<State,State>::same_size(x, y)`	`bool`	Returns true if the size of x equals the size of y.
Resize	`resize_impl<State,State>::resize(x , y)`	`void`	Resizes x to have the same size as y.

As an example we will use a `std::list` as state type in odeint. Because `std::list` is not supported by `boost::size` we have to replace the same_size and resize implementation to get list to work with odeint. The following code shows the required template specializations:

```
typedef std::list< double > state_type;

namespace boost { namespace numeric { namespace odeint {

template< >
struct is_resizeable< state_type >
{ // declare resizeability
    typedef boost::true_type type;
    const static bool value = type::value;
};

template< >
struct same_size_impl< state_type , state_type >
{ // define how to check size
    static bool same_size( const state_type &v1 ,
                           const state_type &v2 )
    {
        return v1.size() == v2.size();
    }
};

template< >
struct resize_impl< state_type , state_type >
{ // define how to resize
    static void resize( state_type &v1 ,
                        const state_type &v2 )
    {
        v1.resize( v2.size() );
    }
};

} } }
```

With these definitions odeint knows how to resize `std::lists` and so they can be used as state types. A complete example can be found in list_lattice.cpp.

Algebras and Operations

To provide maximum flexibility odeint is implemented in a highly modularized way. This means it is possible to change the underlying mathematical operations without touching the integration algorithms. The fundamental mathematical operations are those of a vector space, that is addition of `state_types` and multiplication of `state_types` with a scalar (`time_type`). In odeint this is realized in two concepts: Algebra and Operations. The standard way how this works is by the range algebra which provides functions that apply a specific operation to each of the individual elements of a container based on the Boost.Range library. If your state type is not supported by Boost.Range there are several possibilities to tell odeint how to do algebraic operations:

- Implement `boost::begin` and `boost::end` for your state type so it works with Boost.Range.
- Implement vector-vector addition operator + and scalar-vector multiplication operator * and use the non-standard `vector_space_algebra`.
- Implement your own algebra that implements the required functions.

GSL Vector

In the following example we will try to use the `gsl_vector` type from GSL (GNU Scientific Library) as state type in odeint. We will realize this by implementing a wrapper around the gsl_vector that takes care of construction/destruction. Also, Boost.Range is extended such that it works with `gsl_vectors` as well which required also the implementation of a new `gsl_iterator`.

Note

odeint already includes all the code presented here, see gsl_wrapper.hpp, so `gsl_vectors` can be used straight out-of-box. The following description is just for educational purpose.

The GSL is a C library, so `gsl_vector` has neither constructor, nor destructor or any `begin` or `end` function, no iterators at all. So to make it work with odeint plenty of things have to be implemented. Note that all of the work shown here is already included in odeint, so using `gsl_vectors` in odeint doesn't require any further adjustments. We present it here just as an educational example. We start with defining appropriate constructors and destructors. This is done by specializing the `state_wrapper` for `gsl_vector`. State wrappers are used by the steppers internally to create and manage temporary instances of state types:

```
template<>
struct state_wrapper< gsl_vector* >
{
    typedef double value_type;
    typedef gsl_vector* state_type;
    typedef state_wrapper< gsl_vector* > state_wrapper_type;

    state_type m_v;

    state_wrapper( )
    {
        m_v = gsl_vector_alloc( 1 );
    }

    state_wrapper( const state_wrapper_type &x )
    {
        resize( m_v , x.m_v );
        gsl_vector_memcpy( m_v , x.m_v );
    }


    ~state_wrapper()
    {
        gsl_vector_free( m_v );
    }

};
```

This `state_wrapper` specialization tells odeint how gsl_vectors are created, copied and destroyed. Next we need resizing, this is required because gsl_vectors are dynamically sized objects:

```
template<>
struct is_resizeable< gsl_vector* >
{
    typedef boost::true_type type;
    const static bool value = type::value;
};

template <>
struct same_size_impl< gsl_vector* , gsl_vector* >
{
    static bool same_size( const gsl_vector* x , const gsl_vector* y )
    {
        return x->size == y->size;
    }
};

template <>
struct resize_impl< gsl_vector* , gsl_vector* >
{
    static void resize( gsl_vector* x , const gsl_vector* y )
    {
        gsl_vector_free( x );
        x = gsl_vector_alloc( y->size );
    }
};
```

Up to now, we defined creation/destruction and resizing, but gsl_vectors also don't support iterators, so we first implement a gsl iterator:

```
/*
 * defines an iterator for gsl_vector
 */
class gsl_vector_iterator
      : public boost::iterator_facade< gsl_vector_iterator , double ,
                                       boost::random_access_traversal_tag >
{
public :

    gsl_vector_iterator( void ): m_p(0) , m_stride( 0 ) { }
    explicit gsl_vector_iterator( gsl_vector *p ) : m_p( p->data ) , m_stride( p->stride ) { }
    friend gsl_vector_iterator end_iterator( gsl_vector * );

private :

    friend class boost::iterator_core_access;
    friend class const_gsl_vector_iterator;

    void increment( void ) { m_p += m_stride; }
    void decrement( void ) { m_p -= m_stride; }
    void advance( ptrdiff_t n ) { m_p += n*m_stride; }
    bool equal( const gsl_vector_iterator &other ) const { return this->m_p == other.m_p; }
    bool equal( const const_gsl_vector_iterator &other ) const;
    double& dereference( void ) const { return *m_p; }

    double *m_p;
    size_t m_stride;
};
```

A similar class exists for the `const` version of the iterator. Then we have a function returning the end iterator (similarly for `const` again):

```
gsl_vector_iterator end_iterator( gsl_vector *x )
{
    gsl_vector_iterator iter( x );
    iter.m_p += iter.m_stride * x->size;
    return iter;
}
```

Finally, the bindings for Boost.Range are added:

```
// template<>
inline gsl_vector_iterator range_begin( gsl_vector *x )
{
    return gsl_vector_iterator( x );
}

// template<>
inline gsl_vector_iterator range_end( gsl_vector *x )
{
    return end_iterator( x );
}
```

Again with similar definitions for the `const` versions. This eventually makes odeint work with gsl vectors as state types. The full code for these bindings is found in gsl_wrapper.hpp. It might look rather complicated but keep in mind that gsl is a pre-compiled C library.

Vector Space Algebra

As seen above, the standard way of performing algebraic operations on container-like state types in odeint is to iterate through the elements of the container and perform the operations element-wise on the underlying value type. This is realized by means of the `range_algebra` that uses Boost.Range for obtaining iterators of the state types. However, there are other ways to implement the algebraic operations on containers, one of which is defining the addition/multiplication operators for the containers directly and then using the `vector_space_algebra`. If you use this algebra, the following operators have to be defined for the state_type:

Name	Expression	Type	Semantics
Addition	`x + y`	`state_type`	Calculates the vector sum 'x+y'.
Assign addition	`x += y`	`state_type`	Performs x+y in place.
Scalar multiplication	`a * x`	`state_type`	Performs multiplication of vector x with scalar a.
Assign scalar multiplication	`x *= a`	`state_type`	Performs in-place multiplication of vector x with scalar a.

Defining these operators makes your state type work with any basic Runge-Kutta stepper. However, if you want to use step-size control, some more functionality is required. Specifically, operations like $max_i(|err_i| / (alpha * |s_i|))$ have to be performed. *err* and *s* are state_types, alpha is a scalar. As you can see, we need element wise absolute value and division as well as an reduce operation to get the maximum value. So for controlled steppers the following things have to be implemented:

Name	Expression	Type	Semantics
Division	`x / y`	`state_type`	Calculates the element-wise division 'x/y'
Absolute value	`abs( x )`	`state_type`	Element wise absolute value

Reduce	`vector_space_reduce_impl< state_type >::reduce( state , operation , init )`	`value_type`	Performs the `operation` for subsequently each element of `state` and returns the aggregate value. E.g.
			`init = operator( init , state[0] );`
			`init = operator( init , state[1] )`
			`...`

Boost.Ublas

As an example for the employment of the `vector_space_algebra` we will adopt `ublas::vector` from Boost.uBLAS to work as a state type in odeint. This is particularly easy because `ublas::vector` supports vector-vector addition and scalar-vector multiplication described above as well as `boost::size`. It also has a resize member function so all that has to be done in this case is to declare resizability:

```
typedef boost::numeric::ublas::vector< double > state_type;

namespace boost { namespace numeric { namespace odeint {

template<>
struct is_resizeable< state_type >
{
    typedef boost::true_type type;
    const static bool value = type::value;
};

} } }
```

Now ublas::vector can be used as state type for simple Runge-Kutta steppers in odeint by specifying the `vector_space_algebra` as algebra in the template parameter list of the stepper. The following code shows the corresponding definitions:

```
int main()
{
    state_type x(3);
    x[0] = 10.0; x[1] = 5.0 ; x[2] = 0.0;
   typedef runge_kutta4< state_type , double , state_type , double , vector_space_algebra > step↵
per;
    integrate_const( stepper() , lorenz , x ,
                     0.0 , 10.0 , 0.1 );
}
```

Note again, that we haven't supported the requirements for controlled steppers, but only for simple Runge-Kutta methods. You can find the full example in lorenz_ublas.cpp.

Point type

Here we show how to implement the required operators on a state type. As example we define a new class `point3D` representing a three-dimensional vector with components x,y,z and define addition and scalar multiplication operators for it. We use Boost.Operators to reduce the amount of code to be written. The class for the point type looks as follows:

```
class point3D :
    boost::additive1< point3D ,
    boost::additive2< point3D , double ,
    boost::multiplicative2< point3D , double > > >
{
public:

    double x , y , z;

    point3D()
        : x( 0.0 ) , y( 0.0 ) , z( 0.0 )
    { }

    point3D( const double val )
        : x( val ) , y( val ) , z( val )
    { }

    point3D( const double _x , const double _y , const double _z  )
        : x( _x ) , y( _y ) , z( _z )
    { }

    point3D& operator+=( const point3D &p )
    {
        x += p.x; y += p.y; z += p.z;
        return *this;
    }

    point3D& operator*=( const double a )
    {
        x *= a; y *= a; z *= a;
        return *this;
    }

};
```

By deriving from Boost.Operators classes we don't have to define outer class operators like `operator+( point3D , point3D )` because that is taken care of by the operators library. Note that for simple Runge-Kutta schemes (like `runge_kutta4`) only the `+` and `*` operators are required. If, however, a controlled stepper is used one also needs to specify the division operator `/` because calculation of the error term involves an element wise division of the state types. Additionally, controlled steppers require an `abs` function calculating the element-wise absolute value for the state type:

```
// only required for steppers with error control
point3D operator/( const point3D &p1 , const point3D &p2 )
{
    return point3D( p1.x/p2.x , p1.y/p2.y , p1.z/p1.z );
}

point3D abs( const point3D &p )
{
    return point3D( std::abs(p.x) , std::abs(p.y) , std::abs(p.z) );
}
```

Finally, we have to add a specialization for `reduce` implementing a reduction over the state type:

```
namespace boost { namespace numeric { namespace odeint {
// specialization of vector_space_reduce, only required for steppers with error control
template<>
struct vector_space_reduce< point3D >
{
    template< class Value , class Op >
    Value operator() ( const point3D &p , Op op , Value init )
    {
        init = op( init , p.x );
        //std::cout << init << " ";
        init = op( init , p.y );
        //std::cout << init << " ";
        init = op( init , p.z );
        //std::cout << init << std::endl;
        return init;
    }
};
} } }
```

Again, note that the two last steps were only required if you want to use controlled steppers. For simple steppers definition of the simple += and *= operators are sufficient. Having defined such a point type, we can easily perform the integration on a Lorenz system by using the `vector_space_algebra` again:

```
const double sigma = 10.0;
const double R = 28.0;
const double b = 8.0 / 3.0;

void lorenz( const point3D &x , point3D &dxdt , const double t )
{
    dxdt.x = sigma * ( x.y - x.x );
    dxdt.y = R * x.x - x.y - x.x * x.z;
    dxdt.z = -b * x.z + x.x * x.y;
}

using namespace boost::numeric::odeint;

int main()
{

    point3D x( 10.0 , 5.0 , 5.0 );
    // point type defines it's own operators -> use vector_space_algebra !
    typedef runge_kutta_dopri5< point3D , double , point3D ,
                                double , vector_space_algebra > stepper;
    int steps = integrate_adaptive( make_controlled<stepper>( 1E-10 , 1E-10 ) , lorenz , x ,
                                    0.0 , 10.0 , 0.1 );
    std::cout << x << std::endl;
    std::cout << "steps: " << steps << std::endl;
}
```

The whole example can be found in lorenz_point.cpp

gsl_vector, gsl_matrix, ublas::matrix, blitz::matrix, thrust

Adapt your own operations

to be continued

- thrust
- gsl_complex

- min, max, pow

Using boost::ref

In odeint all system functions and observers are passed by value. For example, if you call a `do_step` method of a particular stepper or the integration functions, your system and your stepper will be passed by value:

```
rk4.do_step( sys , x , t , dt );   // pass sys by value
```

This behavior is suitable for most systems, especially if your system does not contain any data or only a few parameters. However, in some cases you might contain some large amount of data with you system function and passing them by value is not desired since the data would be copied.

In such cases you can easily use `boost::ref` (and its relative `boost::cref`) which passes its argument by reference (or constant reference). odeint will unpack the arguments and no copying at all of your system object will take place:

```
rk4.do_step( boost::ref( sys ) , x , t , dt );   // pass sys as references
```

The same mechanism can be used for the observers in the integrate functions.

Tip

If you are using C++11 you can also use `std::ref` and `std::cref`

Using boost::range

Most steppers in odeint also accept the state give as a range. A range is sequence of values modeled by a range concept. See Boost.Range for an overview over existing concepts and examples of ranges. This means that the `state_type` of the stepper need not necessarily be used to call the `do_step` method.

One use-case for Boost.Range in odeint has been shown in Chaotic System where the state consists of two parts: one for the original system and one for the perturbations. The ranges are used to initialize (solve) only the system part where the perturbation part is not touched, that is a range consisting only of the system part is used. After that the complete state including the perturbations is solved.

Another use case is a system consisting of coupled units where you want to initialize each unit separately with the ODE of the uncoupled unit. An example is a chain of coupled van-der-Pol-oscillators which are initialized uniformly from the uncoupled van-der-Pol-oscillator. Then you can use Boost.Range to solve only one individual oscillator in the chain.

In short, you can Boost.Range to use one state within two system functions which expect states with different sizes.

An example was given in the Chaotic System tutorial. Using Boost.Range usually means that your system function needs to adapt to the iterators of Boost.Range. That is, your function is called with a range and you need to get the iterators from that range. This can easily be done. You have to implement your system as a class or a struct and you have to templatize the `operator()`. Then you can use the `range_iterator`-meta function and `boost::begin` and `boost::end` to obtain the iterators of your range:

```
class sys
{
    template< class State , class Deriv >
    void operator()( const State &x_ , Deriv &dxdt_ , double t ) const
    {
         typename boost::range_iterator< const State >::type x = boost::begin( x_ );
         typename boost::range_iterator< Deriv >::type dxdt = boost::begin( dxdt_ );

         // fill dxdt
    }
};
```

If your range is a random access-range you can also apply the bracket operator to the iterator to access the elements in the range:

```
class sys
{
    template< class State , class Deriv >
    void operator()( const State &x_ , Deriv &dxdt_ , double t ) const
    {
         typename boost::range_iterator< const State >::type x = boost::begin( x_ );
         typename boost::range_iterator< Deriv >::type dxdt = boost::begin( dxdt_ );

         dxdt[0] = f1( x[0] , x[1] );
         dxdt[1] = f2( x[0] , x[1] );
    }
};
```

The following two tables show which steppers and which algebras are compatible with Boost.Range.

Table 9. Steppers supporting Boost.Range

Stepper
adams_bashforth_moulton
bulirsch_stoer_dense_out
bulirsch_stoer
controlled_runge_kutta
dense_output_runge_kutta
euler
explicit_error_generic_rk
explicit_generic_rk
rosenbrock4_controller
rosenbrock4_dense_output
rosenbrock4
runge_kutta4_classic
runge_kutta4
runge_kutta_cash_karp54_classic
runge_kutta_cash_karp54
runge_kutta_dopri5
runge_kutta_fehlberg78
symplectic_euler
symplectic_rkn_sb3a_mclachlan

Table 10. Algebras supporting Boost.Range

algebra
range_algebra
thrust_algebra

Binding member functions

Binding member functions to a function objects suitable for odeint system function is not easy, at least in C++03. The usual way of using __boost_bind does not work because of the forwarding problem. odeint provides two `do_step` method which only differ in the const specifiers of the arguments and __boost_bind binders only provide the specializations up to two argument which is not enough for odeint.

But one can easily implement the according binders themself:

```
template< class Obj , class Mem >
class ode_wrapper
{
    Obj m_obj;
    Mem m_mem;

public:

    ode_wrapper( Obj obj , Mem mem ) : m_obj( obj ) , m_mem( mem ) { }

    template< class State , class Deriv , class Time >
    void operator()( const State &x , Deriv &dxdt , Time t )
    {
        (m_obj.*m_mem)( x , dxdt , t );
    }
};

template< class Obj , class Mem >
ode_wrapper< Obj , Mem > make_ode_wrapper( Obj obj , Mem mem )
{
    return ode_wrapper< Obj , Mem >( obj , mem );
}
```

One can use this binder as follows

```
struct lorenz
{
    void ode( const state_type &x , state_type &dxdt , double t ) const
    {
        dxdt[0] = 10.0 * ( x[1] - x[0] );
        dxdt[1] = 28.0 * x[0] - x[1] - x[0] * x[2];
        dxdt[2] = -8.0 / 3.0 * x[2] + x[0] * x[1];
    }
};

int main( int argc , char *argv[] )
{
    using namespace boost::numeric::odeint;
    state_type x = {{ 10.0 , 10.0 , 10.0 }};
   integrate_const( runge_kutta4< state_type >() , make_ode_wrapper( lorenz() , &lorenz::ode ) ,
                    x , 0.0 , 10.0 , 0.01 );
    return 0;
}
```

Binding member functions in C++11

In C++11 one can use `std::bind` and one does not need to implement the bind themself:

```
namespace pl = std::placeholders;

state_type x = {{ 10.0 , 10.0 , 10.0 }};
integrate_const( runge_kutta4< state_type >() ,
                 std::bind( &lorenz::ode , lorenz() , pl::_1 , pl::_2 , pl::_3 ) ,
                 x , 0.0 , 10.0 , 0.01  );
```

Concepts

System

Description

The System concept models the algorithmic implementation of the rhs. of the ODE $x' = f(x,t)$. The only requirement for this concept is that it should be callable with a specific parameter syntax (see below). A System is typically implemented as a function or a functor. Systems fulfilling this concept are required by all Runge-Kutta steppers as well as the Bulirsch-Stoer steppers. However, symplectic and implicit steppers work with other system concepts, see Symplectic System and Implicit System.

Notation

`System`	A type that is a model of System
`State`	A type representing the state x of the ODE
`Deriv`	A type representing the derivative x' of the ODE
`Time`	A type representing the time
`sys`	An object of type `System`
`x`	Object of type `State`
`dxdt`	Object of type `Deriv`
`t`	Object of type `Time`

Valid expressions

Name	Expression	Type	Semantics
Calculate $dx/dt := f(x,t)$	`sys( x , dxdt , t )`	`void`	Calculates f(x,t), the result is stored into dxdt

Symplectic System

Description

This concept describes how to define a symplectic system written with generalized coordinate `q` and generalized momentum `p`:

$q'(t) = f(p)$

$p'(t) = g(q)$

Such a situation is typically found for Hamiltonian systems with a separable Hamiltonian:

$H(p,q) = H_{kin}(p) + V(q)$

which gives the equations of motion:

$q'(t) = dH_{kin} / dp = f(p)$

$p'(t) = dV / dq = g(q)$

The algorithmic implementation of this situation is described by a pair of callable objects for *f* and *g* with a specific parameter signature. Such a system should be implemented as a std::pair of functions or a functors. Symplectic systems are used in symplectic steppers like `symplectic_rkn_sb3a_mclachlan`.

Notation

`System`	A type that is a model of SymplecticSystem
`Coor`	The type of the coordinate *q*
`Momentum`	The type of the momentum *p*
`CoorDeriv`	The type of the derivative of coordinate *q'*
`MomentumDeriv`	The type of the derivative of momentum *p'*
`sys`	An object of the type `System`
`q`	Object of type Coor
`p`	Object of type Momentum
`dqdt`	Object of type CoorDeriv
`dpdt`	Object of type MomentumDeriv

Valid expressions

Name	Expression	Type	Semantics
Check for pair	`boost::is_pair< System >::type`	`boost::mpl::true_`	Check if System is a pair
Calculate *dq/dt = f(p)*	`sys.first( p , dqdt )`	`void`	Calculates *f(p)*, the result is stored into `dqdt`
Calculate *dp/dt = g(q)*	`sys.second( q , dpdt )`	`void`	Calculates *g(q)*, the result is stored into `dpdt`

Simple Symplectic System

Description

In most Hamiltonian systems the kinetic term is a quadratic term in the momentum $H_{kin} = p\hat{}2 / 2m$ and in many cases it is possible to rescale coordinates and set *m=1* which leads to a trivial equation of motion:

q'(t) = f(p) = p.

while for *p'* we still have the general form

p'(t) = g(q)

As this case is very frequent we introduced a concept where only the nontrivial equation for *p'* has to be provided to the symplectic stepper. We call this concept *SimpleSymplecticSystem*

Notation

System	A type that is a model of SimpleSymplecticSystem

Coor	The type of the coordinate *q*
MomentumDeriv	The type of the derivative of momentum *p'*
sys	An object that models System
q	Object of type Coor
dpdt	Object of type MomentumDeriv

Valid Expressions

Name	Expression	Type	Semantics
Check for pair	`boost::is_pair< System >::type`	`boost::mpl::false_`	Check if System is a pair, should be evaluated to false in this case.
Calculate *dp/dt = g(q)*	`sys( q , dpdt )`	`void`	Calculates *g(q)*, the result is stored into `dpdt`

Implicit System

Description

This concept describes how to define a ODE that can be solved by an implicit routine. Implicit routines need not only the function *f(x,t)* but also the Jacobian *df/dx = A(x,t)*. *A* is a matrix and implicit routines need to solve the linear problem *Ax = b*. In odeint this is implemented with use of Boost.uBLAS, therefore, the *state_type* implicit routines is *ublas::vector* and the matrix is defined as *ublas::matrix*.

Notation

`System`	A type that is a model of `Implicit System`
`Time`	A type representing the time of the ODE
`sys`	An object of type `System`
`x`	Object of type ublas::vector
`dxdt`	Object of type ublas::vector
`jacobi`	Object of type ublas::matrix
`t`	Object of type `Time`

Valid Expressions

Name	Expression	Type	Semantics
Calculate *dx/dt := f(x,t)*	`sys.first( x , dxdt , t )`	`void`	Calculates `f(x,t)`, the result is stored into dxdt
Calculate *A := df/dx (x,t)*	`sys.second( x , jacobi , t )`	`void`	Calculates the Jacobian of *f* at *x,t*, the result is stored into `jacobi`

Stepper

This concepts specifies the interface a simple stepper has to fulfill to be used within the integrate functions.

Description

The basic stepper concept. A basic stepper following this Stepper concept is able to perform a single step of the solution *x(t)* of an ODE to obtain *x(t+dt)* using a given step size *dt*. Basic steppers can be Runge-Kutta steppers, symplectic steppers as well as implicit steppers. Depending on the actual stepper, the ODE is defined as System, Symplectic System, Simple Symplectic System or Implicit System. Note that all error steppers are also basic steppers.

Refinement of

- DefaultConstructable
- CopyConstructable

Associated types

- **state_type**

  `Stepper::state_type`

  The type characterizing the state of the ODE, hence *x*.

- **deriv_type**

  `Stepper::deriv_type`

  The type characterizing the derivative of the ODE, hence *d x/dt*.

- **time_type**

  `Stepper::time_type`

  The type characterizing the dependent variable of the ODE, hence the time *t*.

- **value_type**

  `Stepper::value_type`

  The numerical data type which is used within the stepper, something like `float`, `double`, `complex< double >`.

- **order_type**

  `Stepper::order_type`

  The type characterizing the order of the ODE, typically `unsigned short`.

- **stepper_category**

  `Stepper::stepper_category`

  A tag type characterizing the category of the stepper. This type must be convertible to `stepper_tag`.

Notation

`Stepper`	A type that is a model of Stepper
`State`	A type representing the state *x* of the ODE

`Time`	A type representing the time *t* of the ODE
`stepper`	An object of type `Stepper`
`x`	Object of type `State`
`t`, `dt`	Objects of type `Time`
`sys`	An object defining the ODE. Depending on the Stepper this might be a model of System, Symplectic System, Simple Symplectic System or Implicit System

Valid Expressions

Name	Expression	Type	Semantics
Get the order	`stepper.order()`	`order_type`	Returns the order of the stepper.
Do step	`stepper.do_step( sys , x , t , dt )`	`void`	Performs one step of step size `dt`. The newly obtained state is written in place in `x`.

Models

- `runge_kutta4`
- `euler`
- `runge_kutta_cash_karp54`
- `runge_kutta_dopri5`
- `runge_kutta_fehlberg78`
- `modified_midpoint`
- `rosenbrock4`

Error Stepper

This concepts specifies the interface an error stepper has to fulfill to be used within a ControlledErrorStepper. An error stepper must always fulfill the stepper concept. This can trivially implemented by

```
template< class System >
error_stepper::do_step( System sys , state_type &x , time_type t , time_type dt )
{
    state_type xerr;
    // allocate xerr
    do_step( sys , x , t , dt , xerr );
}
```

Description

An error stepper following this Error Stepper concept is capable of doing one step of the solution *x(t)* of an ODE with step-size *dt* to obtain *x(t+dt)* and also computing an error estimate x_{err} of the result. Error Steppers can be Runge-Kutta steppers, symplectic steppers as well as implicit steppers. Based on the stepper type, the ODE is defined as System, Symplectic System, Simple Symplectic System or Implicit System.

Refinement of

- DefaultConstructable
- CopyConstructable
- Stepper

Associated types

- **state_type**

  `Stepper::state_type`

  The type characterizing the state of the ODE, hence *x*.

- **deriv_type**

  `Stepper::deriv_type`

  The type characterizing the derivative of the ODE, hence *d x/dt*.

- **time_type**

  `Stepper::time_type`

  The type characterizing the dependent variable of the ODE, hence the time *t*.

- **value_type**

  `Stepper::value_type`

  The numerical data type which is used within the stepper, something like `float`, `double`, `complex< double >`.

- **order_type**

  `Stepper::order_type`

  The type characterizing the order of the ODE, typically `unsigned short`.

- **stepper_category**

  `Stepper::stepper_category`

  A tag type characterizing the category of the stepper. This type must be convertible to `error_stepper_tag`.

Notation

`ErrorStepper`	A type that is a model of Error Stepper
`State`	A type representing the state *x* of the ODE
`Error`	A type representing the error calculated by the stepper, usually same as `State`
`Time`	A type representing the time *t* of the ODE
`stepper`	An object of type `ErrorStepper`
`x`	Object of type `State`
`xerr`	Object of type `Error`

`t`, `dt`	Objects of type `Time`
`sys`	An object defining the ODE, should be a model of either System, Symplectic System, Simple Symplectic System or Implicit System.

Valid Expressions

Name	Expression	Type	Semantics
Get the stepper order	`stepper.order()`	`order_type`	Returns the order of the stepper for one step without error estimation.
Get the stepper order	`stepper.stepper_order()`	`order_type`	Returns the order of the stepper for one error estimation step which is used for error calculation.
Get the error order	`stepper.errorr_order()`	`order_type`	Returns the order of the error step which is used for error calculation.
Do step	`stepper.do_step( sys , x , t , dt )`	`void`	Performs one step of step size `dt`. The newly obtained state is written in-place to `x`.
Do step with error estimation	`stepper.do_step( sys , x , t , dt , xerr )`	`void`	Performs one step of step size `dt` with error estimation. The newly obtained state is written in-place to `x` and the estimated error to `xerr`.

Models

- `runge_kutta_cash_karp54`
- `runge_kutta_dopri5`
- `runge_kutta_fehlberg78`
- `rosenbrock4`

Controlled Stepper

This concept specifies the interface a controlled stepper has to fulfill to be used within integrate functions.

Description

A controlled stepper following this Controlled Stepper concept provides the possibility to perform one step of the solution *x(t)* of an ODE with step-size *dt* to obtain *x(t+dt)* with a given step-size *dt*. Depending on an error estimate of the solution the step might be rejected and a smaller step-size is suggested.

Associated types

- **state_type**

  `Stepper::state_type`

The type characterizing the state of the ODE, hence *x*.

- **deriv_type**

  `Stepper::deriv_type`

  The type characterizing the derivative of the ODE, hence *d x/dt*.

- **time_type**

  `Stepper::time_type`

  The type characterizing the dependent variable of the ODE, hence the time *t*.

- **value_type**

  `Stepper::value_type`

  The numerical data type which is used within the stepper, something like `float`, `double`, `complex< double >`.

- **stepper_category**

  `Stepper::stepper_category`

  A tag type characterizing the category of the stepper. This type must be convertible to `controlled_stepper_tag`.

Notation

`ControlledStepper`	A type that is a model of Controlled Stepper
`State`	A type representing the state *x* of the ODE
`Time`	A type representing the time *t* of the ODE
`stepper`	An object of type `ControlledStepper`
`x`	Object of type `State`
`t`, `dt`	Objects of type `Time`
`sys`	An object defining the ODE, should be a model of System, Symplectic System, Simple Symplectic System or Implicit System.

Valid Expressions

Name	Expression	Type	Semantics
Do step	`stepper.try_step( sys , x , t , dt )`	`controlled_step_result`	Tries one step of step size `dt`. If the step was successful, `success` is returned, the resulting state is written to `x`, the new time is stored in `t` and `dt` now contains a new (possibly larger) step-size for the next step. If the error was too big, `rejected` is returned and the results are neglected - `x` and `t` are unchanged and `dt` now contains a reduced step-size to be used for the next try.

Models

- `controlled_error_stepper< runge_kutta_cash_karp54 >`
- `controlled_error_stepper_fsal< runge_kutta_dopri5 >`
- `controlled_error_stepper< runge_kutta_fehlberg78 >`
- `rosenbrock4_controller`
- `bulirsch_stoer`

Dense Output Stepper

This concept specifies the interface a dense output stepper has to fulfill to be used within integrate functions.

Description

A dense output stepper following this Dense Output Stepper concept provides the possibility to perform a single step of the solution *x(t)* of an ODE to obtain *x(t+dt)*. The step-size `dt` might be adjusted automatically due to error control. Dense output steppers also can interpolate the solution to calculate the state *x(t')* at any point *t <= t' <= t+dt*.

Associated types

- **state_type**

  `Stepper::state_type`

  The type characterizing the state of the ODE, hence *x*.

- **deriv_type**

  `Stepper::deriv_type`

  The type characterizing the derivative of the ODE, hence *d x/dt*.

- **time_type**

  `Stepper::time_type`

  The type characterizing the dependent variable of the ODE, hence the time *t*.

- **value_type**

  `Stepper::value_type`

  The numerical data type which is used within the stepper, something like `float`, `double`, `complex< double >`.

- **stepper_category**

  `Stepper::stepper_category`

  A tag type characterizing the category of the stepper. This type must be convertible to `dense_output_stepper_tag`.

Notation

`Stepper`	A type that is a model of Dense Output Stepper
`State`	A type representing the state *x* of the ODE
`stepper`	An object of type `Stepper`

`x0`, `x`	Object of type `State`
`t0`, `dt0`, `t`	Objects of type `Stepper::time_type`
`sys`	An object defining the ODE, should be a model of System, Symplectic System, Simple Symplectic System or Implicit System.

Valid Expressions

Name	Expression	Type	Semantics
Initialize integration	`stepper.initialize( x0 , t0 , dt0 )`	void	Initializes the stepper with initial values `x0`, `t0` and `dt0`.
Do step	`stepper.do_step( sys )`	`std::pair< Stepper::time_type , Stepper::time_type >`	Performs one step using the ODE defined by `sys`. The step-size might be changed internally due to error control. This function returns a pair containing `t` and `t+dt` representing the interval for which interpolation can be performed.
Do interpolation	`stepper.calc_state( t_inter , x )`	`void`	Performs the interpolation to calculate /x(t_{inter}/) where /t <= t_{inter} <= t+dt/.
Get current time	`stepper.current_time()`	`const Stepper::time_type&`	Returns the current time *t+dt* of the stepper, that is the end time of the last step and the starting time for the next call of `do_step`
Get current state	`stepper.current_state()`	`const Stepper::state_type&`	Returns the current state of the stepper, that is *x(t+dt)*, the state at the time returned by `stepper.current_time()`

Models

- `dense_output_controlled_explicit_fsal< controlled_error_stepper_fsal< runge_kutta_dopri5 >`
- `bulirsch_stoer_dense_out`
- `rosenbrock4_dense_output`

State Algebra Operations

Note

The following does not apply to implicit steppers like implicit_euler or Rosenbrock 4 as there the `state_type` can not be changed from `ublas::vector` and no algebra/operations are used.

Description

The `State`, `Algebra` and `Operations` together define a concept describing how the mathematical vector operations required for the stepper algorithms are performed. The typical vector operation done within steppers is

$y = \Sigma \alpha_i x_i$.

The `State` represents the state variable of an ODE, usually denoted with *x*. Algorithmically, the state is often realized as a `vector< double >` or `array< double , N >`, however, the genericity of odeint enables you to basically use anything as a state type. The algorithmic counterpart of such mathematical expressions is divided into two parts. First, the `Algebra` is used to account for the vector character of the equation. In the case of a `vector` as state type this means the `Algebra` is responsible for iteration over all vector elements. Second, the `Operations` are used to represent the actual operation applied to each of the vector elements. So the `Algebra` iterates over all elements of the `State`s and calls an operation taken from the `Operations` for each element. This is where `State`, `Algebra` and `Operations` have to work together to make odeint running. Please have a look at the `range_algebra` and `default_operations` to see an example how this is implemented.

In the following we describe how `State`, `Algebra` and `Operations` are used together within the stepper implementations.

Operations

Notation

`Operations`	The operations type
`Value1`, ... , `ValueN`	Types representing the value or time type of stepper
`Scale`	Type of the scale operation
`scale`	Object of type `Scale`
`ScaleSum`*N*	Type that represents a general scale_sum operation, *N* should be replaced by a number from 1 to 14.
`scale_sum`*N*	Object of type `ScaleSum`*N*, *N* should be replaced by a number from 1 to 14.
`ScaleSumSwap2`	Type of the scale sum swap operation
`scale_sum_swap2`	Object of type `ScaleSumSwap2`
`a1, a2, ...`	Objects of type `Value1`, `Value2`, ...
`y, x1, x2, ...`	Objects of `State`'s value type

Valid Expressions

Name	Expression	Type	Semantics
Get scale operation	`Operations::scale< Value >`	`Scale`	Get `Scale` from `Operations`
`Scale` constructor	`Scale< Value >( a )`	`Scale`	Constructs a `Scale` object
`Scale` operation	`scale( x )`	`void`	Calculates `x *= a`
Get general `scale_sum` operation	`Operations::scale_sumN< Value1 , ... , ValueN >`	`ScaleSumN`	Get the `ScaleSumN` type from `Operations`, *N* should be replaced by a number from 1 to 14.
`scale_sum` constructor	`ScaleSumN< Value1 , ... , ValueN >( a1 , ... , aN )`	`ScaleSumN`	Constructs a `scale_sum` object given *N* parameter values with *N* between 1 and 14.
`scale_sum` operation	`scale_sumN( y , x1 , ... , xN )`	`void`	Calculates `y = a1*x1 + a2*x2 + ... + aN*xN`. Note that this is an *N*+1-ary function call.
Get scale sum swap operation	`Operations::scale_sum_swap2< Value1 , Value2 >`	`ScaleSumSwap2`	Get scale sum swap from operations
`ScaleSumSwap2` constructor	`ScaleSumSwap2< Value1 , Value2 >( a1 , a2 )`	`ScaleSumSwap2`	Constructor
`ScaleSumSwap2` operation	`scale_sum_swap2( x1 , x2 , x3 )`	`void`	Calculates `tmp = x1`, `x1 = a1*x2 + a2*x3` and `x2 = tmp`.

Algebra

Notation

`State`	The state type
`Algebra`	The algebra type
`OperationN`	An *N*-ary operation type, *N* should be a number from 1 to 14.
`algebra`	Object of type `Algebra`
`operationN`	Object of type `OperationN`
`y, x1, x2, ...`	Objects of type `State`

Valid Expressions

Name	Expression	Type	Semantics
Vector Operation with arity 2	`algebra.for_each2( y , x , operation2 )`	void	Calls `operation2( y_i , x_i )` for each element `y_i` of `y` and `x_i` of `x`.
Vector Operation with arity 3	`algebra.for_each3( y , x1 , x2 , operation3 )`	void	Calls `operation3( y_i , x1_i , x2_i )` for each element `y_i` of `y` and `x1_i` of `x1` and `x2_i` of `x2`.
Vector Operation with arity *N*	`algebra.for_eachN( y , x1 , ... , xN , operationN )`	void	Calls `operationN( y_i , x1_i , ... , xN_i )` for each element `y_i` of `y` and `x1_i` of `x1` and so on. *N* should be replaced by a number between 1 and 14.

Pre-Defined implementations

As standard configuration odeint uses the `range_algebra` and `default_operations` which suffices most situations. However, a few more possibilities exist either to gain better performance or to ensure interoperability with other libraries. In the following we list the existing `Algebra`/`Operations` configurations that can be used in the steppers.

State	Algebra	Operations	Remarks
Anything supporting Boost.Range, like `std::vector`, `std::list`, `boost::array`,... based on a `value_type` that supports operators +,* (typically `double`)	`range_algebra`	`default_operations`	Standard implementation, applicable for most typical situations.
`boost::array` based on a `value_type` that supports operators +,*	`array_algebra`	`default_operations`	Special implementation for boost::array with better performance than `range_algebra`
Anything that defines operators + within itself and * with scalar (Mathematically spoken, anything that is a vector space).	`vector_space_algebra`	`default_operations`	For the use of Controlled Stepper, the template `vector_space_reduce` has to be instantiated.
`thrust::device_vector`, `thrust::host_vector`	`thrust_algebra`	`thrust_operations`	For running odeint on CUDA devices by using Thrust
`boost::array` or anything which allocates the elements in a C-like manner	`vector_space_algebra`	`mkl_operations`	Using the Intel Math Kernel Library in odeint for maximum performance. Currently, only the RK4 stepper is supported.

Example expressions

Name	Expression	Type	Semantics
Vector operation	`algebra.for_each3( y , x1 , x2 , Operations::scale_sum2< Value1 , Value2 >( a1 , a2 ) )`	void	Calculates ***y*** = *a1* ***x1*** + *a2* ***x2***

State Wrapper

Description

The `State Wrapper` concept describes the way odeint creates temporary state objects to store intermediate results within the stepper's `do_step` methods.

Notation

`State`	A type that is the `state_type` of the ODE
`WrappedState`	A type that is a model of State Wrapper for the state type `State`.
`x`	Object of type `State`
`w`	Object of type `WrappedState`

Valid Expressions

Name	Expression	Type	Semantics
Get resizeability	`is_resizeable< State >`	`boost::false_type` or `boost::true_type`	Returns `boost::true_type` if the `State` is resizeable, `boost::false_type` otherwise.
Create `WrappedState` type	`state_wrapper< State >`	`WrappedState`	Creates the type for a `WrappedState` for the state type `State`
Constructor	`WrappedState()`	`WrappedState`	Constructs a state wrapper with an empty state
Copy Constructor	`WrappedState( w )`	`WrappedState`	Constructs a state wrapper with a state of the same size as the state in `w`
Get state	`w.m_v`	`State`	Returns the `State` object of this state wrapper.

Literature

General information about numerical integration of ordinary differential equations:

[1] Press William H et al., Numerical Recipes 3rd Edition: The Art of Scientific Computing, 3rd ed. (Cambridge University Press, 2007).

[2] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems, 2nd ed. (Springer, Berlin, 2009).

[3] Ernst Hairer and Gerhard Wanner, Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems, 2nd ed. (Springer, Berlin, 2010).

Symplectic integration of numerical integration:

[4] Ernst Hairer, Gerhard Wanner, and Christian Lubich, Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations, 2nd ed. (Springer-Verlag Gmbh, 2006).

[5] Leimkuhler Benedict and Reich Sebastian, Simulating Hamiltonian Dynamics (Cambridge University Press, 2005).

Special symplectic methods:

[6] Haruo Yoshida, “Construction of higher order symplectic integrators,” Physics Letters A 150, no. 5 (November 12, 1990): 262-268.

[7] Robert I. McLachlan, “On the numerical integration of ordinary differential equations by symmetric composition methods,” SIAM J. Sci. Comput. 16, no. 1 (1995): 151-168.

Special systems:

[8] Fermi-Pasta-Ulam nonlinear lattice oscillations

[9] Arkady Pikovsky, Michael Rosemblum, and Jürgen Kurths, Synchronization: A Universal Concept in Nonlinear Sciences. (Cambridge University Press, 2001).

Acknowledgments

- Steven Watanabe for managing the Boost review process.
- All people who participated in the odeint review process on the Boost mailing list.
- Paul Bristow for helping with the documentation.
- The Google Summer Of Code (GSOC) program for funding and Andrew Sutton for supervising us during the GSOC and for lots of useful discussions and feedback about many implementation details..
- Joachim Faulhaber for motivating us to participate in the Boost review process and many detailed comments about the library.
- All users of odeint. They are the main motivation for our efforts.

Contributers

- Andreas Angelopoulos implemented the sparse matrix implicit Euler stepper using the MTL4 library.
- Rajeev Singh implemented the stiff Van der Pol oscillator example.
- Sylwester Arabas improved the documentation.
- Denis Demidov provided the adaption to the VexCL and Viennacl libraries.
- Christoph Koke provided improved binders.
- Lee Hodgkinson provided the black hole example.
- Michael Morin fixed several typos in the documentation and the the source code comments.

odeint Reference

Header <boost/numeric/odeint/integrate/integrate.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename System, typename State, typename Time,
               typename Observer>
        size_t integrate(System, State &, Time, Time, Time, Observer);
      template<typename System, typename State, typename Time>
        size_t integrate(System, State &, Time, Time, Time);
    }
  }
}
```

Function template integrate

boost::numeric::odeint::integrate — Integrates the ODE.

Synopsis

```
// In header: <boost/numeric/odeint/integrate/integrate.hpp>


template<typename System, typename State, typename Time, typename Observer>
  size_t integrate(System system, State & start_state, Time start_time,
                   Time end_time, Time dt, Observer observer);
```

Description

Integrates the ODE given by system from start_time to end_time starting with start_state as initial condition and dt as initial time step. This function uses a dense output dopri5 stepper and performs an adaptive integration with step size control, thus dt changes during the integration. This method uses standard error bounds of 1E-6. After each step, the observer is called.

Parameters:	`dt`	Initial step size, will be adjusted during the integration.
	`end_time`	End time of the integration.
	`observer`	Observer that will be called after each time step.
	`start_state`	The initial state.
	`start_time`	Start time of the integration.
	`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation.
Returns:	The number of steps performed.	

Function template integrate

boost::numeric::odeint::integrate — Integrates the ODE without observer calls.

Synopsis

```
// In header: <boost/numeric/odeint/integrate/integrate.hpp>


template<typename System, typename State, typename Time>
  size_t integrate(System system, State & start_state, Time start_time,
                   Time end_time, Time dt);
```

Description

Integrates the ODE given by system from start_time to end_time starting with start_state as initial condition and dt as initial time step. This function uses a dense output dopri5 stepper and performs an adaptive integration with step size control, thus dt changes during the integration. This method uses standard error bounds of 1E-6. No observer is called.

Parameters:	`dt`	Initial step size, will be adjusted during the integration.
	`end_time`	End time of the integration.
	`start_state`	The initial state.
	`start_time`	Start time of the integration.
	`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation.
Returns:	The number of steps performed.	

Header <boost/numeric/odeint/integrate/integrate_adaptive.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Stepper, typename System, typename State,
               typename Time, typename Observer>
        size_t integrate_adaptive(Stepper, System, State &, Time, Time, Time,
                                  Observer);

      // Second version to solve the forwarding problem, can be called with Boost.Range as ↵
start_state.
      template<typename Stepper, typename System, typename State,
               typename Time, typename Observer>
        size_t integrate_adaptive(Stepper stepper, System system,
                                  const State & start_state, Time start_time,
                                  Time end_time, Time dt, Observer observer);

      // integrate_adaptive without an observer.
      template<typename Stepper, typename System, typename State,
               typename Time>
        size_t integrate_adaptive(Stepper stepper, System system,
                                  State & start_state, Time start_time,
                                  Time end_time, Time dt);

      // Second version to solve the forwarding problem, can be called with Boost.Range as ↵
start_state.
      template<typename Stepper, typename System, typename State,
               typename Time>
        size_t integrate_adaptive(Stepper stepper, System system,
                                  const State & start_state, Time start_time,
                                  Time end_time, Time dt);
    }
  }
}
```

Function template integrate_adaptive

boost::numeric::odeint::integrate_adaptive — Integrates the ODE with adaptive step size.

Synopsis

```
// In header: <boost/numeric/odeint/integrate/integrate_adaptive.hpp>


template<typename Stepper, typename System, typename State, typename Time,
         typename Observer>
  size_t integrate_adaptive(Stepper stepper, System system,
                            State & start_state, Time start_time,
                            Time end_time, Time dt, Observer observer);
```

Description

This function integrates the ODE given by system with the given stepper. The observer is called after each step. If the stepper has no error control, the step size remains constant and the observer is called at equidistant time points t0+n*dt. If the stepper is a ControlledStepper, the step size is adjusted and the observer is called in non-equidistant intervals.

Parameters:	`dt`	The time step between observer calls, *not* necessarily the time step of the integration.
	`end_time`	The final integration time tend.
	`observer`	Function/Functor called at equidistant time intervals.
	`start_state`	The initial condition x0.
	`start_time`	The initial time t0.
	`stepper`	The stepper to be used for numerical integration.
	`system`	Function/Functor defining the rhs of the ODE.
Returns:	The number of steps performed.	

Header <boost/numeric/odeint/integrate/integrate_const.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Stepper, typename System, typename State,
               typename Time, typename Observer>
        size_t integrate_const(Stepper, System, State &, Time, Time, Time,
                               Observer);

      // Second version to solve the forwarding problem, can be called with Boost.Range as ↵
start_state.
      template<typename Stepper, typename System, typename State,
               typename Time, typename Observer>
        size_t integrate_const(Stepper stepper, System system,
                               const State & start_state, Time start_time,
                               Time end_time, Time dt, Observer observer);

      // integrate_const without observer calls
      template<typename Stepper, typename System, typename State,
               typename Time>
        size_t integrate_const(Stepper stepper, System system,
                               State & start_state, Time start_time,
                               Time end_time, Time dt);

      // Second version to solve the forwarding problem, can be called with Boost.Range as ↵
start_state.
      template<typename Stepper, typename System, typename State,
               typename Time>
        size_t integrate_const(Stepper stepper, System system,
                               const State & start_state, Time start_time,
                               Time end_time, Time dt);
    }
  }
}
```

Function template integrate_const

boost::numeric::odeint::integrate_const — Integrates the ODE with constant step size.

Synopsis

```
// In header: <boost/numeric/odeint/integrate/integrate_const.hpp>


template<typename Stepper, typename System, typename State, typename Time,
         typename Observer>
  size_t integrate_const(Stepper stepper, System system, State & start_state,
                         Time start_time, Time end_time, Time dt,
                         Observer observer);
```

Description

Integrates the ODE defined by system using the given stepper. This method ensures that the observer is called at constant intervals dt. If the Stepper is a normal stepper without step size control, dt is also used for the numerical scheme. If a ControlledStepper is provided, the algorithm might reduce the step size to meet the error bounds, but it is ensured that the observer is always called at equidistant time points t0 + n*dt. If a DenseOutputStepper is used, the step size also may vary and the dense output is used to call the observer at equidistant time points.

Parameters:	`dt`	The time step between observer calls, *not* necessarily the time step of the integration.
	`end_time`	The final integration time tend.
	`observer`	Function/Functor called at equidistant time intervals.
	`start_state`	The initial condition x0.
	`start_time`	The initial time t0.
	`stepper`	The stepper to be used for numerical integration.
	`system`	Function/Functor defining the rhs of the ODE.
Returns:	The number of steps performed.	

Header <boost/numeric/odeint/integrate/integrate_n_steps.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Stepper, typename System, typename State,
               typename Time, typename Observer>
        Time integrate_n_steps(Stepper, System, State &, Time, Time, size_t,
                               Observer);

      // Solves the forwarding problem, can be called with Boost.Range as start_state.
      template<typename Stepper, typename System, typename State,
               typename Time, typename Observer>
        Time integrate_n_steps(Stepper stepper, System system,
                               const State & start_state, Time start_time,
                               Time dt, size_t num_of_steps,
                               Observer observer);

      // The same function as above, but without observer calls.
      template<typename Stepper, typename System, typename State,
               typename Time>
        Time integrate_n_steps(Stepper stepper, System system,
                               State & start_state, Time start_time, Time dt,
                               size_t num_of_steps);

      // Solves the forwarding problem, can be called with Boost.Range as start_state.
      template<typename Stepper, typename System, typename State,
               typename Time>
        Time integrate_n_steps(Stepper stepper, System system,
                               const State & start_state, Time start_time,
                               Time dt, size_t num_of_steps);
    }
  }
}
```

Function template integrate_n_steps

boost::numeric::odeint::integrate_n_steps — Integrates the ODE with constant step size.

Synopsis

```
// In header: <boost/numeric/odeint/integrate/integrate_n_steps.hpp>


template<typename Stepper, typename System, typename State, typename Time,
         typename Observer>
  Time integrate_n_steps(Stepper stepper, System system, State & start_state,
                         Time start_time, Time dt, size_t num_of_steps,
                         Observer observer);
```

Description

This function is similar to integrate_const. The observer is called at equidistant time intervals t0 + n*dt. If the Stepper is a normal stepper without step size control, dt is also used for the numerical scheme. If a ControlledStepper is provided, the algorithm might reduce the step size to meet the error bounds, but it is ensured that the observer is always called at equidistant time points t0 + n*dt. If a DenseOutputStepper is used, the step size also may vary and the dense output is used to call the observer at equidistant time points. The final integration time is always t0 + num_of_steps*dt.

Parameters:	`dt`	The time step between observer calls, *not* necessarily the time step of the integration.
	`num_of_steps`	Number of steps to be performed
	`observer`	Function/Functor called at equidistant time intervals.
	`start_state`	The initial condition x0.
	`start_time`	The initial time t0.
	`stepper`	The stepper to be used for numerical integration.
	`system`	Function/Functor defining the rhs of the ODE.
Returns:	The number of steps performed.	

Header <boost/numeric/odeint/integrate/integrate_times.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Stepper, typename System, typename State,
               typename TimeIterator, typename Time, typename Observer>
        size_t integrate_times(Stepper, System, State &, TimeIterator,
                               TimeIterator, Time, Observer);

      // Solves the forwarding problem, can be called with Boost.Range as start_state.
      template<typename Stepper, typename System, typename State,
               typename TimeIterator, typename Time, typename Observer>
        size_t integrate_times(Stepper stepper, System system,
                               const State & start_state,
                               TimeIterator times_start,
                               TimeIterator times_end, Time dt,
                               Observer observer);

      // The same function as above, but without observer calls.
      template<typename Stepper, typename System, typename State,
               typename TimeRange, typename Time, typename Observer>
        size_t integrate_times(Stepper stepper, System system,
                               State & start_state, const TimeRange & times,
                               Time dt, Observer observer);

      // Solves the forwarding problem, can be called with Boost.Range as start_state.
      template<typename Stepper, typename System, typename State,
               typename TimeRange, typename Time, typename Observer>
        size_t integrate_times(Stepper stepper, System system,
                               const State & start_state,
                               const TimeRange & times, Time dt,
                               Observer observer);
    }
  }
}
```

Function template integrate_times

boost::numeric::odeint::integrate_times — Integrates the ODE with observer calls at given time points.

Synopsis

```
// In header: <boost/numeric/odeint/integrate/integrate_times.hpp>


template<typename Stepper, typename System, typename State,
         typename TimeIterator, typename Time, typename Observer>
  size_t integrate_times(Stepper stepper, System system, State & start_state,
                         TimeIterator times_start, TimeIterator times_end,
                         Time dt, Observer observer);
```

Description

Integrates the ODE given by system using the given stepper. This function does observer calls at the subsequent time points given by the range times_start, times_end. If the stepper has not step size control, the step size might be reduced occasionally to ensure observer calls exactly at the time points from the given sequence. If the stepper is a ControlledStepper, the step size is adjusted to meet the error bounds, but also might be reduced occasionally to ensure correct observer calls. If a DenseOutputStepper is provided, the dense output functionality is used to call the observer at the given times. The end time of the integration is always *(end_time-1).

Parameters:	`dt`	The time step between observer calls, *not* necessarily the time step of the integration.
	`observer`	Function/Functor called at equidistant time intervals.
	`start_state`	The initial condition x0.
	`stepper`	The stepper to be used for numerical integration.
	`system`	Function/Functor defining the rhs of the ODE.
	`times_end`	Iterator to the end time
	`times_start`	Iterator to the start time
Returns:	The number of steps performed.	

Header <boost/numeric/odeint/integrate/null_observer.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      struct null_observer;
    }
  }
}
```

Struct null_observer

boost::numeric::odeint::null_observer

Synopsis

```
// In header: <boost/numeric/odeint/integrate/null_observer.hpp>


struct null_observer {

  // public member functions
  template<typename State, typename Time>
    void operator()(const State &, Time) const;
};
```

Description

`null_observer` public member functions

1.
```
template<typename State, typename Time>
  void operator()(const State &, Time) const;
```

Header <boost/numeric/odeint/integrate/observer_collection.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Time> class observer_collection;
    }
  }
}
```

Class template observer_collection

boost::numeric::odeint::observer_collection

Synopsis

```
// In header: <boost/numeric/odeint/integrate/observer_collection.hpp>

template<typename State, typename Time>
class observer_collection {
public:
  // types
  typedef boost::function< void(const State &, const Time &) > observer_type;
  typedef std::vector< observer_type >                         collection_type;

  // public member functions
  void operator()(const State &, Time);
  collection_type & observers(void);
  const collection_type & observers(void) const;
};
```

Description

`observer_collection` public member functions

1.
```
void operator()(const State & x, Time t);
```

2.
```
collection_type & observers(void);
```

3.
```
const collection_type & observers(void) const;
```

Header <boost/numeric/odeint/stepper/adams_bashforth.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<size_t Steps, typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer,
               typename InitializingStepper = runge_kutta4< State , Value , Deriv , Time , Al↵
gebra , Operations, Resizer > >
        class adams_bashforth;
    }
  }
}
```

Class template adams_bashforth

boost::numeric::odeint::adams_bashforth — The Adams-Bashforth multistep algorithm.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/adams_bashforth.hpp>

template<size_t Steps, typename State, typename Value = double,
         typename Deriv = State, typename Time = Value,
         typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer,
        typename InitializingStepper = runge_kutta4< State , Value , Deriv , Time , Algebra , Op↵
erations, Resizer > >
class adams_bashforth :
  public boost::numeric::odeint::algebra_stepper_base< Algebra, Operations >
{
public:
  // types
  typedef State                                                 state_type;             ↵

  typedef state_wrapper< state_type >                           wrapped_state_type;     ↵

  typedef Value                                                 value_type;             ↵

  typedef Deriv                                                 deriv_type;             ↵

  typedef state_wrapper< deriv_type >                           wrapped_deriv_type;     ↵

  typedef Time                                                  time_type;              ↵

  typedef Resizer                                               resizer_type;           ↵

  typedef stepper_tag                                           stepper_category;       ↵

  typedef InitializingStepper                                  initializing_stepper_type;
  typedef algebra_stepper_base< Algebra, Operations >::algebra_type    algebra_type;          ↵

  typedef algebra_stepper_base< Algebra, Operations >::operations_type operations_type;       ↵

  typedef unsigned short                                        order_type;             ↵

  typedef unspecified                                           step_storage_type;      ↵


  // construct/copy/destruct
  adams_bashforth(const algebra_type & = algebra_type());
  adams_bashforth(const adams_bashforth &);
  adams_bashforth& operator=(const adams_bashforth &);

  // public member functions
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, const StateOut &,
                 time_type);
  template<typename StateType> void adjust_size(const StateType &);
  const step_storage_type & step_storage(void) const;
  step_storage_type & step_storage(void);
  template<typename ExplicitStepper, typename System, typename StateIn>
```

```
    void initialize(ExplicitStepper, System, StateIn &, time_type &,
                    time_type);
  template<typename System, typename StateIn>
    void initialize(System, StateIn &, time_type &, time_type);
  void reset(void);
  bool is_initialized(void) const;
  const initializing_stepper_type & initializing_stepper(void) const;
  initializing_stepper_type & initializing_stepper(void);
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  template<typename System, typename StateIn, typename StateOut>
    void do_step_impl(System, const StateIn &, time_type, StateOut &,
                      time_type);
  template<typename StateIn> bool resize_impl(const StateIn &);

  // public data members
  static const size_t steps;
  static const order_type order_value;
};
```

Description

The Adams-Bashforth method is a multi-step algorithm with configurable step number. The step number is specified as template parameter Steps and it then uses the result from the previous Steps steps. See also en.wikipedia.org/wiki/Linear_multistep_method. Currently, a maximum of Steps=8 is supported. The method is explicit and fulfills the Stepper concept. Step size control or continuous output are not provided.

This class derives from algebra_base and inherits its interface via CRTP (current recurring template pattern). For more details see algebra_stepper_base.

Template Parameters

1. `size_t Steps`

   The number of steps (maximal 8).

2. `typename State`

   The state type.

3. `typename Value = double`

   The value type.

4. `typename Deriv = State`

   The type representing the time derivative of the state.

5. `typename Time = Value`

   The time representing the independent variable - the time.

6. `typename Algebra = range_algebra`

The algebra type.

7.
```
typename Operations = default_operations
```

The operations type.

8.
```
typename Resizer = initially_resizer
```

The resizer policy type.

9.
```
typename InitializingStepper = runge_kutta4< State , Value , Deriv , Time , Algebra , Opera↵
tions, Resizer >
```

The stepper for the first two steps.

`adams_bashforth` public construct/copy/destruct

1.
```
adams_bashforth(const algebra_type & algebra = algebra_type());
```

Constructs the `adams_bashforth` class. This constructor can be used as a default constructor if the algebra has a default constructor.

Parameters: `algebra` A copy of algebra is made and stored.

2.
```
adams_bashforth(const adams_bashforth & stepper);
```

3.
```
adams_bashforth& operator=(const adams_bashforth & stepper);
```

`adams_bashforth` public member functions

1.
```
order_type order(void) const;
```

Returns the order of the algorithm, which is equal to the number of steps.

Returns: order of the method.

2.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

3.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

4.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place.

Parameters:		
	`dt`	The step size.
	`in`	The state of the ODE which should be solved. in is not modified in this method
	`out`	The result of the step is written in out.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time, at which the step should be performed.

5.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t,
               const StateOut & out, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateOut.

6.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters:		
	`x`	A state from which the size of the temporaries to be resized is deduced.

7.
```
const step_storage_type & step_storage(void) const;
```

Returns the storage of intermediate results.

Returns: The storage of intermediate results.

8.
```
step_storage_type & step_storage(void);
```

Returns the storage of intermediate results.

Returns: The storage of intermediate results.

9.
```
template<typename ExplicitStepper, typename System, typename StateIn>
  void initialize(ExplicitStepper explicit_stepper, System system,
                  StateIn & x, time_type & t, time_type dt);
```

Initialized the stepper. Does Steps-1 steps with the explicit_stepper to fill the buffer.

Parameters:		
	`dt`	The step size.
	`explicit_stepper`	the stepper used to fill the buffer of previous step results
	`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
	`t`	The value of the time, at which the step should be performed.
	`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

10.
```
template<typename System, typename StateIn>
  void initialize(System system, StateIn & x, time_type & t, time_type dt);
```

Initialized the stepper. Does Steps-1 steps with an internal instance of InitializingStepper to fill the buffer.

Note

The state x and time t are updated to the values after Steps-1 initial steps.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The initial value of the time, updated in this method.
`x`	The initial state of the ODE which should be solved, updated in this method.

11.
```
void reset(void);
```

Resets the internal buffer of the stepper.

12.
```
bool is_initialized(void) const;
```

Returns true if the stepper has been initialized.

Returns: bool true if stepper is initialized, false otherwise

13.
```
const initializing_stepper_type & initializing_stepper(void) const;
```

Returns the internal initializing stepper instance.

Returns: initializing_stepper

14.
```
initializing_stepper_type & initializing_stepper(void);
```

Returns the internal initializing stepper instance.

Returns: initializing_stepper

15.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

16.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`adams_bashforth` private member functions

1.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step_impl(System system, const StateIn & in, time_type t,
                    StateOut & out, time_type dt);
```

2.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

Header <boost/numeric/odeint/stepper/adams_bashforth_moulton.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<size_t Steps, typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class adams_bashforth_moulton;
    }
  }
}
```

Class template adams_bashforth_moulton

boost::numeric::odeint::adams_bashforth_moulton — The Adams-Bashforth-Moulton multistep algorithm.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/adams_bashforth_moulton.hpp>

template<size_t Steps, typename State, typename Value = double,
         typename Deriv = State, typename Time = Value,
         typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class adams_bashforth_moulton {
public:
  // types
  typedef State                       state_type;
  typedef state_wrapper< state_type > wrapped_state_type;
  typedef Value                       value_type;
  typedef Deriv                       deriv_type;
  typedef state_wrapper< deriv_type > wrapped_deriv_type;
  typedef Time                        time_type;
  typedef Algebra                     algebra_type;
  typedef Operations                  operations_type;
  typedef Resizer                     resizer_type;
  typedef stepper_tag                 stepper_category;
  typedef unsigned short              order_type;

  // construct/copy/destruct
  adams_bashforth_moulton(void);
  adams_bashforth_moulton(const algebra_type &);

  // public member functions
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, const StateOut &,
                 time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename StateType> void adjust_size(const StateType &);
  template<typename ExplicitStepper, typename System, typename StateIn>
    void initialize(ExplicitStepper, System, StateIn &, time_type &,
                    time_type);
  template<typename System, typename StateIn>
    void initialize(System, StateIn &, time_type &, time_type);

  // public data members
  static const size_t steps;
  static const order_type order_value;
};
```

Description

The Adams-Bashforth method is a multi-step predictor-corrector algorithm with configurable step number. The step number is specified as template parameter Steps and it then uses the result from the previous Steps steps. See also en.wikipedia.org/wiki/Linear_multistep_method. Currently, a maximum of Steps=8 is supported. The method is explicit and fulfills the Stepper concept. Step size control or continuous output are not provided.

This class derives from algebra_base and inherits its interface via CRTP (current recurring template pattern). For more details see algebra_stepper_base.

Template Parameters

1. ```
   size_t Steps
   ```

   The number of steps (maximal 8).

2. ```
   typename State
   ```

   The state type.

3. ```
   typename Value = double
   ```

   The value type.

4. ```
   typename Deriv = State
   ```

   The type representing the time derivative of the state.

5. ```
   typename Time = Value
   ```

   The time representing the independent variable - the time.

6. ```
   typename Algebra = range_algebra
   ```

   The algebra type.

7. ```
   typename Operations = default_operations
   ```

   The operations type.

8. ```
   typename Resizer = initially_resizer
   ```

   The resizer policy type.

`adams_bashforth_moulton` public construct/copy/destruct

1. ```
   adams_bashforth_moulton(void);
   ```

   Constructs the `adams_bashforth` class.

2. ```
   adams_bashforth_moulton(const algebra_type & algebra);
   ```

   Constructs the `adams_bashforth` class. This constructor can be used as a default constructor if the algebra has a default constructor.

   Parameters: `algebra` A copy of algebra is made and stored.

`adams_bashforth_moulton` public member functions

1. ```
   order_type order(void) const;
   ```

   Returns the order of the algorithm, which is equal to the number of steps+1.
   ```

Returns: order of the method.

2.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

dt	The step size.
system	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
t	The value of the time, at which the step should be performed.
x	The state of the ODE which should be solved. After calling do_step the result is updated in x.

3.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

4.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t,
               const StateOut & out, time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place.

Parameters:

dt	The step size.
in	The state of the ODE which should be solved. in is not modified in this method
out	The result of the step is written in out.
system	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
t	The value of the time, at which the step should be performed.

5.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateOut.

6.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters:

x	A state from which the size of the temporaries to be resized is deduced.

7.
```
template<typename ExplicitStepper, typename System, typename StateIn>
  void initialize(ExplicitStepper explicit_stepper, System system,
                  StateIn & x, time_type & t, time_type dt);
```

Initialized the stepper. Does Steps-1 steps with the explicit_stepper to fill the buffer.

Note

The state x and time t are updated to the values after Steps-1 initial steps.

Parameters:

dt	The step size.
explicit_stepper	the stepper used to fill the buffer of previous step results

system	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
t	The initial time, updated in this method.
x	The initial state of the ODE which should be solved, updated after in this method.

8.
```
template<typename System, typename StateIn>
  void initialize(System system, StateIn & x, time_type & t, time_type dt);
```

Initialized the stepper. Does Steps-1 steps using the standard initializing stepper of the underlying `adams_bashforth` stepper.

Parameters:

dt	The step size.
system	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
t	The value of the time, at which the step should be performed.
x	The state of the ODE which should be solved. After calling do_step the result is updated in x.

Header <boost/numeric/odeint/stepper/adams_moulton.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<size_t Steps, typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class adams_moulton;
    }
  }
}
```

Class template adams_moulton

boost::numeric::odeint::adams_moulton

Synopsis

```
// In header: <boost/numeric/odeint/stepper/adams_moulton.hpp>

template<size_t Steps, typename State, typename Value = double,
         typename Deriv = State, typename Time = Value,
         typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class adams_moulton {
public:
  // types
  typedef State                                                                 ↵
 state_type;
  typedef state_wrapper< state_type >                                           ↵
 wrapped_state_type;
  typedef Value                                                                 ↵
 value_type;
  typedef Deriv                                                                  de↵
riv_type;
  typedef state_wrapper< deriv_type >                                           ↵
 wrapped_deriv_type;
  typedef Time                                                                  ↵
 time_type;
  typedef Algebra                                                                al↵
gebra_type;
  typedef Operations                                                            opera↵
tions_type;
  typedef Resizer                                                                res↵
izer_type;
  typedef stepper_tag                                                           step↵
per_category;
  typedef adams_moulton< Steps, State, Value, Deriv, Time, Algebra, Operations, Resizer > step↵
per_type;
  typedef unsigned short                                                         or↵
der_type;
  typedef unspecified                                                       step_stor↵
age_type;

  // construct/copy/destruct
  adams_moulton();
  adams_moulton(algebra_type &);
  adams_moulton& operator=(const adams_moulton &);

  // public member functions
  order_type order(void) const;
  template<typename System, typename StateInOut, typename ABBuf>
    void do_step(System, StateInOut &, time_type, time_type, const ABBuf &);
  template<typename System, typename StateInOut, typename ABBuf>
    void do_step(System, const StateInOut &, time_type, time_type,
                 const ABBuf &);
  template<typename System, typename StateIn, typename StateOut,
           typename ABBuf>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type,
                 const ABBuf &);
  template<typename System, typename StateIn, typename StateOut,
           typename ABBuf>
    void do_step(System, const StateIn &, time_type, const StateOut &,
                 time_type, const ABBuf &);
  template<typename StateType> void adjust_size(const StateType &);
  algebra_type & algebra();
  const algebra_type & algebra() const;
```

```
  // private member functions
  template<typename StateIn> bool resize_impl(const StateIn &);

  // public data members
  static const size_t steps;
  static const order_type order_value;
};
```

Description

`adams_moulton` public construct/copy/destruct

1.
```
adams_moulton();
```

2.
```
adams_moulton(algebra_type & algebra);
```

3.
```
adams_moulton& operator=(const adams_moulton & stepper);
```

`adams_moulton` public member functions

1.
```
order_type order(void) const;
```

2.
```
template<typename System, typename StateInOut, typename ABBuf>
  void do_step(System system, StateInOut & in, time_type t, time_type dt,
               const ABBuf & buf);
```

3.
```
template<typename System, typename StateInOut, typename ABBuf>
  void do_step(System system, const StateInOut & in, time_type t,
               time_type dt, const ABBuf & buf);
```

4.
```
template<typename System, typename StateIn, typename StateOut, typename ABBuf>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt, const ABBuf & buf);
```

5.
```
template<typename System, typename StateIn, typename StateOut, typename ABBuf>
  void do_step(System system, const StateIn & in, time_type t,
               const StateOut & out, time_type dt, const ABBuf & buf);
```

6.
```
template<typename StateType> void adjust_size(const StateType & x);
```

7.
```
algebra_type & algebra();
```

8.
```
const algebra_type & algebra() const;
```

`adams_moulton` private member functions

1.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

Header <boost/numeric/odeint/stepper/base/algebra_stepper_base.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Algebra, typename Operations> class algebra_stepper_base;
    }
  }
}
```

Class template algebra_stepper_base

boost::numeric::odeint::algebra_stepper_base — Base class for all steppers with algebra and operations.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/base/algebra_stepper_base.hpp>

template<typename Algebra, typename Operations>
class algebra_stepper_base {
public:
  // types
  typedef Algebra    algebra_type;
  typedef Operations operations_type;

  // construct/copy/destruct
  algebra_stepper_base(const algebra_type & = algebra_type());

  // public member functions
  algebra_type & algebra();
  const algebra_type & algebra() const;
};
```

Description

This class serves a base class for all steppers with algebra and operations. It holds the algebra and provides access to the algebra. The operations are not instantiated, since they are static classes inside the operations class.

Template Parameters

1.
```
typename Algebra
```

The type of the algebra. Must fulfill the Algebra Concept, at least partially to work with the stepper.

2.
```
typename Operations
```

The type of the operations. Must fulfill the Operations Concept, at least partially to work with the stepper.

`algebra_stepper_base` public construct/copy/destruct

1.
```
algebra_stepper_base(const algebra_type & algebra = algebra_type());
```

Constructs a `algebra_stepper_base` and creates the algebra. This constructor can be used as a default constructor if the algebra has a default constructor.

Parameters: `algebra` The `algebra_stepper_base` stores and uses a copy of algebra.

`algebra_stepper_base` public member functions

1.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

2.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

Header <boost/numeric/odeint/stepper/base/explicit_error_stepper_base.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Stepper, unsigned short Order,
               unsigned short StepperOrder, unsigned short ErrorOrder,
               typename State, typename Value, typename Deriv, typename Time,
               typename Algebra, typename Operations, typename Resizer>
        class explicit_error_stepper_base;
    }
  }
}
```

Class template explicit_error_stepper_base

boost::numeric::odeint::explicit_error_stepper_base — Base class for explicit steppers with error estimation. This class can used with controlled steppers for step size control.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/base/explicit_error_stepper_base.hpp>

template<typename Stepper, unsigned short Order, unsigned short StepperOrder,
         unsigned short ErrorOrder, typename State, typename Value,
         typename Deriv, typename Time, typename Algebra, typename Operations,
         typename Resizer>
class explicit_error_stepper_base :
  public boost::numeric::odeint::algebra_stepper_base< Algebra, Operations >
{
public:
  // types
  typedef algebra_stepper_base< Algebra, Operations > algebra_stepper_base_type;
  typedef algebra_stepper_base_type::algebra_type     algebra_type;
  typedef State                                       state_type;
  typedef Value                                       value_type;
  typedef Deriv                                       deriv_type;
  typedef Time                                        time_type;
  typedef Resizer                                     resizer_type;
  typedef Stepper                                     stepper_type;
  typedef explicit_error_stepper_tag                  stepper_category;
  typedef unsigned short                              order_type;

  // construct/copy/destruct
  explicit_error_stepper_base(const algebra_type & = algebra_type());

  // public member functions
  order_type order(void) const;
  order_type stepper_order(void) const;
  order_type error_order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
    do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, const StateIn &, const DerivIn &, time_type, StateOut &,
            time_type);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, const StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename DerivIn,
           typename Err>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type,
            Err &);
  template<typename System, typename StateIn, typename StateOut, typename Err>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type,
                 Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename Err>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type, Err &);
```

```
  template<typename StateIn> void adjust_size(const StateIn &);
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  template<typename System, typename StateInOut>
    void do_step_v1(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename Err>
    void do_step_v5(System, StateInOut &, time_type, time_type, Err &);
  template<typename StateIn> bool resize_impl(const StateIn &);
  stepper_type & stepper(void);
  const stepper_type & stepper(void) const;

  // public data members
  static const order_type order_value;
  static const order_type stepper_order_value;
  static const order_type error_order_value;
};
```

Description

This class serves as the base class for all explicit steppers with algebra and operations. In contrast to explicit_stepper_base it also estimates the error and can be used in a controlled stepper to provide step size control.

Note

This stepper provides `do_step` methods with and without error estimation. It has therefore three orders, one for the order of a step if the error is not estimated. The other two orders are the orders of the step and the error step if the error estimation is performed.

explicit_error_stepper_base is used as the interface in a CRTP (currently recurring template pattern). In order to work correctly the parent class needs to have a method `do_step_impl( system , in , dxdt_in , t , out , dt , xerr )`. explicit_error_stepper_base derives from algebra_stepper_base.

explicit_error_stepper_base provides several overloaded `do_step` methods, see the list below. Only two of them are needed to fulfill the Error Stepper concept. The other ones are for convenience and for performance. Some of them simply update the state out-of-place, while other expect that the first derivative at `t` is passed to the stepper.

- `do_step( sys , x , t , dt )` - The classical `do_step` method needed to fulfill the Error Stepper concept. The state is updated in-place. A type modelling a Boost.Range can be used for x.
- `do_step( sys , x , dxdt , t , dt )` - This method updates the state in-place, but the derivative at the point `t` must be explicitly passed in `dxdt`.
- `do_step( sys , in , t , out , dt )` - This method updates the state out-of-place, hence the result of the step is stored in `out`.
- `do_step( sys , in , dxdt , t , out , dt )` - This method update the state out-of-place and expects that the derivative at the point `t` is explicitly passed in `dxdt`. It is a combination of the two `do_step` methods above.
- `do_step( sys , x , t , dt , xerr )` - This `do_step` method is needed to fulfill the Error Stepper concept. The state is updated in-place and an error estimate is calculated. A type modelling a Boost.Range can be used for x.
- `do_step( sys , x , dxdt , t , dt , xerr )` - This method updates the state in-place, but the derivative at the point `t` must be passed in `dxdt`. An error estimate is calculated.
- `do_step( sys , in , t , out , dt , xerr )` - This method updates the state out-of-place and estimates the error during the step.

- `do_step( sys , in , dxdt , t , out , dt , xerr )` - This methods updates the state out-of-place and estimates the error during the step. Furthermore, the derivative at `t` must be passed in `dxdt`.

Note

The system is always passed as value, which might result in poor performance if it contains data. In this case it can be used with `boost::ref` or `std::ref`, for example `stepper.do_step( boost::ref( sys ) , x , t , dt );`

The time `t` is not advanced by the stepper. This has to done manually, or by the appropriate `integrate` routines or `iterators`.

Template Parameters

1. ```
   typename Stepper
   ```

   The stepper on which this class should work. It is used via CRTP, hence `explicit_stepper_base` provides the interface for the Stepper.

2. ```
   unsigned short Order
   ```

   The order of a stepper if the stepper is used without error estimation.

3. ```
   unsigned short StepperOrder
   ```

   The order of a step if the stepper is used with error estimation. Usually Order and StepperOrder have the same value.

4. ```
   unsigned short ErrorOrder
   ```

   The order of the error step if the stepper is used with error estimation.

5. ```
   typename State
   ```

   The state type for the stepper.

6. ```
   typename Value
   ```

   The value type for the stepper. This should be a floating point type, like float, double, or a multiprecision type. It must not necessary be the value_type of the State. For example the State can be a `vector< complex< double > >` in this case the Value must be double. The default value is double.

7. ```
   typename Deriv
   ```

   The type representing time derivatives of the state type. It is usually the same type as the state type, only if used with Boost.Units both types differ.

8. ```
   typename Time
   ```

   The type representing the time. Usually the same type as the value type. When Boost.Units is used, this type has usually a unit.

9. ```
   typename Algebra
   ```
   ```

The algebra type which must fulfill the Algebra Concept.

10.
```
typename Operations
```

The type for the operations which must fulfill the Operations Concept.

11.
```
typename Resizer
```

The resizer policy class.

`explicit_error_stepper_base` public construct/copy/destruct

1.
```
explicit_error_stepper_base(const algebra_type & algebra = algebra_type());
```

Constructs a `explicit_error_stepper_base` class. This constructor can be used as a default constructor if the algebra has a default constructor.

Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`explicit_error_stepper_base` public member functions

1.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper if it used without error estimation.

2.
```
order_type stepper_order(void) const;
```

Returns: Returns the order of a step if the stepper is used without error estimation.

3.
```
order_type error_order(void) const;
```

Returns: Returns the order of an error step if the stepper is used without error estimation.

4.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

5.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

6.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

7.
```
template<typename System, typename StateIn, typename StateOut>
  boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
  do_step(System system, const StateIn & in, time_type t, StateOut & out,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. This method is disabled if StateIn and Time are the same type. In this case the method can not be distinguished from other `do_step` variants.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

8.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, const StateIn & in, const DerivIn & dxdt,
          time_type t, StateOut & out, time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

This method is disabled if DerivIn and Time are of same type.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

9.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

The method performs one step with the stepper passed by Stepper and estimates the error. The state of the ODE is updated in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. x is updated by this method.
`xerr`	The estimation of the error is stored in xerr.

10.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

11.
```
template<typename System, typename StateInOut, typename DerivIn, typename Err>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt , xerr );
```

The result is updated in place in x. This method is disabled if Time and DerivIn are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.
`xerr`	The error estimate is stored in xerr.

12.
```
template<typename System, typename StateIn, typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the error is estimated.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

13.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper and the error is estimated. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

This method is disabled if DerivIn and Time are of same type.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

14.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters:

`x`	A state from which the size of the temporaries to be resized is deduced.

15.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

16.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`explicit_error_stepper_base` private member functions

1.
```
template<typename System, typename StateInOut>
  void do_step_v1(System system, StateInOut & x, time_type t, time_type dt);
```

2.
```
template<typename System, typename StateInOut, typename Err>
  void do_step_v5(System system, StateInOut & x, time_type t, time_type dt,
                  Err & xerr);
```

3.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

4.
```
stepper_type & stepper(void);
```

5.
```
const stepper_type & stepper(void) const;
```

Header <boost/numeric/odeint/stepper/base/explicit_error_stepper_fsal_base.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Stepper, unsigned short Order,
               unsigned short StepperOrder, unsigned short ErrorOrder,
               typename State, typename Value, typename Deriv, typename Time,
               typename Algebra, typename Operations, typename Resizer>
        class explicit_error_stepper_fsal_base;
    }
  }
}
```

Class template explicit_error_stepper_fsal_base

boost::numeric::odeint::explicit_error_stepper_fsal_base — Base class for explicit steppers with error estimation and stepper fulfilling the FSAL (first-same-as-last) property. This class can be used with controlled steppers for step size control.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/base/explicit_error_stepper_fsal_base.hpp>

template<typename Stepper, unsigned short Order, unsigned short StepperOrder,
         unsigned short ErrorOrder, typename State, typename Value,
         typename Deriv, typename Time, typename Algebra, typename Operations,
         typename Resizer>
class explicit_error_stepper_fsal_base :
  public boost::numeric::odeint::algebra_stepper_base< Algebra, Operations >
{
public:
  // types
  typedef algebra_stepper_base< Algebra, Operations > algebra_stepper_base_type;
  typedef algebra_stepper_base_type::algebra_type     algebra_type;
  typedef State                                       state_type;
  typedef Value                                       value_type;
  typedef Deriv                                       deriv_type;
  typedef Time                                        time_type;
  typedef Resizer                                     resizer_type;
  typedef Stepper                                     stepper_type;
  typedef explicit_error_stepper_fsal_tag             stepper_category;
  typedef unsigned short                              order_type;

  // construct/copy/destruct
  explicit_error_stepper_fsal_base(const algebra_type & = algebra_type());

  // public member functions
  order_type order(void) const;
  order_type stepper_order(void) const;
  order_type error_order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivInOut>
    boost::disable_if< boost::is_same< StateInOut, time_type >, void >::type
    do_step(System, StateInOut &, DerivInOut &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
    do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename DerivOut>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, DerivOut &, time_type);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, const StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename DerivInOut,
           typename Err>
    boost::disable_if< boost::is_same< StateInOut, time_type >, void >::type
    do_step(System, StateInOut &, DerivInOut &, time_type, time_type, Err &);
  template<typename System, typename StateIn, typename StateOut, typename Err>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type,
                 Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename DerivOut, typename Err>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, DerivOut &, time_type, Err &);
  template<typename StateIn> void adjust_size(const StateIn &);
  void reset(void);
```

```
  template<typename DerivIn> void initialize(const DerivIn &);
  template<typename System, typename StateIn>
    void initialize(System, const StateIn &, time_type);
  bool is_initialized(void) const;
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  template<typename System, typename StateInOut>
    void do_step_v1(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename Err>
    void do_step_v5(System, StateInOut &, time_type, time_type, Err &);
  template<typename StateIn> bool resize_impl(const StateIn &);
  stepper_type & stepper(void);
  const stepper_type & stepper(void) const;

  // public data members
  static const order_type order_value;
  static const order_type stepper_order_value;
  static const order_type error_order_value;
};
```

Description

This class serves as the base class for all explicit steppers with algebra and operations and which fulfill the FSAL property. In contrast to explicit_stepper_base it also estimates the error and can be used in a controlled stepper to provide step size control.

The FSAL property means that the derivative of the system at t+dt is already used in the current step going from t to t +dt. Therefore, some more do_steps method can be introduced and the controlled steppers can explicitly make use of this property.

Note

This stepper provides `do_step` methods with and without error estimation. It has therefore three orders, one for the order of a step if the error is not estimated. The other two orders are the orders of the step and the error step if the error estimation is performed.

explicit_error_stepper_fsal_base is used as the interface in a CRTP (currently recurring template pattern). In order to work correctly the parent class needs to have a method `do_step_impl( system , in , dxdt_in , t , out , dxdt_out , dt , xerr )`. explicit_error_stepper_fsal_base derives from algebra_stepper_base.

This class can have an intrinsic state depending on the explicit usage of the `do_step` method. This means that some `do_step` methods are expected to be called in order. For example the `do_step( sys , x , t , dt , xerr )` will keep track of the derivative of `x` which is the internal state. The first call of this method is recognized such that one does not explicitly initialize the internal state, so it is safe to use this method like

```
stepper_type stepper;
stepper.do_step( sys , x , t , dt , xerr );
stepper.do_step( sys , x , t , dt , xerr );
stepper.do_step( sys , x , t , dt , xerr );
```

But it is unsafe to call this method with different system functions after each other. Do do so, one must initialize the internal state with the `initialize` method or reset the internal state with the `reset` method.

explicit_error_stepper_fsal_base provides several overloaded `do_step` methods, see the list below. Only two of them are needed to fulfill the Error Stepper concept. The other ones are for convenience and for better performance. Some of them simply update the state out-of-place, while other expect that the first derivative at `t` is passed to the stepper.

- `do_step( sys , x , t , dt )` - The classical `do_step` method needed to fulfill the Error Stepper concept. The state is updated in-place. A type modelling a Boost.Range can be used for x.
- `do_step( sys , x , dxdt , t , dt )` - This method updates the state x and the derivative dxdt in-place. It is expected that dxdt has the value of the derivative of x at time t.
- `do_step( sys , in , t , out , dt )` - This method updates the state out-of-place, hence the result of the step is stored in `out`.
- `do_step( sys , in , dxdt_in , t , out , dxdt_out , dt )` - This method updates the state and the derivative out-of-place. It expects that the derivative at the point `t` is explicitly passed in `dxdt_in`.
- `do_step( sys , x , t , dt , xerr )` - This `do_step` method is needed to fulfill the Error Stepper concept. The state is updated in-place and an error estimate is calculated. A type modelling a Boost.Range can be used for x.
- `do_step( sys , x , dxdt , t , dt , xerr )` - This method updates the state and the derivative in-place. It is assumed that the dxdt has the value of the derivative of x at time t. An error estimate is calculated.
- `do_step( sys , in , t , out , dt , xerr )` - This method updates the state out-of-place and estimates the error during the step.
- `do_step( sys , in , dxdt_in , t , out , dxdt_out , dt , xerr )` - This methods updates the state and the derivative out-of-place and estimates the error during the step. It is assumed the dxdt_in is derivative of in at time t.

Note

The system is always passed as value, which might result in poor performance if it contains data. In this case it can be used with `boost::ref` or `std::ref`, for example `stepper.do_step( boost::ref( sys ) , x , t , dt );`

The time `t` is not advanced by the stepper. This has to done manually, or by the appropriate `integrate` routines or `iterators`.

Template Parameters

1. ```
   typename Stepper
   ```

   The stepper on which this class should work. It is used via CRTP, hence `explicit_stepper_base` provides the interface for the Stepper.

2. ```
   unsigned short Order
   ```

   The order of a stepper if the stepper is used without error estimation.

3. ```
   unsigned short StepperOrder
   ```

   The order of a step if the stepper is used with error estimation. Usually Order and StepperOrder have the same value.

4. ```
   unsigned short ErrorOrder
   ```

   The order of the error step if the stepper is used with error estimation.

5. ```
   typename State
   ```

   The state type for the stepper.
   ```

6.
```
typename Value
```

The value type for the stepper. This should be a floating point type, like float, double, or a multiprecision type. It must not necessary be the value_type of the State. For example the State can be a `vector< complex< double > >` in this case the Value must be double. The default value is double.

7.
```
typename Deriv
```

The type representing time derivatives of the state type. It is usually the same type as the state type, only if used with Boost.Units both types differ.

8.
```
typename Time
```

The type representing the time. Usually the same type as the value type. When Boost.Units is used, this type has usually a unit.

9.
```
typename Algebra
```

The algebra type which must fulfill the Algebra Concept.

10.
```
typename Operations
```

The type for the operations which must fulfill the Operations Concept.

11.
```
typename Resizer
```

The resizer policy class.

`explicit_error_stepper_fsal_base` public construct/copy/destruct

1.
```
explicit_error_stepper_fsal_base(const algebra_type & algebra = algebra_type());
```

Constructs a explicit_stepper_fsal_base class. This constructor can be used as a default constructor if the algebra has a default constructor.

Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`explicit_error_stepper_fsal_base` public member functions

1.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper if it used without error estimation.

2.
```
order_type stepper_order(void) const;
```

Returns: Returns the order of a step if the stepper is used without error estimation.

3.
```
order_type error_order(void) const;
```

Returns: Returns the order of an error step if the stepper is used without error estimation.

4.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Note

This method uses the internal state of the stepper.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

5.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

6.
```
template<typename System, typename StateInOut, typename DerivInOut>
  boost::disable_if< boost::is_same< StateInOut, time_type >, void >::type
  do_step(System system, StateInOut & x, DerivInOut & dxdt, time_type t,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other methods the derivative of x is also passed to this method. Therefore, dxdt must be evaluated initially:

```
ode( x , dxdt , t );
for( ... )
{
    stepper.do_step( ode , x , dxdt , t , dt );
    t += dt;
}
```

Note

This method does NOT use the initial state, since the first derivative is explicitly passed to this method.

The result is updated in place in x as well as the derivative dxdt. This method is disabled if Time and StateInOut are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t. After calling `do_step` dxdt is updated to the new value.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

7.
```
template<typename System, typename StateIn, typename StateOut>
  boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
  do_step(System system, const StateIn & in, time_type t, StateOut & out,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. This method is disabled if StateIn and Time are the same type. In this case the method can not be distinguished from other `do_step` variants.

Note

This method uses the internal state of the stepper.

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

8.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename DerivOut>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt_in,
               time_type t, StateOut & out, DerivOut & dxdt_out,
               time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper and updated by the stepper to its new value at t+dt.

Note

This method does not solve the forwarding problem.

This method does NOT use the internal state of the stepper.

Parameters:

`dt`	The step size.
`dxdt_in`	The derivative of x at t.
`dxdt_out`	The updated derivative of `out` at `t+dt`.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

9.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

The method performs one step with the stepper passed by Stepper and estimates the error. The state of the ODE is updated in-place.

Note

This method uses the internal state of the stepper.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. x is updated by this method.
`xerr`	The estimation of the error is stored in xerr.

10.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

11.
```
template<typename System, typename StateInOut, typename DerivInOut,
         typename Err>
  boost::disable_if< boost::is_same< StateInOut, time_type >, void >::type
  do_step(System system, StateInOut & x, DerivInOut & dxdt, time_type t,
          time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method and updated by this method.

Note

This method does NOT use the internal state of the stepper.

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions. This method is disabled if StateInOut and Time are of the same type.

Note

This method does NOT use the internal state of the stepper.

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t. After calling `do_step` this value is updated to the new value at `t+dt`.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.
`xerr`	The error estimate is stored in xerr.

12.
```
template<typename System, typename StateIn, typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the error is estimated.

Note

This method uses the internal state of the stepper.

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

13.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename DerivOut, typename Err>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt_in,
               time_type t, StateOut & out, DerivOut & dxdt_out,
               time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper and the error is estimated.

Note

This method does NOT use the internal state of the stepper.

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt_in`	The derivative of x at t.
`dxdt_out`	The new derivative at `t+dt` is written into this variable.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

14.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters:

`x`	A state from which the size of the temporaries to be resized is deduced.

15.
```
void reset(void);
```

Resets the internal state of this stepper. After calling this method it is safe to use all `do_step` method without explicitly initializing the stepper.

16.
```
template<typename DerivIn> void initialize(const DerivIn & deriv);
```

Initializes the internal state of the stepper.

Parameters:

`deriv`	The derivative of x. The next call of `do_step` expects that the derivative of `x` passed to `do_step` has the value of `deriv`.

17.
```
template<typename System, typename StateIn>
  void initialize(System system, const StateIn & x, time_type t);
```

Initializes the internal state of the stepper.

This method is equivalent to

```
Deriv dxdt;
system( x , dxdt , t );
stepper.initialize( dxdt );
```

Parameters:

`system`	The system function for the next calls of `do_step`.
`t`	The current time of the ODE.
`x`	The current state of the ODE.

18.
```
bool is_initialized(void) const;
```

Returns if the stepper is already initialized. If the stepper is not initialized, the first call of `do_step` will initialize the state of the stepper. If the stepper is already initialized the system function can not be safely exchanged between consecutive `do_step` calls.

19.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

20.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`explicit_error_stepper_fsal_base` private member functions

1.
```
template<typename System, typename StateInOut>
  void do_step_v1(System system, StateInOut & x, time_type t, time_type dt);
```

2.
```
template<typename System, typename StateInOut, typename Err>
  void do_step_v5(System system, StateInOut & x, time_type t, time_type dt,
                  Err & xerr);
```

3.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

4.
```
stepper_type & stepper(void);
```

5.
```
const stepper_type & stepper(void) const;
```

Header <boost/numeric/odeint/stepper/base/explicit_stepper_base.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Stepper, unsigned short Order, typename State,
               typename Value, typename Deriv, typename Time,
               typename Algebra, typename Operations, typename Resizer>
        class explicit_stepper_base;
    }
  }
}
```

Class template explicit_stepper_base

boost::numeric::odeint::explicit_stepper_base — Base class for explicit steppers without step size control and without dense output.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/base/explicit_stepper_base.hpp>

template<typename Stepper, unsigned short Order, typename State,
         typename Value, typename Deriv, typename Time, typename Algebra,
         typename Operations, typename Resizer>
class explicit_stepper_base :
  public boost::numeric::odeint::algebra_stepper_base< Algebra, Operations >
{
public:
  // types
  typedef State                                          state_type;
  typedef Value                                          value_type;
  typedef Deriv                                          deriv_type;
  typedef Time                                           time_type;
  typedef Resizer                                        resizer_type;
  typedef Stepper                                        stepper_type;
  typedef stepper_tag                                    stepper_category;
  typedef algebra_stepper_base< Algebra, Operations >    algebra_stepper_base_type;
  typedef algebra_stepper_base_type::algebra_type        algebra_type;
  typedef algebra_stepper_base_type::operations_type     operations_type;
  typedef unsigned short                                 order_type;

  // construct/copy/destruct
  explicit_stepper_base(const algebra_type & = algebra_type());

  // public member functions
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type);
  template<typename StateIn> void adjust_size(const StateIn &);
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  stepper_type & stepper(void);
  const stepper_type & stepper(void) const;
  template<typename StateIn> bool resize_impl(const StateIn &);
  template<typename System, typename StateInOut>
    void do_step_v1(System, StateInOut &, time_type, time_type);

  // public data members
  static const order_type order_value;
};
```

Description

This class serves as the base class for all explicit steppers with algebra and operations. Step size control and error estimation as well as dense output are not provided. explicit_stepper_base is used as the interface in a CRTP (currently recurring template pattern). In order to work correctly the parent class needs to have a method `do_step_impl( system , in , dxdt_in , t , out , dt`

). This is method is used by explicit_stepper_base. explicit_stepper_base derives from algebra_stepper_base. An example how this class can be used is

```
template< class State , class Value , class Deriv , class Time , class Algebra , class Opera↵
tions , class Resizer >
class custom_euler : public explicit_stepper_base< 1 , State , Value , Deriv , Time , Algebra , Op↵
erations , Resizer >
{
 public:

    typedef explicit_stepper_base< 1 , State , Value , Deriv , Time , Algebra , Operations , Res↵
izer > base_type;

    custom_euler( const Algebra &algebra = Algebra() ) { }

    template< class Sys , class StateIn , class DerivIn , class StateOut >
    void do_step_impl( Sys sys , const StateIn &in , const Deriv↵
In &dxdt , Time t , StateOut &out , Time dt )
    {
       m_algebra.for_each3( out , in , dxdt , Operations::scale_sum2< Value , Time >( 1.0 , dt );
    }

    template< class State >
    void adjust_size( const State &x )
    {
        base_type::adjust_size( x );
    }
};
```

For the Stepper concept only the `do_step( sys , x , t , dt )` needs to be implemented. But this class provides additional `do_step` variants since the stepper is explicit. These methods can be used to increase the performance in some situation, for example if one needs to analyze `dxdt` during each step. In this case one can use

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );  // the value of dxdt is used here
t += dt;
```

In detail explicit_stepper_base provides the following `do_step` variants

- `do_step( sys , x , t , dt )` - The classical `do_step` method needed to fulfill the Stepper concept. The state is updated in-place. A type modelling a Boost.Range can be used for x.
- `do_step( sys , in , t , out , dt )` - This method updates the state out-of-place, hence the result of the step is stored in `out`.
- `do_step( sys , x , dxdt , t , dt )` - This method updates the state in-place, but the derivative at the point `t` must be explicitly passed in `dxdt`. For an example see the code snippet above.
- `do_step( sys , in , dxdt , t , out , dt )` - This method update the state out-of-place and expects that the derivative at the point `t` is explicitly passed in `dxdt`. It is a combination of the two `do_step` methods above.

Note

The system is always passed as value, which might result in poor performance if it contains data. In this case it can be used with `boost::ref` or `std::ref`, for example `stepper.do_step( boost::ref( sys ) , x , t , dt );`

The time `t` is not advanced by the stepper. This has to done manually, or by the appropriate `integrate` routines or `iterators`.

Template Parameters

1. ```
   typename Stepper
   ```

   The stepper on which this class should work. It is used via CRTP, hence `explicit_stepper_base` provides the interface for the Stepper.

2. ```
   unsigned short Order
   ```

   The order of the stepper.

3. ```
   typename State
   ```

   The state type for the stepper.

4. ```
   typename Value
   ```

   The value type for the stepper. This should be a floating point type, like float, double, or a multiprecision type. It must not necessary be the value_type of the State. For example the State can be a `vector< complex< double > >` in this case the Value must be double. The default value is double.

5. ```
   typename Deriv
   ```

   The type representing time derivatives of the state type. It is usually the same type as the state type, only if used with Boost.Units both types differ.

6. ```
   typename Time
   ```

   The type representing the time. Usually the same type as the value type. When Boost.Units is used, this type has usually a unit.

7. ```
   typename Algebra
   ```

   The algebra type which must fulfill the Algebra Concept.

8. ```
   typename Operations
   ```

   The type for the operations which must fulfill the Operations Concept.

9. ```
   typename Resizer
   ```

   The resizer policy class.

`explicit_stepper_base` public construct/copy/destruct

1. ```
   explicit_stepper_base(const algebra_type & algebra = algebra_type());
   ```

   Constructs a `explicit_stepper_base` class. This constructor can be used as a default constructor if the algebra has a default constructor.

   Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`explicit_stepper_base` public member functions

1.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper.

2.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

3.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

4.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

5.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

6.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

7.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

8.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

9.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`explicit_stepper_base` private member functions

1.
```
stepper_type & stepper(void);
```

2.
```
const stepper_type & stepper(void) const;
```

3.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

4.
```
template<typename System, typename StateInOut>
  void do_step_v1(System system, StateInOut & x, time_type t, time_type dt);
```

Header <boost/numeric/odeint/stepper/base/symplectic_rkn_stepper_base.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<size_t NumOfStages, unsigned short Order, typename Coor,
               typename Momentum, typename Value, typename CoorDeriv,
               typename MomentumDeriv, typename Time, typename Algebra,
               typename Operations, typename Resizer>
        class symplectic_nystroem_stepper_base;
    }
  }
}
```

Class template symplectic_nystroem_stepper_base

boost::numeric::odeint::symplectic_nystroem_stepper_base — Base class for all symplectic steppers of Nystroem type.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/base/symplectic_rkn_stepper_base.hpp>

template<size_t NumOfStages, unsigned short Order, typename Coor,
         typename Momentum, typename Value, typename CoorDeriv,
         typename MomentumDeriv, typename Time, typename Algebra,
         typename Operations, typename Resizer>
class symplectic_nystroem_stepper_base :
  public boost::numeric::odeint::algebra_stepper_base< Algebra, Operations >
{
public:
  // types
  typedef algebra_stepper_base< Algebra, Operations >       algebra_stepper_base_type;
  typedef algebra_stepper_base_type::algebra_type           algebra_type;
  typedef algebra_stepper_base_type::operations_type        operations_type;
  typedef Coor                                              coor_type;
  typedef Momentum                                          momentum_type;
  typedef std::pair< coor_type, momentum_type >             state_type;
  typedef CoorDeriv                                         coor_deriv_type;
  typedef state_wrapper< coor_deriv_type >                  wrapped_coor_deriv_type;
  typedef MomentumDeriv                                     momentum_deriv_type;
  typedef state_wrapper< momentum_deriv_type >              wrapped_momentum_deriv_type;
  typedef std::pair< coor_deriv_type, momentum_deriv_type > deriv_type;
  typedef Value                                             value_type;
  typedef Time                                              time_type;
  typedef Resizer                                           resizer_type;
  typedef stepper_tag                                       stepper_category;
  typedef unsigned short                                    order_type;
  typedef boost::array< value_type, num_of_stages >         coef_type;

  // construct/copy/destruct
  symplectic_nystroem_stepper_base(const coef_type &, const coef_type &,
                                   const algebra_type & = algebra_type());

  // public member functions
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename CoorInOut, typename MomentumInOut>
    void do_step(System, CoorInOut &, MomentumInOut &, time_type, time_type);
  template<typename System, typename CoorInOut, typename MomentumInOut>
    void do_step(System, const CoorInOut &, const MomentumInOut &, time_type,
                 time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename StateType> void adjust_size(const StateType &);
  const coef_type & coef_a(void) const;
  const coef_type & coef_b(void) const;
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  template<typename System, typename StateIn, typename StateOut>
    void do_step_impl(System, const StateIn &, time_type, StateOut &,
                      time_type, boost::mpl::true_);
  template<typename System, typename StateIn, typename StateOut>
    void do_step_impl(System, const StateIn &, time_type, StateOut &,
                      time_type, boost::mpl::false_);
  template<typename StateIn> bool resize_dqdt(const StateIn &);
```

```
  template<typename StateIn> bool resize_dpdt(const StateIn &);

  // public data members
  static const size_t num_of_stages;
  static const order_type order_value;
};
```

Description

This class is the base class for the symplectic Runge-Kutta-Nystroem steppers. Symplectic steppers are usually used to solve Hamiltonian systems and they conserve the phase space volume, see en.wikipedia.org/wiki/Symplectic_integrator. Furthermore, the energy is conserved in average. In detail this class of steppers can be used to solve separable Hamiltonian systems which can be written in the form H(q,p) = H1(p) + H2(q). q is usually called the coordinate, while p is the momentum. The equations of motion are dq/dt = dH1/dp, dp/dt = -dH2/dq.

ToDo : add formula for solver and explanation of the coefficients

symplectic_nystroem_stepper_base uses odeints algebra and operation system. Step size and error estimation are not provided for this class of solvers. It derives from algebra_stepper_base. Several `do_step` variants are provided:

- `do_step( sys , x , t , dt )` - The classical `do_step` method. The sys can be either a pair of function objects for the coordinate or the momentum part or one function object for the momentum part. `x` is a pair of coordinate and momentum. The state is updated in-place.
- `do_step( sys , q , p , t , dt )` - This method is similar to the method above with the difference that the coordinate and the momentum are passed explicitly and not packed into a pair.
- `do_step( sys , x_in , t , x_out , dt )` - This method transforms the state out-of-place. `x_in` and `x_out` are here pairs of coordinate and momentum.

Template Parameters

1. ```
   size_t NumOfStages
   ```

   Number of stages.

2. ```
   unsigned short Order
   ```

   The order of the stepper.

3. ```
   typename Coor
   ```

   The type representing the coordinates q.

4. ```
   typename Momentum
   ```

   The type representing the coordinates p.

5. ```
   typename Value
   ```

   The basic value type. Should be something like float, double or a high-precision type.

6. ```
   typename CoorDeriv
   ```

   The type representing the time derivative of the coordinate dq/dt.

7.
```
typename MomentumDeriv
```

8.
```
typename Time
```

The type representing the time t.

9.
```
typename Algebra
```

The algebra.

10.
```
typename Operations
```

The operations.

11.
```
typename Resizer
```

The resizer policy.

`symplectic_nystroem_stepper_base` public construct/copy/destruct

1.
```
symplectic_nystroem_stepper_base(const coef_type & coef_a,
                                 const coef_type & coef_b,
                                 const algebra_type & algebra = algebra_type());
```

Constructs a `symplectic_nystroem_stepper_base` class. The parameters of the specific Nystroem method and the algebra have to be passed.

Parameters:

`algebra`	A copy of algebra is made and stored inside `explicit_stepper_base`.
`coef_a`	The coefficients a.
`coef_b`	The coefficients b.

`symplectic_nystroem_stepper_base` public member functions

1.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper.

2.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & state, time_type t,
               time_type dt);
```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated in-place.

Note

boost::ref or std::ref can be used for the system as well as for the state. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , make_pair( std::ref( q ) , std::ref( p ) ) , t , dt )`.

This method solves the forwarding problem.

Parameters:

`dt`	The time step.
`state`	The state of the ODE. It is a pair of Coor and Momentum. The state is updated in-place, therefore, the new value of the state will be written into this variable.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

3.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & state, time_type t, time_type dt);
```

Same function as above. It differs only in a different const specifier in order to solve the forwarding problem, can be used with Boost.Range.

4.
```
template<typename System, typename CoorInOut, typename MomentumInOut>
  void do_step(System system, CoorInOut & q, MomentumInOut & p, time_type t,
               time_type dt);
```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated in-place.

Note

boost::ref or std::ref can be used for the system. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , q , p , t , dt )`.

This method solves the forwarding problem.

Parameters:

`dt`	The time step.
`p`	The momentum of the ODE. It is updated in-place. Therefore, the new value of the momentum will be written info this variable.
`q`	The coordinate of the ODE. It is updated in-place. Therefore, the new value of the coordinate will be written into this variable.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

5.
```
template<typename System, typename CoorInOut, typename MomentumInOut>
  void do_step(System system, const CoorInOut & q, const MomentumInOut & p,
               time_type t, time_type dt);
```

Same function as do_step(system , q , p , t , dt). It differs only in a different const specifier in order to solve the forwarding problem, can be called with Boost.Range.

6.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated out-of-place.

Note

boost::ref or std::ref can be used for the system. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , x_in , t , x_out , dt )`.

This method NOT solve the forwarding problem.

Parameters:

`dt`	The time step.
`in`	The state of the ODE, which is a pair of coordinate and momentum. The state is updated out-of-place, therefore the new value is written into out
`out`	The new state of the ODE.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

7.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

8.
```
const coef_type & coef_a(void) const;
```

Returns the coefficients a.

9.
```
const coef_type & coef_b(void) const;
```

Returns the coefficients b.

10.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

11.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`symplectic_nystroem_stepper_base` private member functions

1.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step_impl(System system, const StateIn & in, time_type t,
                    StateOut & out, time_type dt, boost::mpl::true_);
```

2.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step_impl(System system, const StateIn & in, time_type,
                    StateOut & out, time_type dt, boost::mpl::false_);
```

3.
```
template<typename StateIn> bool resize_dqdt(const StateIn & x);
```

4.
```
template<typename StateIn> bool resize_dpdt(const StateIn & x);
```

Header <boost/numeric/odeint/stepper/bulirsch_stoer.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class bulirsch_stoer;
    }
  }
}
```

Class template bulirsch_stoer

boost::numeric::odeint::bulirsch_stoer — The Bulirsch-Stoer algorithm.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/bulirsch_stoer.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class bulirsch_stoer {
public:
  // types
  typedef State      state_type;
  typedef Value      value_type;
  typedef Deriv      deriv_type;
  typedef Time       time_type;
  typedef Algebra    algebra_type;
  typedef Operations operations_type;
  typedef Resizer    resizer_type;

  // construct/copy/destruct
  bulirsch_stoer(value_type = 1E-6, value_type = 1E-6, value_type = 1.0,
                 value_type = 1.0);

  // public member functions
  template<typename System, typename StateInOut>
    controlled_step_result
    try_step(System, StateInOut &, time_type &, time_type &);
  template<typename System, typename StateInOut>
    controlled_step_result
    try_step(System, const StateInOut &, time_type &, time_type &);
  template<typename System, typename StateInOut, typename DerivIn>
    controlled_step_result
    try_step(System, StateInOut &, const DerivIn &, time_type &, time_type &);
  template<typename System, typename StateIn, typename StateOut>
    boost::disable_if< boost::is_same< StateIn, time_type >, controlled_step_result >::type
    try_step(System, const StateIn &, time_type &, StateOut &, time_type &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    controlled_step_result
    try_step(System, const StateIn &, const DerivIn &, time_type &,
             StateOut &, time_type &);
  void reset();
  template<typename StateIn> void adjust_size(const StateIn &);

  // private member functions
  template<typename StateIn> bool resize_m_dxdt(const StateIn &);
  template<typename StateIn> bool resize_m_xnew(const StateIn &);
  template<typename StateIn> bool resize_impl(const StateIn &);
  template<typename System, typename StateInOut>
    controlled_step_result
    try_step_v1(System, StateInOut &, time_type &, time_type &);
  template<typename StateInOut>
    void extrapolate(size_t, state_table_type &, const value_matrix &,
                     StateInOut &);
  time_type calc_h_opt(time_type, value_type, size_t) const;
  controlled_step_result
  set_k_opt(size_t, const inv_time_vector &, const time_vector &, time_type &);
```

```
  bool in_convergence_window(size_t) const;
  bool should_reject(value_type, size_t) const;

  // public data members
  static const size_t m_k_max;
};
```

Description

The Bulirsch-Stoer is a controlled stepper that adjusts both step size and order of the method. The algorithm uses the modified midpoint and a polynomial extrapolation compute the solution.

Template Parameters

1. ```
   typename State
   ```

   The state type.

2. ```
   typename Value = double
   ```

   The value type.

3. ```
   typename Deriv = State
   ```

   The type representing the time derivative of the state.

4. ```
   typename Time = Value
   ```

   The time representing the independent variable - the time.

5. ```
   typename Algebra = range_algebra
   ```

   The algebra type.

6. ```
   typename Operations = default_operations
   ```

   The operations type.

7. ```
   typename Resizer = initially_resizer
   ```

   The resizer policy type.

`bulirsch_stoer` public construct/copy/destruct

1. ```
   bulirsch_stoer(value_type eps_abs = 1E-6, value_type eps_rel = 1E-6,
                  value_type factor_x = 1.0, value_type factor_dxdt = 1.0);
   ```

   Constructs the `bulirsch_stoer` class, including initialization of the error bounds.

   Parameters:

`eps_abs`	Absolute tolerance level.
`eps_rel`	Relative tolerance level.
`factor_dxdt`	Factor for the weight of the derivative.
`factor_x`	Factor for the weight of the state.

`bulirsch_stoer` public member functions

1.
```
template<typename System, typename StateInOut>
  controlled_step_result
  try_step(System system, StateInOut & x, time_type & t, time_type & dt);
```

Tries to perform one step.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed. Also, the internal order of the stepper is adjusted if required.

Parameters:	`dt`	The step size. Updated.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time. Updated if the step is successful.
	`x`	The state of the ODE which should be solved. Overwritten if the step is successful.
Returns:	success if the step was accepted, fail otherwise.	

2.
```
template<typename System, typename StateInOut>
  controlled_step_result
  try_step(System system, const StateInOut & x, time_type & t, time_type & dt);
```

Second version to solve the forwarding problem, can be used with Boost.Range as StateInOut.

3.
```
template<typename System, typename StateInOut, typename DerivIn>
  controlled_step_result
  try_step(System system, StateInOut & x, const DerivIn & dxdt, time_type & t,
           time_type & dt);
```

Tries to perform one step.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed. Also, the internal order of the stepper is adjusted if required.

Parameters:	`dt`	The step size. Updated.
	`dxdt`	The derivative of state.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time. Updated if the step is successful.
	`x`	The state of the ODE which should be solved. Overwritten if the step is successful.
Returns:	success if the step was accepted, fail otherwise.	

4.
```
template<typename System, typename StateIn, typename StateOut>
  boost::disable_if< boost::is_same< StateIn, time_type >, controlled_step_result >::type
  try_step(System system, const StateIn & in, time_type & t, StateOut & out,
           time_type & dt);
```

Tries to perform one step.

Note

This method is disabled if state_type=time_type to avoid ambiguity.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed. Also, the internal order of the stepper is adjusted if required.

Parameters:	`dt`	The step size. Updated.
	`in`	The state of the ODE which should be solved.
	`out`	Used to store the result of the step.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time. Updated if the step is successful.
Returns:	success if the step was accepted, fail otherwise.	

5.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  controlled_step_result
  try_step(System system, const StateIn & in, const DerivIn & dxdt,
           time_type & t, StateOut & out, time_type & dt);
```

Tries to perform one step.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed. Also, the internal order of the stepper is adjusted if required.

Parameters:	`dt`	The step size. Updated.
	`dxdt`	The derivative of state.
	`in`	The state of the ODE which should be solved.
	`out`	Used to store the result of the step.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time. Updated if the step is successful.
Returns:	success if the step was accepted, fail otherwise.	

6.
```
void reset();
```

Resets the internal state of the stepper.

7.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters:	`x`	A state from which the size of the temporaries to be resized is deduced.

`bulirsch_stoer` private member functions

1.
```
template<typename StateIn> bool resize_m_dxdt(const StateIn & x);
```

2.
```
template<typename StateIn> bool resize_m_xnew(const StateIn & x);
```

3.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

4.
```
template<typename System, typename StateInOut>
  controlled_step_result
  try_step_v1(System system, StateInOut & x, time_type & t, time_type & dt);
```

5.
```
template<typename StateInOut>
  void extrapolate(size_t k, state_table_type & table,
                   const value_matrix & coeff, StateInOut & xest);
```

6.
```
time_type calc_h_opt(time_type h, value_type error, size_t k) const;
```

7.
```
controlled_step_result
set_k_opt(size_t k, const inv_time_vector & work, const time_vector & h_opt,
          time_type & dt);
```

8.
```
bool in_convergence_window(size_t k) const;
```

9.
```
bool should_reject(value_type error, size_t k) const;
```

Header <boost/numeric/odeint/stepper/bulirsch_stoer_dense_out.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class bulirsch_stoer_dense_out;
    }
  }
}
```

Class template bulirsch_stoer_dense_out

boost::numeric::odeint::bulirsch_stoer_dense_out — The Bulirsch-Stoer algorithm.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/bulirsch_stoer_dense_out.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class bulirsch_stoer_dense_out {
public:
  // types
  typedef State                    state_type;
  typedef Value                    value_type;
  typedef Deriv                    deriv_type;
  typedef Time                     time_type;
  typedef Algebra                  algebra_type;
  typedef Operations               operations_type;
  typedef Resizer                  resizer_type;
  typedef dense_output_stepper_tag stepper_category;

  // construct/copy/destruct
  bulirsch_stoer_dense_out(value_type = 1E-6, value_type = 1E-6,
                           value_type = 1.0, value_type = 1.0, bool = false);

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename DerivOut>
    controlled_step_result
    try_step(System, const StateIn &, const DerivIn &, time_type &,
             StateOut &, DerivOut &, time_type &);
  template<typename StateType>
    void initialize(const StateType &, const time_type &, const time_type &);
  template<typename System> std::pair< time_type, time_type > do_step(System);
  template<typename StateOut> void calc_state(time_type, StateOut &) const;
  const state_type & current_state(void) const;
  time_type current_time(void) const;
  const state_type & previous_state(void) const;
  time_type previous_time(void) const;
  time_type current_time_step(void) const;
  void reset();
  template<typename StateIn> void adjust_size(const StateIn &);

  // private member functions
  template<typename StateInOut, typename StateVector>
    void extrapolate(size_t, StateVector &, const value_matrix &,
                     StateInOut &, size_t = 0);
  template<typename StateVector>
    void extrapolate_dense_out(size_t, StateVector &, const value_matrix &,
                               size_t = 0);
  time_type calc_h_opt(time_type, value_type, size_t) const;
  bool in_convergence_window(size_t) const;
  bool should_reject(value_type, size_t) const;
  template<typename StateIn1, typename DerivIn1, typename StateIn2,
           typename DerivIn2>
    value_type prepare_dense_output(int, const StateIn1 &, const DerivIn1 &,
                                    const StateIn2 &, const DerivIn2 &,
                                    time_type);
  template<typename DerivIn>
    void calculate_finite_difference(size_t, size_t, value_type,
                                     const DerivIn &);
  template<typename StateOut>
    void do_interpolation(time_type, StateOut &) const;
```

```
  template<typename StateIn> bool resize_impl(const StateIn &);
  state_type & get_current_state(void);
  const state_type & get_current_state(void) const;
  state_type & get_old_state(void);
  const state_type & get_old_state(void) const;
  deriv_type & get_current_deriv(void);
  const deriv_type & get_current_deriv(void) const;
  deriv_type & get_old_deriv(void);
  const deriv_type & get_old_deriv(void) const;
  void toggle_current_state(void);

  // public data members
  static const size_t m_k_max;
};
```

Description

The Bulirsch-Stoer is a controlled stepper that adjusts both step size and order of the method. The algorithm uses the modified midpoint and a polynomial extrapolation compute the solution. This class also provides dense output facility.

Template Parameters

1. `typename State`

   The state type.

2. `typename Value = double`

   The value type.

3. `typename Deriv = State`

   The type representing the time derivative of the state.

4. `typename Time = Value`

   The time representing the independent variable - the time.

5. `typename Algebra = range_algebra`

   The algebra type.

6. `typename Operations = default_operations`

   The operations type.

7. `typename Resizer = initially_resizer`

   The resizer policy type.

`bulirsch_stoer_dense_out` public construct/copy/destruct

1.
```
bulirsch_stoer_dense_out(value_type eps_abs = 1E-6, value_type eps_rel = 1E-6,
                         value_type factor_x = 1.0,
                         value_type factor_dxdt = 1.0,
                         bool control_interpolation = false);
```

Constructs the `bulirsch_stoer` class, including initialization of the error bounds.

Parameters:		
	`control_interpolation`	Set true to additionally control the error of the interpolation.
	`eps_abs`	Absolute tolerance level.
	`eps_rel`	Relative tolerance level.
	`factor_dxdt`	Factor for the weight of the derivative.
	`factor_x`	Factor for the weight of the state.

`bulirsch_stoer_dense_out` public member functions

1.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename DerivOut>
  controlled_step_result
  try_step(System system, const StateIn & in, const DerivIn & dxdt,
           time_type & t, StateOut & out, DerivOut & dxdt_new,
           time_type & dt);
```

Tries to perform one step.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed. Also, the internal order of the stepper is adjusted if required.

Parameters:		
	`dt`	The step size. Updated.
	`dxdt`	The derivative of state.
	`in`	The state of the ODE which should be solved.
	`out`	Used to store the result of the step.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time. Updated if the step is successful.
Returns:	success if the step was accepted, fail otherwise.	

2.
```
template<typename StateType>
  void initialize(const StateType & x0, const time_type & t0,
                  const time_type & dt0);
```

Initializes the dense output stepper.

Parameters:		
	`dt0`	The initial time step.
	`t0`	The initial time.
	`x0`	The initial state.

3.
```
template<typename System>
  std::pair< time_type, time_type > do_step(System system);
```

Does one time step. This is the main method that should be used to integrate an ODE with this stepper.

Note

initialize has to be called before using this method to set the initial conditions x,t and the stepsize.

Parameters: `system` The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
Returns: Pair with start and end time of the integration step.

4.
```
template<typename StateOut> void calc_state(time_type t, StateOut & x) const;
```

Calculates the solution at an intermediate point within the last step.

Parameters: `t` The time at which the solution should be calculated, has to be in the current time interval.
`x` The output variable where the result is written into.

5.
```
const state_type & current_state(void) const;
```

Returns the current state of the solution.

Returns: The current state of the solution x(t).

6.
```
time_type current_time(void) const;
```

Returns the current time of the solution.

Returns: The current time of the solution t.

7.
```
const state_type & previous_state(void) const;
```

Returns the last state of the solution.

Returns: The last state of the solution x(t-dt).

8.
```
time_type previous_time(void) const;
```

Returns the last time of the solution.

Returns: The last time of the solution t-dt.

9.
```
time_type current_time_step(void) const;
```

Returns the current step size.

Returns: The current step size.

10.
```
void reset();
```

Resets the internal state of the stepper.

11.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: x A state from which the size of the temporaries to be resized is deduced.

`bulirsch_stoer_dense_out` private member functions

1.
```
template<typename StateInOut, typename StateVector>
  void extrapolate(size_t k, StateVector & table, const value_matrix & coeff,
                   StateInOut & xest, size_t order_start_index = 0);
```

2.
```
template<typename StateVector>
  void extrapolate_dense_out(size_t k, StateVector & table,
                             const value_matrix & coeff,
                             size_t order_start_index = 0);
```

3.
```
time_type calc_h_opt(time_type h, value_type error, size_t k) const;
```

4.
```
bool in_convergence_window(size_t k) const;
```

5.
```
bool should_reject(value_type error, size_t k) const;
```

6.
```
template<typename StateIn1, typename DerivIn1, typename StateIn2,
         typename DerivIn2>
  value_type prepare_dense_output(int k, const StateIn1 & x_start,
                                  const DerivIn1 & dxdt_start,
                                  const StateIn2 &, const DerivIn2 &,
                                  time_type dt);
```

7.
```
template<typename DerivIn>
  void calculate_finite_difference(size_t j, size_t kappa, value_type fac,
                                   const DerivIn & dxdt);
```

8.
```
template<typename StateOut>
  void do_interpolation(time_type t, StateOut & out) const;
```

9.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

10.
```
state_type & get_current_state(void);
```

11.
```
const state_type & get_current_state(void) const;
```

12.
```
state_type & get_old_state(void);
```

```
13. const state_type & get_old_state(void) const;

14. deriv_type & get_current_deriv(void);

15. const deriv_type & get_current_deriv(void) const;

16. deriv_type & get_old_deriv(void);

17. const deriv_type & get_old_deriv(void) const;

18. void toggle_current_state(void);
```

Header <boost/numeric/odeint/stepper/controlled_runge_kutta.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Value, typename Algebra = range_algebra,
               typename Operations = default_operations>
        class default_error_checker;
      template<typename ErrorStepper,
               typename ErrorChecker = default_error_checker< typename ErrorStep↵
per::value_type ,typename ErrorStepper::algebra_type ,typename ErrorStepper::operations_type >,
               typename Resizer = typename ErrorStepper::resizer_type,
               typename ErrorStepperCategory = typename ErrorStepper::stepper_category>
        class controlled_runge_kutta;

      template<typename ErrorStepper, typename ErrorChecker, typename Resizer>
        class controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_step↵
per_tag>;
      template<typename ErrorStepper, typename ErrorChecker, typename Resizer>
        class controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_step↵
per_fsal_tag>;
    }
  }
}
```

Class template default_error_checker

boost::numeric::odeint::default_error_checker — The default error checker to be used with Runge-Kutta error steppers.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/controlled_runge_kutta.hpp>

template<typename Value, typename Algebra = range_algebra,
         typename Operations = default_operations>
class default_error_checker {
public:
  // types
  typedef Value      value_type;
  typedef Algebra    algebra_type;
  typedef Operations operations_type;

  // construct/copy/destruct
  default_error_checker(value_type = static_cast< value_type >(1.0e-6),
                        value_type = static_cast< value_type >(1.0e-6),
                        value_type = static_cast< value_type >(1),
                        value_type = static_cast< value_type >(1));

  // public member functions
  template<typename State, typename Deriv, typename Err, typename Time>
    value_type error(const State &, const Deriv &, Err &, Time) const;
  template<typename State, typename Deriv, typename Err, typename Time>
    value_type error(algebra_type &, const State &, const Deriv &, Err &,
                     Time) const;
};
```

Description

This class provides the default mechanism to compare the error estimates reported by Runge-Kutta error steppers with user defined error bounds. It is used by the controlled_runge_kutta steppers.

Template Parameters

1. ```
   typename Value
   ```

   The value type.

2. ```
   typename Algebra = range_algebra
   ```

   The algebra type.

3. ```
   typename Operations = default_operations
   ```

   The operations type.

`default_error_checker` public construct/copy/destruct

1. ```
   default_error_checker(value_type eps_abs = static_cast< value_type >(1.0e-6),
                         value_type eps_rel = static_cast< value_type >(1.0e-6),
                         value_type a_x = static_cast< value_type >(1),
                         value_type a_dxdt = static_cast< value_type >(1));
   ```

`default_error_checker` public member functions

1.
```
template<typename State, typename Deriv, typename Err, typename Time>
  value_type error(const State & x_old, const Deriv & dxdt_old, Err & x_err,
                   Time dt) const;
```

2.
```
template<typename State, typename Deriv, typename Err, typename Time>
  value_type error(algebra_type & algebra, const State & x_old,
                   const Deriv & dxdt_old, Err & x_err, Time dt) const;
```

Class template controlled_runge_kutta

boost::numeric::odeint::controlled_runge_kutta

Synopsis

```
// In header: <boost/numeric/odeint/stepper/controlled_runge_kutta.hpp>

template<typename ErrorStepper,
         typename ErrorChecker = default_error_checker< typename ErrorStepper::value_type ,type↵
name ErrorStepper::algebra_type ,typename ErrorStepper::operations_type >,
          typename Resizer = typename ErrorStepper::resizer_type,
          typename ErrorStepperCategory = typename ErrorStepper::stepper_category>
class controlled_runge_kutta {
};
```

Description

Specializations

- Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>
- Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>

Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>

boost::numeric::odeint::controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag> — Implements step size control for Runge-Kutta steppers with error estimation.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/controlled_runge_kutta.hpp>

template<typename ErrorStepper, typename ErrorChecker, typename Resizer>
class controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag> {
public:
  // types
  typedef ErrorStepper                    stepper_type;
  typedef stepper_type::state_type        state_type;
  typedef stepper_type::value_type        value_type;
  typedef stepper_type::deriv_type        deriv_type;
  typedef stepper_type::time_type         time_type;
  typedef stepper_type::algebra_type      algebra_type;
  typedef stepper_type::operations_type   operations_type;
  typedef Resizer                         resizer_type;
  typedef ErrorChecker                    error_checker_type;
  typedef explicit_controlled_stepper_tag stepper_category;

  // construct/copy/destruct
  controlled_runge_kutta(const error_checker_type & = error_checker_type(),
                         const stepper_type & = stepper_type());

  // public member functions
  template<typename System, typename StateInOut>
    controlled_step_result
    try_step(System, StateInOut &, time_type &, time_type &);
  template<typename System, typename StateInOut>
    controlled_step_result
    try_step(System, const StateInOut &, time_type &, time_type &);
  template<typename System, typename StateInOut, typename DerivIn>
    controlled_step_result
    try_step(System, StateInOut &, const DerivIn &, time_type &, time_type &);
  template<typename System, typename StateIn, typename StateOut>
    boost::disable_if< boost::is_same< StateIn, time_type >, controlled_step_result >::type
    try_step(System, const StateIn &, time_type &, StateOut &, time_type &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    controlled_step_result
    try_step(System, const StateIn &, const DerivIn &, time_type &,
             StateOut &, time_type &);
  value_type last_error(void) const;
  template<typename StateType> void adjust_size(const StateType &);
  stepper_type & stepper(void);
  const stepper_type & stepper(void) const;

  // private member functions
  template<typename System, typename StateInOut>
    controlled_step_result
    try_step_v1(System, StateInOut &, time_type &, time_type &);
  template<typename StateIn> bool resize_m_xerr_impl(const StateIn &);
  template<typename StateIn> bool resize_m_dxdt_impl(const StateIn &);
  template<typename StateIn> bool resize_m_xnew_impl(const StateIn &);
};
```

Description

This class implements the step size control for standard Runge-Kutta steppers with error estimation.

Template Parameters

1.
```
typename ErrorStepper
```

The stepper type with error estimation, has to fulfill the ErrorStepper concept.

2.
```
typename ErrorChecker
```

The error checker

3.
```
typename Resizer
```

The resizer policy type.

`controlled_runge_kutta` public construct/copy/destruct

1.
```
controlled_runge_kutta(const error_checker_type & error_checker = error_checker_type(),
                       const stepper_type & stepper = stepper_type());
```

Constructs the controlled Runge-Kutta stepper.

Parameters:	`error_checker`	An instance of the error checker.
	`stepper`	An instance of the underlying stepper.

`controlled_runge_kutta` public member functions

1.
```
template<typename System, typename StateInOut>
  controlled_step_result
  try_step(System system, StateInOut & x, time_type & t, time_type & dt);
```

Tries to perform one step.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed.

Parameters:	`dt`	The step size. Updated.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time. Updated if the step is successful.
	`x`	The state of the ODE which should be solved. Overwritten if the step is successful.
Returns:	success if the step was accepted, fail otherwise.	

2.
```
template<typename System, typename StateInOut>
  controlled_step_result
  try_step(System system, const StateInOut & x, time_type & t, time_type & dt);
```

Tries to perform one step. Solves the forwarding problem and allows for using boost range as state_type.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed.

Parameters:	`dt`	The step size. Updated.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time. Updated if the step is successful.

	`x`	The state of the ODE which should be solved. Overwritten if the step is successful. Can be a boost range.
Returns:	success if the step was accepted, fail otherwise.	

3.
```
template<typename System, typename StateInOut, typename DerivIn>
  controlled_step_result
  try_step(System system, StateInOut & x, const DerivIn & dxdt, time_type & t,
           time_type & dt);
```

Tries to perform one step.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed.

Parameters:	`dt`	The step size. Updated.
	`dxdt`	The derivative of state.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time. Updated if the step is successful.
	`x`	The state of the ODE which should be solved. Overwritten if the step is successful.
Returns:	success if the step was accepted, fail otherwise.	

4.
```
template<typename System, typename StateIn, typename StateOut>
  boost::disable_if< boost::is_same< StateIn, time_type >, controlled_step_result >::type
  try_step(System system, const StateIn & in, time_type & t, StateOut & out,
           time_type & dt);
```

Tries to perform one step.

Note

This method is disabled if state_type=time_type to avoid ambiguity.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed.

Parameters:	`dt`	The step size. Updated.
	`in`	The state of the ODE which should be solved.
	`out`	Used to store the result of the step.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time. Updated if the step is successful.
Returns:	success if the step was accepted, fail otherwise.	

5.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  controlled_step_result
  try_step(System system, const StateIn & in, const DerivIn & dxdt,
           time_type & t, StateOut & out, time_type & dt);
```

Tries to perform one step.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed.

Parameters:	`dt`	The step size. Updated.

`dxdt`	The derivative of state.
`in`	The state of the ODE which should be solved.
`out`	Used to store the result of the step.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time. Updated if the step is successful.

Returns: success if the step was accepted, fail otherwise.

6.
```
value_type last_error(void) const;
```

Returns the error of the last step.

returns The last error of the step.

7.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

8.
```
stepper_type & stepper(void);
```

Returns the instance of the underlying stepper.

Returns: The instance of the underlying stepper.

9.
```
const stepper_type & stepper(void) const;
```

Returns the instance of the underlying stepper.

Returns: The instance of the underlying stepper.

`controlled_runge_kutta` private member functions

1.
```
template<typename System, typename StateInOut>
  controlled_step_result
  try_step_v1(System system, StateInOut & x, time_type & t, time_type & dt);
```

2.
```
template<typename StateIn> bool resize_m_xerr_impl(const StateIn & x);
```

3.
```
template<typename StateIn> bool resize_m_dxdt_impl(const StateIn & x);
```

4.
```
template<typename StateIn> bool resize_m_xnew_impl(const StateIn & x);
```

Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>

boost::numeric::odeint::controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag> — Implements step size control for Runge-Kutta FSAL steppers with error estimation.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/controlled_runge_kutta.hpp>

template<typename ErrorStepper, typename ErrorChecker, typename Resizer>
class controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_step↵
per_fsal_tag> {
public:
  // types
  typedef ErrorStepper                          stepper_type;
  typedef stepper_type::state_type              state_type;
  typedef stepper_type::value_type              value_type;
  typedef stepper_type::deriv_type              deriv_type;
  typedef stepper_type::time_type               time_type;
  typedef stepper_type::algebra_type            algebra_type;
  typedef stepper_type::operations_type         operations_type;
  typedef Resizer                               resizer_type;
  typedef ErrorChecker                          error_checker_type;
  typedef explicit_controlled_stepper_fsal_tag stepper_category;

  // construct/copy/destruct
  controlled_runge_kutta(const error_checker_type & = error_checker_type(),
                         const stepper_type & = stepper_type());

  // public member functions
  template<typename System, typename StateInOut>
    controlled_step_result
    try_step(System, StateInOut &, time_type &, time_type &);
  template<typename System, typename StateInOut>
    controlled_step_result
    try_step(System, const StateInOut &, time_type &, time_type &);
  template<typename System, typename StateIn, typename StateOut>
    boost::disable_if< boost::is_same< StateIn, time_type >, controlled_step_result >::type
    try_step(System, const StateIn &, time_type &, StateOut &, time_type &);
  template<typename System, typename StateInOut, typename DerivInOut>
    controlled_step_result
    try_step(System, StateInOut &, DerivInOut &, time_type &, time_type &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename DerivOut>
    controlled_step_result
    try_step(System, const StateIn &, const DerivIn &, time_type &,
             StateOut &, DerivOut &, time_type &);
  void reset(void);
  template<typename DerivIn> void initialize(const DerivIn &);
  template<typename System, typename StateIn>
    void initialize(System, const StateIn &, time_type);
  bool is_initialized(void) const;
  template<typename StateType> void adjust_size(const StateType &);
  stepper_type & stepper(void);
  const stepper_type & stepper(void) const;

  // private member functions
  template<typename StateIn> bool resize_m_xerr_impl(const StateIn &);
  template<typename StateIn> bool resize_m_dxdt_impl(const StateIn &);
  template<typename StateIn> bool resize_m_dxdt_new_impl(const StateIn &);
  template<typename StateIn> bool resize_m_xnew_impl(const StateIn &);
  template<typename System, typename StateInOut>
    controlled_step_result
    try_step_v1(System, StateInOut &, time_type &, time_type &);
};
```

Description

This class implements the step size control for FSAL Runge-Kutta steppers with error estimation.

Template Parameters

1. ```
   typename ErrorStepper
   ```

   The stepper type with error estimation, has to fulfill the ErrorStepper concept.

2. ```
   typename ErrorChecker
   ```

   The error checker

3. ```
   typename Resizer
   ```

   The resizer policy type.

`controlled_runge_kutta` public construct/copy/destruct

1. ```
   controlled_runge_kutta(const error_checker_type & error_checker = error_checker_type(),
                          const stepper_type & stepper = stepper_type());
   ```

   Constructs the controlled Runge-Kutta stepper.

Parameters:	`error_checker`	An instance of the error checker.
	`stepper`	An instance of the underlying stepper.

`controlled_runge_kutta` public member functions

1. ```
   template<typename System, typename StateInOut>
     controlled_step_result
     try_step(System system, StateInOut & x, time_type & t, time_type & dt);
   ```

   Tries to perform one step.

   This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed.

Parameters:	`dt`	The step size. Updated.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time. Updated if the step is successful.
	`x`	The state of the ODE which should be solved. Overwritten if the step is successful.
Returns:	success if the step was accepted, fail otherwise.	

2. ```
   template<typename System, typename StateInOut>
     controlled_step_result
     try_step(System system, const StateInOut & x, time_type & t, time_type & dt);
   ```

   Tries to perform one step. Solves the forwarding problem and allows for using boost range as state_type.

   This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed.

Parameters:
- `dt` The step size. Updated.
- `system` The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
- `t` The value of the time. Updated if the step is successful.
- `x` The state of the ODE which should be solved. Overwritten if the step is successful. Can be a boost range.

Returns: success if the step was accepted, fail otherwise.

3.
```
template<typename System, typename StateIn, typename StateOut>
  boost::disable_if< boost::is_same< StateIn, time_type >, controlled_step_result >::type
  try_step(System system, const StateIn & in, time_type & t, StateOut & out,
           time_type & dt);
```

Tries to perform one step.

> **Note**
>
> This method is disabled if state_type=time_type to avoid ambiguity.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed.

Parameters:
- `dt` The step size. Updated.
- `in` The state of the ODE which should be solved.
- `out` Used to store the result of the step.
- `system` The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
- `t` The value of the time. Updated if the step is successful.

Returns: success if the step was accepted, fail otherwise.

4.
```
template<typename System, typename StateInOut, typename DerivInOut>
  controlled_step_result
  try_step(System system, StateInOut & x, DerivInOut & dxdt, time_type & t,
           time_type & dt);
```

Tries to perform one step.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed.

Parameters:
- `dt` The step size. Updated.
- `dxdt` The derivative of state.
- `system` The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
- `t` The value of the time. Updated if the step is successful.
- `x` The state of the ODE which should be solved. Overwritten if the step is successful.

Returns: success if the step was accepted, fail otherwise.

5.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename DerivOut>
  controlled_step_result
  try_step(System system, const StateIn & in, const DerivIn & dxdt_in,
           time_type & t, StateOut & out, DerivOut & dxdt_out,
           time_type & dt);
```

Tries to perform one step.

This method tries to do one step with step size dt. If the error estimate is to large, the step is rejected and the method returns fail and the step size dt is reduced. If the error estimate is acceptably small, the step is performed, success is returned and dt might be increased to make the steps as large as possible. This method also updates t if a step is performed.

Parameters:
- `dt` The step size. Updated.
- `in` The state of the ODE which should be solved.
- `out` Used to store the result of the step.
- `system` The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
- `t` The value of the time. Updated if the step is successful.

Returns: success if the step was accepted, fail otherwise.

6.
```
void reset(void);
```

Resets the internal state of the underlying FSAL stepper.

7.
```
template<typename DerivIn> void initialize(const DerivIn & deriv);
```

Initializes the internal state storing an internal copy of the derivative.

Parameters:
- `deriv` The initial derivative of the ODE.

8.
```
template<typename System, typename StateIn>
  void initialize(System system, const StateIn & x, time_type t);
```

Initializes the internal state storing an internal copy of the derivative.

Parameters:
- `system` The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
- `t` The initial time.
- `x` The initial state of the ODE which should be solved.

9.
```
bool is_initialized(void) const;
```

Returns true if the stepper has been initialized, false otherwise.

Returns: true, if the stepper has been initialized, false otherwise.

10.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters:
- `x` A state from which the size of the temporaries to be resized is deduced.

11.
```
stepper_type & stepper(void);
```

Returns the instance of the underlying stepper.

Returns: The instance of the underlying stepper.

12.
```
const stepper_type & stepper(void) const;
```

Returns the instance of the underlying stepper.

Returns: The instance of the underlying stepper.

`controlled_runge_kutta` private member functions

1. ```
   template<typename StateIn> bool resize_m_xerr_impl(const StateIn & x);
   ```

2. ```
   template<typename StateIn> bool resize_m_dxdt_impl(const StateIn & x);
   ```

3. ```
   template<typename StateIn> bool resize_m_dxdt_new_impl(const StateIn & x);
   ```

4. ```
   template<typename StateIn> bool resize_m_xnew_impl(const StateIn & x);
   ```

5. ```
   template<typename System, typename StateInOut>
     controlled_step_result
     try_step_v1(System system, StateInOut & x, time_type & t, time_type & dt);
   ```

Header <boost/numeric/odeint/stepper/controlled_step_result.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {

      // Enum representing the return values of the controlled steppers.
      enum controlled_step_result { success, fail };
    }
  }
}
```

Header <boost/numeric/odeint/stepper/dense_output_runge_kutta.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Stepper,
               typename StepperCategory = typename Stepper::stepper_category>
        class dense_output_runge_kutta;

      template<typename Stepper>
        class dense_output_runge_kutta<Stepper, stepper_tag>;
      template<typename Stepper>
        class dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>;
    }
  }
}
```

Class template dense_output_runge_kutta

boost::numeric::odeint::dense_output_runge_kutta
```

Synopsis

```
// In header: <boost/numeric/odeint/stepper/dense_output_runge_kutta.hpp>

template<typename Stepper,
         typename StepperCategory = typename Stepper::stepper_category>
class dense_output_runge_kutta {
};
```

Description

Specializations

- Class template dense_output_runge_kutta<Stepper, stepper_tag>
- Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>

Class template dense_output_runge_kutta<Stepper, stepper_tag>

boost::numeric::odeint::dense_output_runge_kutta<Stepper, stepper_tag> — The class representing dense-output Runge-Kutta steppers.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/dense_output_runge_kutta.hpp>

template<typename Stepper>
class dense_output_runge_kutta<Stepper, stepper_tag> {
public:
  // types
  typedef Stepper                              stepper_type;
  typedef stepper_type::state_type             state_type;
  typedef stepper_type::wrapped_state_type     wrapped_state_type;
  typedef stepper_type::value_type             value_type;
  typedef stepper_type::deriv_type             deriv_type;
  typedef stepper_type::wrapped_deriv_type     wrapped_deriv_type;
  typedef stepper_type::time_type              time_type;
  typedef stepper_type::algebra_type           algebra_type;
  typedef stepper_type::operations_type        operations_type;
  typedef stepper_type::resizer_type           resizer_type;
  typedef dense_output_stepper_tag             stepper_category;
  typedef dense_output_runge_kutta< Stepper > dense_output_stepper_type;

  // construct/copy/destruct
  dense_output_runge_kutta(const stepper_type & = stepper_type());

  // public member functions
  template<typename StateType>
    void initialize(const StateType &, time_type, time_type);
  template<typename System> std::pair< time_type, time_type > do_step(System);
  template<typename StateOut> void calc_state(time_type, StateOut &) const;
  template<typename StateOut>
    void calc_state(time_type, const StateOut &) const;
  template<typename StateType> void adjust_size(const StateType &);
  const state_type & current_state(void) const;
  time_type current_time(void) const;
  const state_type & previous_state(void) const;
  time_type previous_time(void) const;

  // private member functions
  state_type & get_current_state(void);
  const state_type & get_current_state(void) const;
  state_type & get_old_state(void);
  const state_type & get_old_state(void) const;
  void toggle_current_state(void);
  template<typename StateIn> bool resize_impl(const StateIn &);
};
```

Description

Note

In this stepper, the initialize method has to be called before using the do_step method.

The dense-output functionality allows to interpolate the solution between subsequent integration points using intermediate results obtained during the computation. This version works based on a normal stepper without step-size control.

Template Parameters

1. `typename Stepper`

The stepper type of the underlying algorithm.

`dense_output_runge_kutta` public construct/copy/destruct

1.
```
dense_output_runge_kutta(const stepper_type & stepper = stepper_type());
```

Constructs the dense_output_runge_kutta class. An instance of the underlying stepper can be provided.

Parameters: `stepper` An instance of the underlying stepper.

`dense_output_runge_kutta` public member functions

1.
```
template<typename StateType>
  void initialize(const StateType & x0, time_type t0, time_type dt0);
```

Initializes the stepper. Has to be called before do_step can be used to set the initial conditions and the step size.

Parameters:
- `dt0` The step size.
- `t0` The initial time, at which the step should be performed.
- `x0` The initial state of the ODE which should be solved.

2.
```
template<typename System>
  std::pair< time_type, time_type > do_step(System system);
```

Does one time step.

Note

initialize has to be called before using this method to set the initial conditions x,t and the stepsize.

Parameters: `system` The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.

Returns: Pair with start and end time of the integration step.

3.
```
template<typename StateOut> void calc_state(time_type t, StateOut & x) const;
```

Calculates the solution at an intermediate point.

Parameters:
- `t` The time at which the solution should be calculated, has to be in the current time interval.
- `x` The output variable where the result is written into.

4.
```
template<typename StateOut>
  void calc_state(time_type t, const StateOut & x) const;
```

Calculates the solution at an intermediate point. Solves the forwarding problem.

Parameters:
- `t` The time at which the solution should be calculated, has to be in the current time interval.
- `x` The output variable where the result is written into, can be a boost range.

5.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

6. ```
   const state_type & current_state(void) const;
   ```

   Returns the current state of the solution.

   Returns: The current state of the solution x(t).

7. ```
   time_type current_time(void) const;
   ```

   Returns the current time of the solution.

   Returns: The current time of the solution t.

8. ```
   const state_type & previous_state(void) const;
   ```

   Returns the last state of the solution.

   Returns: The last state of the solution x(t-dt).

9. ```
   time_type previous_time(void) const;
   ```

   Returns the last time of the solution.

   Returns: The last time of the solution t-dt.

`dense_output_runge_kutta` private member functions

1. ```
   state_type & get_current_state(void);
   ```

2. ```
   const state_type & get_current_state(void) const;
   ```

3. ```
   state_type & get_old_state(void);
   ```

4. ```
   const state_type & get_old_state(void) const;
   ```

5. ```
   void toggle_current_state(void);
   ```

6. ```
   template<typename StateIn> bool resize_impl(const StateIn & x);
   ```

Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>

boost::numeric::odeint::dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag> — The class representing dense-output Runge-Kutta steppers with FSAL property.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/dense_output_runge_kutta.hpp>

template<typename Stepper>
class dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag> {
public:
  // types
  typedef Stepper                                 controlled_stepper_type;
  typedef controlled_stepper_type::stepper_type stepper_type;
  typedef stepper_type::state_type                state_type;
  typedef stepper_type::wrapped_state_type        wrapped_state_type;
  typedef stepper_type::value_type                value_type;
  typedef stepper_type::deriv_type                deriv_type;
  typedef stepper_type::wrapped_deriv_type        wrapped_deriv_type;
  typedef stepper_type::time_type                 time_type;
  typedef stepper_type::algebra_type              algebra_type;
  typedef stepper_type::operations_type           operations_type;
  typedef stepper_type::resizer_type              resizer_type;
  typedef dense_output_stepper_tag                stepper_category;
  typedef dense_output_runge_kutta< Stepper >     dense_output_stepper_type;

  // construct/copy/destruct
  dense_output_runge_kutta(const controlled_stepper_type & = controlled_stepper_type());

  // public member functions
  template<typename StateType>
    void initialize(const StateType &, time_type, time_type);
  template<typename System> std::pair< time_type, time_type > do_step(System);
  template<typename StateOut> void calc_state(time_type, StateOut &) const;
  template<typename StateOut>
    void calc_state(time_type, const StateOut &) const;
  template<typename StateIn> bool resize(const StateIn &);
  template<typename StateType> void adjust_size(const StateType &);
  const state_type & current_state(void) const;
  time_type current_time(void) const;
  const state_type & previous_state(void) const;
  time_type previous_time(void) const;
  time_type current_time_step(void) const;

  // private member functions
  state_type & get_current_state(void);
  const state_type & get_current_state(void) const;
  state_type & get_old_state(void);
  const state_type & get_old_state(void) const;
  deriv_type & get_current_deriv(void);
  const deriv_type & get_current_deriv(void) const;
  deriv_type & get_old_deriv(void);
  const deriv_type & get_old_deriv(void) const;
  void toggle_current_state(void);
};
```

Description

The interface is the same as for dense_output_runge_kutta< Stepper , stepper_tag >. This class provides dense output functionality based on methods with step size controlled

Template Parameters

1. `typename Stepper`

The stepper type of the underlying algorithm.

`dense_output_runge_kutta` public construct/copy/destruct

1.
```
dense_output_runge_kutta(const controlled_stepper_type & stepper = controlled_stepper_type());
```

`dense_output_runge_kutta` public member functions

1.
```
template<typename StateType>
  void initialize(const StateType & x0, time_type t0, time_type dt0);
```

2.
```
template<typename System>
  std::pair< time_type, time_type > do_step(System system);
```

3.
```
template<typename StateOut> void calc_state(time_type t, StateOut & x) const;
```

4.
```
template<typename StateOut>
  void calc_state(time_type t, const StateOut & x) const;
```

5.
```
template<typename StateIn> bool resize(const StateIn & x);
```

6.
```
template<typename StateType> void adjust_size(const StateType & x);
```

7.
```
const state_type & current_state(void) const;
```

8.
```
time_type current_time(void) const;
```

9.
```
const state_type & previous_state(void) const;
```

10.
```
time_type previous_time(void) const;
```

11.
```
time_type current_time_step(void) const;
```

`dense_output_runge_kutta` private member functions

1.
```
state_type & get_current_state(void);
```

2.
```
const state_type & get_current_state(void) const;
```

3. `state_type & get_old_state(void);`

4. `const state_type & get_old_state(void) const;`

5. `deriv_type & get_current_deriv(void);`

6. `const deriv_type & get_current_deriv(void) const;`

7. `deriv_type & get_old_deriv(void);`

8. `const deriv_type & get_old_deriv(void) const;`

9. `void toggle_current_state(void);`

Header <boost/numeric/odeint/stepper/euler.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class euler;
    }
  }
}
```

Class template euler

boost::numeric::odeint::euler — An implementation of the Euler method.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/euler.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class euler : public boost::numeric::odeint::explicit_stepper_base< Stepper, Order, State, Value, ↵
Deriv, Time, Algebra, Operations, Resizer >
{
public:
  // types
  typedef explicit_stepper_base< euler< ... >,... > stepper_base_type;
  typedef stepper_base_type::state_type             state_type;
  typedef stepper_base_type::value_type             value_type;
  typedef stepper_base_type::deriv_type             deriv_type;
  typedef stepper_base_type::time_type              time_type;
  typedef stepper_base_type::algebra_type           algebra_type;
  typedef stepper_base_type::operations_type        operations_type;
  typedef stepper_base_type::resizer_type           resizer_type;

  // construct/copy/destruct
  euler(const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type);
  template<typename StateOut, typename StateIn1, typename StateIn2>
    void calc_state(StateOut &, time_type, const StateIn1 &, time_type,
                    const StateIn2 &, time_type) const;
  template<typename StateType> void adjust_size(const StateType &);
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type);
  template<typename StateIn> void adjust_size(const StateIn &);
  algebra_type & algebra();
  const algebra_type & algebra() const;
};
```

Description

The Euler method is a very simply solver for ordinary differential equations. This method should not be used for real applications. It is only useful for demonstration purposes. Step size control is not provided but trivial continuous output is available.

This class derives from explicit_stepper_base and inherits its interface via CRTP (current recurring template pattern), see explicit_stepper_base

Template Parameters

1. ```
   typename State
   ```

   The state type.

2. ```
   typename Value = double
   ```

   The value type.

3. ```
   typename Deriv = State
   ```

   The type representing the time derivative of the state.

4. ```
   typename Time = Value
   ```

   The time representing the independent variable - the time.

5. ```
   typename Algebra = range_algebra
   ```

   The algebra type.

6. ```
   typename Operations = default_operations
   ```

   The operations type.

7. ```
   typename Resizer = initially_resizer
   ```

   The resizer policy type.

`euler` public construct/copy/destruct

1. ```
   euler(const algebra_type & algebra = algebra_type());
   ```

   Constructs the euler class. This constructor can be used as a default constructor of the algebra has a default constructor.

   Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`euler` public member functions

1. ```
   template<typename System, typename StateIn, typename DerivIn,
            typename StateOut>
     void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                       time_type t, StateOut & out, time_type dt);
   ```

   This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out of place, hence the input is in `in` and the output in `out`. Access to this step functionality is provided by `explicit_stepper_base` and `do_step_impl` should not be called directly.

   Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
   ```

`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

2.
```
template<typename StateOut, typename StateIn1, typename StateIn2>
  void calc_state(StateOut & x, time_type t, const StateIn1 & old_state,
                  time_type t_old, const StateIn2 & current_state,
                  time_type t_new) const;
```

This method is used for continuous output and it calculates the state `x` at a time `t` from the knowledge of two states `old_state` and `current_state` at time points `t_old` and `t_new`.

3.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

4.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper.

5.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

6.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

7.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

8.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

9.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

10.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters:

`x`	A state from which the size of the temporaries to be resized is deduced.

11.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

12.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

Header <boost/numeric/odeint/stepper/explicit_error_generic_rk.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<size_t StageCount, size_t Order, size_t StepperOrder,
               size_t ErrorOrder, typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class explicit_error_generic_rk;
    }
  }
}
```

Class template explicit_error_generic_rk

boost::numeric::odeint::explicit_error_generic_rk — A generic implementation of explicit Runge-Kutta algorithms with error estimation. This class is as a base class for all explicit Runge-Kutta steppers with error estimation.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/explicit_error_generic_rk.hpp>

template<size_t StageCount, size_t Order, size_t StepperOrder,
         size_t ErrorOrder, typename State, typename Value = double,
         typename Deriv = State, typename Time = Value,
         typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class explicit_error_generic_rk : public boost::numeric::odeint::explicit_error_stepper_base< ↵
Stepper, Order, StepperOrder, ErrorOrder, State, Value, Deriv, Time, Algebra, Operations, Res↵
izer >
{
public:
  // types
  typedef explicit_stepper_base< ... >          stepper_base_type;
  typedef stepper_base_type::state_type         state_type;
  typedef stepper_base_type::wrapped_state_type wrapped_state_type;
  typedef stepper_base_type::value_type         value_type;
  typedef stepper_base_type::deriv_type         deriv_type;
  typedef stepper_base_type::wrapped_deriv_type wrapped_deriv_type;
  typedef stepper_base_type::time_type          time_type;
  typedef stepper_base_type::algebra_type       algebra_type;
  typedef stepper_base_type::operations_type    operations_type;
  typedef stepper_base_type::resizer_type       resizer_type;
  typedef unspecified                           rk_algorithm_type;
  typedef rk_algorithm_type::coef_a_type        coef_a_type;
  typedef rk_algorithm_type::coef_b_type        coef_b_type;
  typedef rk_algorithm_type::coef_c_type        coef_c_type;

  // construct/copy/destruct
  explicit_error_generic_rk(const coef_a_type &, const coef_b_type &,
                            const coef_b_type &, const coef_c_type &,
                            const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename Err>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type, Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type);
  template<typename StateIn> void adjust_size(const StateIn &);
  order_type order(void) const;
  order_type stepper_order(void) const;
  order_type error_order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
    do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
```

```
    do_step(System, const StateIn &, const DerivIn &, time_type, StateOut &,
            time_type);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, const StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename DerivIn,
           typename Err>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type,
            Err &);
  template<typename System, typename StateIn, typename StateOut, typename Err>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type,
                 Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename Err>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type, Err &);
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  template<typename StateIn> bool resize_impl(const StateIn &);

  // public data members
  static const size_t stage_count;
};
```

Description

This class implements the explicit Runge-Kutta algorithms with error estimation in a generic way. The Butcher tableau is passed to the stepper which constructs the stepper scheme with the help of a template-metaprogramming algorithm. ToDo : Add example!

This class derives explicit_error_stepper_base which provides the stepper interface.

Template Parameters

1. `size_t StageCount`

   The number of stages of the Runge-Kutta algorithm.

2. `size_t Order`

   The order of a stepper if the stepper is used without error estimation.

3. `size_t StepperOrder`

   The order of a step if the stepper is used with error estimation. Usually Order and StepperOrder have the same value.

4. `size_t ErrorOrder`

   The order of the error step if the stepper is used with error estimation.

5. `typename State`

   The type representing the state of the ODE.

6.
```
typename Value = double
```

The floating point type which is used in the computations.

7.
```
typename Deriv = State
```

8.
```
typename Time = Value
```

The type representing the independent variable - the time - of the ODE.

9.
```
typename Algebra = range_algebra
```

The algebra type.

10.
```
typename Operations = default_operations
```

The operations type.

11.
```
typename Resizer = initially_resizer
```

The resizer policy type.

`explicit_error_generic_rk` public construct/copy/destruct

1.
```
explicit_error_generic_rk(const coef_a_type & a, const coef_b_type & b,
                          const coef_b_type & b2, const coef_c_type & c,
                          const algebra_type & algebra = algebra_type());
```

Constructs the explicit_error_generik_rk class with the given parameters a, b, b2 and c. See examples section for details on the coefficients.

Parameters:		
	`a`	Triangular matrix of parameters b in the Butcher tableau.
	`algebra`	A copy of algebra is made and stored inside `explicit_stepper_base`.
	`b`	Last row of the butcher tableau.
	`b2`	Parameters for lower-order evaluation to estimate the error.
	`c`	Parameters to calculate the time points in the Butcher tableau.

`explicit_error_generic_rk` public member functions

1.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename Err>
  void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                    time_type t, StateOut & out, time_type dt, Err & xerr);
```

This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out-of-place, hence the input is in `in` and the output in `out`. Futhermore, an estimation of the error is stored in `xerr`. `do_step_impl` is used by `explicit_error_stepper_base`.

Parameters:		
	`dt`	The step size.
	`dxdt`	The derivative of x at t.
	`in`	The state of the ODE which should be solved. in is not modified in this method
	`out`	The result of the step is written in out.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.

`t`	The value of the time, at which the step should be performed.
`xerr`	The result of the error estimation is written in xerr.

2.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                    time_type t, StateOut & out, time_type dt);
```

This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out-of-place, hence the input is in `in` and the output in `out`. Access to this step functionality is provided by `explicit_stepper_base` and `do_step_impl` should not be called directly.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

3.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters:

`x`	A state from which the size of the temporaries to be resized is deduced.

4.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper if it used without error estimation.

5.
```
order_type stepper_order(void) const;
```

Returns: Returns the order of a step if the stepper is used without error estimation.

6.
```
order_type error_order(void) const;
```

Returns: Returns the order of an error step if the stepper is used without error estimation.

7.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

8.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

9.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

10.
```
template<typename System, typename StateIn, typename StateOut>
  boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
  do_step(System system, const StateIn & in, time_type t, StateOut & out,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. This method is disabled if StateIn and Time are the same type. In this case the method can not be distinguished from other `do_step` variants.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

11.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, const StateIn & in, const DerivIn & dxdt,
          time_type t, StateOut & out, time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

This method is disabled if DerivIn and Time are of same type.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

12.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

The method performs one step with the stepper passed by Stepper and estimates the error. The state of the ODE is updated in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. x is updated by this method.
`xerr`	The estimation of the error is stored in xerr.

13.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

14.
```
template<typename System, typename StateInOut, typename DerivIn, typename Err>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt , xerr );
```

The result is updated in place in x. This method is disabled if Time and DerivIn are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.
`xerr`	The error estimate is stored in xerr.

15.
```
template<typename System, typename StateIn, typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the error is estimated.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

16.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper and the error is estimated. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

This method is disabled if DerivIn and Time are of same type.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

17.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

18.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`explicit_error_generic_rk` private member functions

1. ```
   template<typename StateIn> bool resize_impl(const StateIn & x);
   ```

Header <boost/numeric/odeint/stepper/explicit_generic_rk.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<size_t StageCount, size_t Order, typename State,
               typename Value, typename Deriv, typename Time,
               typename Algebra, typename Operations, typename Resizer>
        class explicit_generic_rk;
    }
  }
}
```

Class template explicit_generic_rk

boost::numeric::odeint::explicit_generic_rk — A generic implementation of explicit Runge-Kutta algorithms. This class is as a base class for all explicit Runge-Kutta steppers.
```

Synopsis

```
// In header: <boost/numeric/odeint/stepper/explicit_generic_rk.hpp>

template<size_t StageCount, size_t Order, typename State, typename Value,
         typename Deriv, typename Time, typename Algebra, typename Operations,
         typename Resizer>
class explicit_generic_rk : public boost::numeric::odeint::explicit_stepper_base< Stepper, Or↵
der, State, Value, Deriv, Time, Algebra, Operations, Resizer >
{
public:
  // types
  typedef explicit_stepper_base< ... >          stepper_base_type;
  typedef stepper_base_type::state_type         state_type;
  typedef stepper_base_type::wrapped_state_type wrapped_state_type;
  typedef stepper_base_type::value_type         value_type;
  typedef stepper_base_type::deriv_type         deriv_type;
  typedef stepper_base_type::wrapped_deriv_type wrapped_deriv_type;
  typedef stepper_base_type::time_type          time_type;
  typedef stepper_base_type::algebra_type       algebra_type;
  typedef stepper_base_type::operations_type    operations_type;
  typedef stepper_base_type::resizer_type       resizer_type;
  typedef unspecified                           rk_algorithm_type;
  typedef rk_algorithm_type::coef_a_type        coef_a_type;
  typedef rk_algorithm_type::coef_b_type        coef_b_type;
  typedef rk_algorithm_type::coef_c_type        coef_c_type;

  // construct/copy/destruct
  explicit_generic_rk(const coef_a_type &, const coef_b_type &,
                      const coef_c_type &,
                      const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type);
  template<typename StateIn> void adjust_size(const StateIn &);
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type);
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  template<typename StateIn> bool resize_impl(const StateIn &);
};
```

Description

This class implements the explicit Runge-Kutta algorithms without error estimation in a generic way. The Butcher tableau is passed to the stepper which constructs the stepper scheme with the help of a template-metaprogramming algorithm. ToDo : Add example!

This class derives explicit_stepper_base which provides the stepper interface.

Template Parameters

1. ```
   size_t StageCount
   ```

   The number of stages of the Runge-Kutta algorithm.

2. ```
   size_t Order
   ```

   The order of the stepper.

3. ```
   typename State
   ```

   The type representing the state of the ODE.

4. ```
   typename Value
   ```

   The floating point type which is used in the computations.

5. ```
   typename Deriv
   ```

6. ```
   typename Time
   ```

   The type representing the independent variable - the time - of the ODE.

7. ```
   typename Algebra
   ```

   The algebra type.

8. ```
   typename Operations
   ```

   The operations type.

9. ```
   typename Resizer
   ```

   The resizer policy type.

`explicit_generic_rk` public construct/copy/destruct

1. ```
   explicit_generic_rk(const coef_a_type & a, const coef_b_type & b,
                       const coef_c_type & c,
                       const algebra_type & algebra = algebra_type());
   ```

   Constructs the `explicit_generic_rk` class. See examples section for details on the coefficients.

   Parameters: `a` Triangular matrix of parameters b in the Butcher tableau.

`algebra`	A copy of algebra is made and stored inside `explicit_stepper_base`.
`b`	Last row of the butcher tableau.
`c`	Parameters to calculate the time points in the Butcher tableau.

`explicit_generic_rk` public member functions

1.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                    time_type t, StateOut & out, time_type dt);
```

This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out of place, hence the input is in `in` and the output in `out`. Access to this step functionality is provided by `explicit_stepper_base` and `do_step_impl` should not be called directly.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

2.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters:

`x`	A state from which the size of the temporaries to be resized is deduced.

3.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper.

4.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

5.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

6.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

7.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

8.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.

`t` The value of the time, at which the step should be performed.

9.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

10.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`explicit_generic_rk` private member functions

1.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

Header <boost/numeric/odeint/stepper/implicit_euler.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename ValueType, typename Resizer = initially_resizer>
        class implicit_euler;
    }
  }
}
```

Class template implicit_euler

boost::numeric::odeint::implicit_euler

Synopsis

```
// In header: <boost/numeric/odeint/stepper/implicit_euler.hpp>

template<typename ValueType, typename Resizer = initially_resizer>
class implicit_euler {
public:
  // types
  typedef ValueType                                          value_type;
  typedef value_type                                         time_type;
  typedef boost::numeric::ublas::vector< value_type >        state_type;
  typedef state_wrapper< state_type >                        wrapped_state_type;
  typedef state_type                                         deriv_type;
  typedef state_wrapper< deriv_type >                        wrapped_deriv_type;
  typedef boost::numeric::ublas::matrix< value_type >        matrix_type;
  typedef state_wrapper< matrix_type >                       wrapped_matrix_type;
  typedef boost::numeric::ublas::permutation_matrix< size_t > pmatrix_type;
  typedef state_wrapper< pmatrix_type >                      wrapped_pmatrix_type;
  typedef Resizer                                            resizer_type;
  typedef stepper_tag                                        stepper_category;
  typedef implicit_euler< ValueType, Resizer >               stepper_type;

  // construct/copy/destruct
  implicit_euler(value_type = 1E-6);

  // public member functions
  template<typename System>
    void do_step(System, state_type &, time_type, time_type);
  template<typename StateType> void adjust_size(const StateType &);

  // private member functions
  template<typename StateIn> bool resize_impl(const StateIn &);
  void solve(state_type &, matrix_type &);
};
```

Description

`implicit_euler` public construct/copy/destruct

1. ```
   implicit_euler(value_type epsilon = 1E-6);
   ```

`implicit_euler` public member functions

1. ```
   template<typename System>
     void do_step(System system, state_type & x, time_type t, time_type dt);
   ```

2. ```
   template<typename StateType> void adjust_size(const StateType & x);
   ```

`implicit_euler` private member functions

1. ```
   template<typename StateIn> bool resize_impl(const StateIn & x);
   ```

2. ```
   void solve(state_type & x, matrix_type & m);
   ```
   ```

Header <boost/numeric/odeint/stepper/modified_midpoint.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class modified_midpoint;
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class modified_midpoint_dense_out;
    }
  }
}
```

Class template modified_midpoint

boost::numeric::odeint::modified_midpoint

Synopsis

```
// In header: <boost/numeric/odeint/stepper/modified_midpoint.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class modified_midpoint : public boost::numeric::odeint::explicit_stepper_base< Stepper, Order, ↵
State, Value, Deriv, Time, Algebra, Operations, Resizer >
{
public:
  // types
  typedef explicit_stepper_base< modified_midpoint< State, Value, Deriv, Time, Algebra, Opera↵
tions, Resizer >, 2, State, Value, Deriv, Time, Algebra, Operations, Resizer > stepper_base_type;
  typedef stepper_base_type::state_type                                                    ↵
                                                                                 state_type;
  typedef stepper_base_type::wrapped_state_type                                            ↵
                                                                         wrapped_state_type;
  typedef stepper_base_type::value_type                                                    ↵
                                                                                 value_type;
  typedef stepper_base_type::deriv_type                                                    ↵
                                                                                 deriv_type;
  typedef stepper_base_type::wrapped_deriv_type                                            ↵
                                                                         wrapped_deriv_type;
  typedef stepper_base_type::time_type                                                     ↵
                                                                                 time_type;
  typedef stepper_base_type::algebra_type                                                  ↵
                                                                                 algebra_type;
  typedef stepper_base_type::operations_type                                               ↵
                                                                                 operations_type;
  typedef stepper_base_type::resizer_type                                                  ↵
                                                                                 resizer_type;
  typedef stepper_base_type::stepper_type                                                  ↵
                                                                                 stepper_type;

  // construct/copy/destruct
  modified_midpoint(unsigned short = 2, const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type);
  void set_steps(unsigned short);
  unsigned short steps(void) const;
  template<typename StateIn> void adjust_size(const StateIn &);
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type);
```

```
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  template<typename StateIn> bool resize_impl(const StateIn &);
};
```

Description

Implementation of the modified midpoint method with a configurable number of intermediate steps. This class is used by the Bulirsch-Stoer algorithm and is not meant for direct usage.

`modified_midpoint` public construct/copy/destruct

1.
```
modified_midpoint(unsigned short steps = 2,
                  const algebra_type & algebra = algebra_type());
```

`modified_midpoint` public member functions

1.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                    time_type t, StateOut & out, time_type dt);
```

2.
```
void set_steps(unsigned short steps);
```

3.
```
unsigned short steps(void) const;
```

4.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

5.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper.

6.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

7.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

8.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

9.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

10.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

11.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

12.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`modified_midpoint` private member functions

1.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

Class template modified_midpoint_dense_out

boost::numeric::odeint::modified_midpoint_dense_out

Synopsis

```
// In header: <boost/numeric/odeint/stepper/modified_midpoint.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class modified_midpoint_dense_out {
public:
  // types
  typedef State                                                                ↵
  state_type;
  typedef Value                                                                ↵
  value_type;
  typedef Deriv                                                                ↵
  deriv_type;
  typedef Time                                                                 ↵
  time_type;
  typedef Algebra                                                              ↵
  algebra_type;
  typedef Operations                                                           ↵
  operations_type;
  typedef Resizer                                                              ↵
  resizer_type;
  typedef state_wrapper< state_type >                                          ↵
  wrapped_state_type;
  typedef state_wrapper< deriv_type >                                          ↵
  wrapped_deriv_type;
  typedef modified_midpoint_dense_out< State, Value, Deriv, Time, Algebra, Operations, Res↵
izer > stepper_type;
  typedef std::vector< wrapped_deriv_type >                                    ↵
  deriv_table_type;

  // construct/copy/destruct
  modified_midpoint_dense_out(unsigned short = 2,
                              const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type, state_type &, deriv_table_type &);
  void set_steps(unsigned short);
  unsigned short steps(void) const;
  template<typename StateIn> bool resize(const StateIn &);
  template<typename StateIn> void adjust_size(const StateIn &);
};
```

Description

Implementation of the modified midpoint method with a configurable number of intermediate steps. This class is used by the dense output Bulirsch-Stoer algorithm and is not meant for direct usage.

Note

This stepper is for internal use only and does not meet any stepper concept.

`modified_midpoint_dense_out` public construct/copy/destruct

1.
```
modified_midpoint_dense_out(unsigned short steps = 2,
                            const algebra_type & algebra = algebra_type());
```

`modified_midpoint_dense_out` public member functions

1.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt, state_type & x_mp,
               deriv_table_type & derivs);
```

2.
```
void set_steps(unsigned short steps);
```

3.
```
unsigned short steps(void) const;
```

4.
```
template<typename StateIn> bool resize(const StateIn & x);
```

5.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Header <boost/numeric/odeint/stepper/rosenbrock4.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Value> struct default_rosenbrock_coefficients;

      template<typename Value,
               typename Coefficients = default_rosenbrock_coefficients< Value >,
               typename Resizer = initially_resizer>
        class rosenbrock4;
    }
  }
}
```

Struct template default_rosenbrock_coefficients

boost::numeric::odeint::default_rosenbrock_coefficients

Synopsis

```
// In header: <boost/numeric/odeint/stepper/rosenbrock4.hpp>

template<typename Value>
struct default_rosenbrock_coefficients {
  // types
  typedef Value          value_type;
  typedef unsigned short order_type;

  // construct/copy/destruct
  default_rosenbrock_coefficients(void);

  // public data members
  const value_type gamma;
  const value_type d1;
  const value_type d2;
  const value_type d3;
  const value_type d4;
  const value_type c2;
  const value_type c3;
  const value_type c4;
  const value_type c21;
  const value_type a21;
  const value_type c31;
  const value_type c32;
  const value_type a31;
  const value_type a32;
  const value_type c41;
  const value_type c42;
  const value_type c43;
  const value_type a41;
  const value_type a42;
  const value_type a43;
  const value_type c51;
  const value_type c52;
  const value_type c53;
  const value_type c54;
  const value_type a51;
  const value_type a52;
  const value_type a53;
  const value_type a54;
  const value_type c61;
  const value_type c62;
  const value_type c63;
  const value_type c64;
  const value_type c65;
  const value_type d21;
  const value_type d22;
  const value_type d23;
  const value_type d24;
  const value_type d25;
  const value_type d31;
  const value_type d32;
  const value_type d33;
  const value_type d34;
  const value_type d35;
  static const order_type stepper_order;
  static const order_type error_order;
};
```

Description

`default_rosenbrock_coefficients` public construct/copy/destruct

1. ```
   default_rosenbrock_coefficients(void);
   ```

Class template rosenbrock4

boost::numeric::odeint::rosenbrock4

Synopsis

```
// In header: <boost/numeric/odeint/stepper/rosenbrock4.hpp>

template<typename Value,
         typename Coefficients = default_rosenbrock_coefficients< Value >,
         typename Resizer = initially_resizer>
class rosenbrock4 {
public:
  // types
  typedef Value                                                 value_type;
  typedef boost::numeric::ublas::vector< value_type >           state_type;
  typedef state_type                                            deriv_type;
  typedef value_type                                            time_type;
  typedef boost::numeric::ublas::matrix< value_type >           matrix_type;
  typedef boost::numeric::ublas::permutation_matrix< size_t > pmatrix_type;
  typedef Resizer                                               resizer_type;
  typedef Coefficients                                          rosenbrock_coefficients;
  typedef stepper_tag                                           stepper_category;
  typedef unsigned short                                        order_type;
  typedef state_wrapper< state_type >                           wrapped_state_type;
  typedef state_wrapper< deriv_type >                           wrapped_deriv_type;
  typedef state_wrapper< matrix_type >                          wrapped_matrix_type;
  typedef state_wrapper< pmatrix_type >                         wrapped_pmatrix_type;
  typedef rosenbrock4< Value, Coefficients, Resizer >           stepper_type;

  // construct/copy/destruct
  rosenbrock4(void);

  // public member functions
  order_type order() const;
  template<typename System>
    void do_step(System, const state_type &, time_type, state_type &,
                 time_type, state_type &);
  template<typename System>
    void do_step(System, state_type &, time_type, time_type, state_type &);
  template<typename System>
    void do_step(System, const state_type &, time_type, state_type &,
                 time_type);
  template<typename System>
    void do_step(System, state_type &, time_type, time_type);
  void prepare_dense_output();
  void calc_state(time_type, state_type &, const state_type &, time_type,
                  const state_type &, time_type);
  template<typename StateType> void adjust_size(const StateType &);

  // protected member functions
  template<typename StateIn> bool resize_impl(const StateIn &);
```
```

```
  template<typename StateIn> bool resize_x_err(const StateIn &);

  // public data members
  static const order_type stepper_order;
  static const order_type error_order;
};
```

Description

rosenbrock4 public construct/copy/destruct

1.
```
rosenbrock4(void);
```

rosenbrock4 public member functions

1.
```
order_type order() const;
```

2.
```
template<typename System>
  void do_step(System system, const state_type & x, time_type t,
               state_type & xout, time_type dt, state_type & xerr);
```

3.
```
template<typename System>
  void do_step(System system, state_type & x, time_type t, time_type dt,
               state_type & xerr);
```

4.
```
template<typename System>
  void do_step(System system, const state_type & x, time_type t,
               state_type & xout, time_type dt);
```

5.
```
template<typename System>
  void do_step(System system, state_type & x, time_type t, time_type dt);
```

6.
```
void prepare_dense_output();
```

7.
```
void calc_state(time_type t, state_type & x, const state_type & x_old,
                time_type t_old, const state_type & x_new, time_type t_new);
```

8.
```
template<typename StateType> void adjust_size(const StateType & x);
```

rosenbrock4 protected member functions

1.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

2.
```
template<typename StateIn> bool resize_x_err(const StateIn & x);
```

Header <boost/numeric/odeint/stepper/rosenbrock4_controller.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Stepper> class rosenbrock4_controller;
    }
  }
}
```

Class template rosenbrock4_controller

boost::numeric::odeint::rosenbrock4_controller

Synopsis

```
// In header: <boost/numeric/odeint/stepper/rosenbrock4_controller.hpp>

template<typename Stepper>
class rosenbrock4_controller {
public:
  // types
  typedef Stepper                             stepper_type;
  typedef stepper_type::value_type            value_type;
  typedef stepper_type::state_type            state_type;
  typedef stepper_type::wrapped_state_type    wrapped_state_type;
  typedef stepper_type::time_type             time_type;
  typedef stepper_type::deriv_type            deriv_type;
  typedef stepper_type::wrapped_deriv_type    wrapped_deriv_type;
  typedef stepper_type::resizer_type          resizer_type;
  typedef controlled_stepper_tag              stepper_category;
  typedef rosenbrock4_controller< Stepper > controller_type;

  // construct/copy/destruct
  rosenbrock4_controller(value_type = 1.0e-6, value_type = 1.0e-6,
                         const stepper_type & = stepper_type());

  // public member functions
  value_type error(const state_type &, const state_type &, const state_type &);
  value_type last_error(void) const;
  template<typename System>
    boost::numeric::odeint::controlled_step_result
    try_step(System, state_type &, time_type &, time_type &);
  template<typename System>
    boost::numeric::odeint::controlled_step_result
    try_step(System, const state_type &, time_type &, state_type &,
             time_type &);
  template<typename StateType> void adjust_size(const StateType &);
  stepper_type & stepper(void);
  const stepper_type & stepper(void) const;

  // private member functions
  template<typename StateIn> bool resize_m_xerr(const StateIn &);
  template<typename StateIn> bool resize_m_xnew(const StateIn &);
};
```

Description

`rosenbrock4_controller` public construct/copy/destruct

1.
```
rosenbrock4_controller(value_type atol = 1.0e-6, value_type rtol = 1.0e-6,
                       const stepper_type & stepper = stepper_type());
```

`rosenbrock4_controller` public member functions

1.
```
value_type error(const state_type & x, const state_type & xold,
                 const state_type & xerr);
```

2.
```
value_type last_error(void) const;
```

3.
```
template<typename System>
  boost::numeric::odeint::controlled_step_result
  try_step(System sys, state_type & x, time_type & t, time_type & dt);
```

4.
```
template<typename System>
  boost::numeric::odeint::controlled_step_result
  try_step(System sys, const state_type & x, time_type & t, state_type & xout,
           time_type & dt);
```

5.
```
template<typename StateType> void adjust_size(const StateType & x);
```

6.
```
stepper_type & stepper(void);
```

7.
```
const stepper_type & stepper(void) const;
```

`rosenbrock4_controller` private member functions

1.
```
template<typename StateIn> bool resize_m_xerr(const StateIn & x);
```

2.
```
template<typename StateIn> bool resize_m_xnew(const StateIn & x);
```

Header <boost/numeric/odeint/stepper/rosenbrock4_dense_output.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename ControlledStepper> class rosenbrock4_dense_output;
    }
  }
}
```

Class template rosenbrock4_dense_output

boost::numeric::odeint::rosenbrock4_dense_output

Synopsis

```
// In header: <boost/numeric/odeint/stepper/rosenbrock4_dense_output.hpp>

template<typename ControlledStepper>
class rosenbrock4_dense_output {
public:
  // types
  typedef ControlledStepper                                 controlled_stepper_type;
  typedef controlled_stepper_type::stepper_type             stepper_type;
  typedef stepper_type::value_type                          value_type;
  typedef stepper_type::state_type                          state_type;
  typedef stepper_type::wrapped_state_type                  wrapped_state_type;
  typedef stepper_type::time_type                           time_type;
  typedef stepper_type::deriv_type                          deriv_type;
  typedef stepper_type::wrapped_deriv_type                  wrapped_deriv_type;
  typedef stepper_type::resizer_type                        resizer_type;
  typedef dense_output_stepper_tag                          stepper_category;
  typedef rosenbrock4_dense_output< ControlledStepper > dense_output_stepper_type;

  // construct/copy/destruct
  rosenbrock4_dense_output(const controlled_stepper_type & = controlled_stepper_type());

  // public member functions
  template<typename StateType>
    void initialize(const StateType &, time_type, time_type);
  template<typename System> std::pair< time_type, time_type > do_step(System);
  template<typename StateOut> void calc_state(time_type, StateOut &);
  template<typename StateOut> void calc_state(time_type, const StateOut &);
  template<typename StateType> void adjust_size(const StateType &);
  const state_type & current_state(void) const;
  time_type current_time(void) const;
  const state_type & previous_state(void) const;
  time_type previous_time(void) const;
  time_type current_time_step(void) const;

  // private member functions
  state_type & get_current_state(void);
  const state_type & get_current_state(void) const;
  state_type & get_old_state(void);
  const state_type & get_old_state(void) const;
  void toggle_current_state(void);
  template<typename StateIn> bool resize_impl(const StateIn &);
};
```

Description

`rosenbrock4_dense_output` public construct/copy/destruct

1.
```
rosenbrock4_dense_output(const controlled_stepper_type & stepper = controlled_stepper_type());
```

`rosenbrock4_dense_output` public member functions

1.
```
template<typename StateType>
  void initialize(const StateType & x0, time_type t0, time_type dt0);
```

2.
```
template<typename System>
  std::pair< time_type, time_type > do_step(System system);
```

3.
```
template<typename StateOut> void calc_state(time_type t, StateOut & x);
```

4.
```
template<typename StateOut> void calc_state(time_type t, const StateOut & x);
```

5.
```
template<typename StateType> void adjust_size(const StateType & x);
```

6.
```
const state_type & current_state(void) const;
```

7.
```
time_type current_time(void) const;
```

8.
```
const state_type & previous_state(void) const;
```

9.
```
time_type previous_time(void) const;
```

10.
```
time_type current_time_step(void) const;
```

`rosenbrock4_dense_output` private member functions

1.
```
state_type & get_current_state(void);
```

2.
```
const state_type & get_current_state(void) const;
```

3.
```
state_type & get_old_state(void);
```

4.
```
const state_type & get_old_state(void) const;
```

5.
```
void toggle_current_state(void);
```

6.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

Header <boost/numeric/odeint/stepper/runge_kutta4.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class runge_kutta4;
    }
  }
}
```

Class template runge_kutta4

boost::numeric::odeint::runge_kutta4 — The classical Runge-Kutta stepper of fourth order.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/runge_kutta4.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class runge_kutta4 : public boost::numeric::odeint::explicit_generic_rk< StageCount, Order, ↵
State, Value, Deriv, Time, Algebra, Operations, Resizer >
{
public:
  // types
  typedef stepper_base_type::state_type      state_type;
  typedef stepper_base_type::value_type      value_type;
  typedef stepper_base_type::deriv_type      deriv_type;
  typedef stepper_base_type::time_type       time_type;
  typedef stepper_base_type::algebra_type    algebra_type;
  typedef stepper_base_type::operations_type operations_type;
  typedef stepper_base_type::resizer_type    resizer_type;

  // construct/copy/destruct
  runge_kutta4(const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type);
  template<typename StateIn> void adjust_size(const StateIn &);
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type);
  algebra_type & algebra();
  const algebra_type & algebra() const;
};
```

Description

The Runge-Kutta method of fourth order is one standard method for solving ordinary differential equations and is widely used, see also en.wikipedia.org/wiki/Runge-Kutta_methods The method is explicit and fulfills the Stepper concept. Step size control or continuous output are not provided.

This class derives from explicit_stepper_base and inherits its interface via CRTP (current recurring template pattern). Furthermore, it derivs from explicit_generic_rk which is a generic Runge-Kutta algorithm. For more details see explicit_stepper_base and explicit_generic_rk.

Template Parameters

1. ```
   typename State
   ```

   The state type.

2. ```
   typename Value = double
   ```

   The value type.

3. ```
   typename Deriv = State
   ```

   The type representing the time derivative of the state.

4. ```
   typename Time = Value
   ```

   The time representing the independent variable - the time.

5. ```
   typename Algebra = range_algebra
   ```

   The algebra type.

6. ```
   typename Operations = default_operations
   ```

   The operations type.

7. ```
   typename Resizer = initially_resizer
   ```

   The resizer policy type.

`runge_kutta4` public construct/copy/destruct

1. ```
   runge_kutta4(const algebra_type & algebra = algebra_type());
   ```

   Constructs the `runge_kutta4` class. This constructor can be used as a default constructor if the algebra has a default constructor.

   Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`runge_kutta4` public member functions

1. ```
   template<typename System, typename StateIn, typename DerivIn,
            typename StateOut>
     void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                       time_type t, StateOut & out, time_type dt);
   ```

   This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out of place, hence the input is in `in` and the output in `out`. Access to this step functionality is provided by `explicit_stepper_base` and `do_step_impl` should not be called directly.

   Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
   ```

system	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
t	The value of the time, at which the step should be performed.

2.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: x A state from which the size of the temporaries to be resized is deduced.

3.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper.

4.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

dt	The step size.
system	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
t	The value of the time, at which the step should be performed.
x	The state of the ODE which should be solved. After calling do_step the result is updated in x.

5.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

6.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

dt	The step size.
dxdt	The derivative of x at t.
system	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
t	The value of the time, at which the step should be performed.
x	The state of the ODE which should be solved. After calling do_step the result is updated in x.

7.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

8.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

9.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

10.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

Header <boost/numeric/odeint/stepper/runge_kutta4_classic.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class runge_kutta4_classic;
    }
  }
}
```

Class template runge_kutta4_classic

boost::numeric::odeint::runge_kutta4_classic — The classical Runge-Kutta stepper of fourth order.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/runge_kutta4_classic.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class runge_kutta4_classic : public boost::numeric::odeint::explicit_stepper_base< Stepper, Or↵
der, State, Value, Deriv, Time, Algebra, Operations, Resizer >
{
public:
  // types
  typedef explicit_stepper_base< runge_kutta4_classic< ... >,... > stepper_base_type;
  typedef stepper_base_type::state_type                             state_type;
  typedef stepper_base_type::value_type                             value_type;
  typedef stepper_base_type::deriv_type                             deriv_type;
  typedef stepper_base_type::time_type                              time_type;
  typedef stepper_base_type::algebra_type                           algebra_type;
  typedef stepper_base_type::operations_type                        operations_type;
  typedef stepper_base_type::resizer_type                           resizer_type;

  // construct/copy/destruct
  runge_kutta4_classic(const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type);
  template<typename StateType> void adjust_size(const StateType &);
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type);
  template<typename StateIn> void adjust_size(const StateIn &);
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  template<typename StateIn> bool resize_impl(const StateIn &);
};
```

Description

The Runge-Kutta method of fourth order is one standard method for solving ordinary differential equations and is widely used, see also en.wikipedia.org/wiki/Runge-Kutta_methods The method is explicit and fulfills the Stepper concept. Step size control or continuous output are not provided. This class implements the method directly, hence the generic Runge-Kutta algorithm is not used.

This class derives from explicit_stepper_base and inherits its interface via CRTP (current recurring template pattern). For more details see explicit_stepper_base.

Template Parameters

1. ```
   typename State
   ```

   The state type.

2. ```
   typename Value = double
   ```

   The value type.

3. ```
   typename Deriv = State
   ```

   The type representing the time derivative of the state.

4. ```
   typename Time = Value
   ```

   The time representing the independent variable - the time.

5. ```
   typename Algebra = range_algebra
   ```

   The algebra type.

6. ```
   typename Operations = default_operations
   ```

   The operations type.

7. ```
   typename Resizer = initially_resizer
   ```

   The resizer policy type.

`runge_kutta4_classic` public construct/copy/destruct

1. ```
   runge_kutta4_classic(const algebra_type & algebra = algebra_type());
   ```

   Constructs the `runge_kutta4_classic` class. This constructor can be used as a default constructor if the algebra has a default constructor.

   Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`runge_kutta4_classic` public member functions

1. ```
   template<typename System, typename StateIn, typename DerivIn,
            typename StateOut>
     void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                       time_type t, StateOut & out, time_type dt);
   ```

   This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out of place, hence the input is in `in` and the output in `out`. Access to this step functionality is provided by `explicit_stepper_base` and `do_step_impl` should not be called directly.

   Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
   ```

out	The result of the step is written in out.
system	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
t	The value of the time, at which the step should be performed.

2.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: x A state from which the size of the temporaries to be resized is deduced.

3.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper.

4.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:		
	dt	The step size.
	system	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
	t	The value of the time, at which the step should be performed.
	x	The state of the ODE which should be solved. After calling do_step the result is updated in x.

5.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

6.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:		
	dt	The step size.
	dxdt	The derivative of x at t.
	system	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	t	The value of the time, at which the step should be performed.
	x	The state of the ODE which should be solved. After calling do_step the result is updated in x.

7.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

8.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt);
```

The method performs one step. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

9.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters:

`x`	A state from which the size of the temporaries to be resized is deduced.

10.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

11.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`runge_kutta4_classic` private member functions

1. ```
   template<typename StateIn> bool resize_impl(const StateIn & x);
   ```

Header <boost/numeric/odeint/stepper/runge_kutta_cash_karp54.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class runge_kutta_cash_karp54;
    }
  }
}
```

Class template runge_kutta_cash_karp54

boost::numeric::odeint::runge_kutta_cash_karp54 — The Runge-Kutta Cash-Karp method.
```

Synopsis

```
// In header: <boost/numeric/odeint/stepper/runge_kutta_cash_karp54.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class runge_kutta_cash_karp54 : public boost::numeric::odeint::explicit_error_generic_rk< Stage↵
Count, Order, StepperOrder, ErrorOrder, State, Value, Deriv, Time, Algebra, Operations, Resizer >
{
public:
  // types
  typedef stepper_base_type::state_type      state_type;
  typedef stepper_base_type::value_type      value_type;
  typedef stepper_base_type::deriv_type      deriv_type;
  typedef stepper_base_type::time_type       time_type;
  typedef stepper_base_type::algebra_type    algebra_type;
  typedef stepper_base_type::operations_type operations_type;
  typedef stepper_base_type::resizer_type    resizer_typ;

  // construct/copy/destruct
  runge_kutta_cash_karp54(const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename Err>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type, Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type);
  template<typename StateIn> void adjust_size(const StateIn &);
  order_type order(void) const;
  order_type stepper_order(void) const;
  order_type error_order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
    do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, const StateIn &, const DerivIn &, time_type, StateOut &,
            time_type);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, const StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename DerivIn,
           typename Err>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type,
            Err &);
  template<typename System, typename StateIn, typename StateOut, typename Err>
```

```
    void do_step(System, const StateIn &, time_type, StateOut &, time_type,
                 Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename Err>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type, Err &);
  algebra_type & algebra();
  const algebra_type & algebra() const;
};
```

Description

The Runge-Kutta Cash-Karp method is one of the standard methods for solving ordinary differential equations, see en.wikipedia.org/wiki/Cash-Karp_methods. The method is explicit and fulfills the Error Stepper concept. Step size control is provided but continuous output is not available for this method.

This class derives from explicit_error_stepper_base and inherits its interface via CRTP (current recurring template pattern). Furthermore, it derivs from explicit_error_generic_rk which is a generic Runge-Kutta algorithm with error estimation. For more details see explicit_error_stepper_base and explicit_error_generic_rk.

Template Parameters

1. `typename State`

   The state type.

2. `typename Value = double`

   The value type.

3. `typename Deriv = State`

   The type representing the time derivative of the state.

4. `typename Time = Value`

   The time representing the independent variable - the time.

5. `typename Algebra = range_algebra`

   The algebra type.

6. `typename Operations = default_operations`

   The operations type.

7. `typename Resizer = initially_resizer`

   The resizer policy type.

`runge_kutta_cash_karp54` public construct/copy/destruct

1. `runge_kutta_cash_karp54(const algebra_type & algebra = algebra_type());`

Constructs the `runge_kutta_cash_karp54` class. This constructor can be used as a default constructor if the algebra has a default constructor.

Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`runge_kutta_cash_karp54` public member functions

1.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename Err>
  void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                    time_type t, StateOut & out, time_type dt, Err & xerr);
```

This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out-of-place, hence the input is in `in` and the output in `out`. Futhermore, an estimation of the error is stored in `xerr`. `do_step_impl` is used by `explicit_error_stepper_base`.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The result of the error estimation is written in xerr.

2.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                    time_type t, StateOut & out, time_type dt);
```

This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out-of-place, hence the input is in `in` and the output in `out`. Access to this step functionality is provided by `explicit_stepper_base` and `do_step_impl` should not be called directly.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

3.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

4.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper if it used without error estimation.

5.
```
order_type stepper_order(void) const;
```

Returns: Returns the order of a step if the stepper is used without error estimation.

6.
```
order_type error_order(void) const;
```

Returns: Returns the order of an error step if the stepper is used without error estimation.

7.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

8.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

9.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

10.
```
template<typename System, typename StateIn, typename StateOut>
  boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
  do_step(System system, const StateIn & in, time_type t, StateOut & out,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. This method is disabled if StateIn and Time are the same type. In this case the method can not be distinguished from other `do_step` variants.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

11.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, const StateIn & in, const DerivIn & dxdt,
          time_type t, StateOut & out, time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

This method is disabled if DerivIn and Time are of same type.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

12.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

The method performs one step with the stepper passed by Stepper and estimates the error. The state of the ODE is updated in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. x is updated by this method.
`xerr`	The estimation of the error is stored in xerr.

13.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

14.
```
template<typename System, typename StateInOut, typename DerivIn, typename Err>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt , xerr );
```

The result is updated in place in x. This method is disabled if Time and DerivIn are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.
`xerr`	The error estimate is stored in xerr.

15.
```
template<typename System, typename StateIn, typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the error is estimated.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

16.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper and the error is estimated. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

This method is disabled if DerivIn and Time are of same type.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

17.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

18.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

Header <boost/numeric/odeint/stepper/runge_kutta_cash_karp54_classic.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = double,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class runge_kutta_cash_karp54_classic;
    }
  }
}
```

Class template runge_kutta_cash_karp54_classic

boost::numeric::odeint::runge_kutta_cash_karp54_classic — The Runge-Kutta Cash-Karp method implemented without the generic Runge-Kutta algorithm.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/runge_kutta_cash_karp54_classic.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = double, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class runge_kutta_cash_karp54_classic : public boost::numeric::odeint::explicit_error_step↵
per_base< Stepper, Order, StepperOrder, ErrorOrder, State, Value, Deriv, Time, Algebra, Opera↵
tions, Resizer >
{
public:
  // types
  typedef explicit_error_stepper_base< runge_kutta_cash_karp54_classic< ... >,... > step↵
per_base_type;
  typedef stepper_base_type::state_type                                    state_type; ↵

  typedef stepper_base_type::value_type                                    value_type; ↵

  typedef stepper_base_type::deriv_type                                    deriv_type; ↵

  typedef stepper_base_type::time_type                                      time_type; ↵

  typedef stepper_base_type::algebra_type                                algebra_type; ↵

  typedef stepper_base_type::operations_type                                opera↵
tions_type;
  typedef stepper_base_type::resizer_type                                resizer_type; ↵


  // construct/copy/destruct
  runge_kutta_cash_karp54_classic(const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename Err>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type, Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type);
  template<typename StateIn> void adjust_size(const StateIn &);
  order_type order(void) const;
  order_type stepper_order(void) const;
  order_type error_order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
    do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, const StateIn &, const DerivIn &, time_type, StateOut &,
            time_type);
```

```
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, const StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename DerivIn,
           typename Err>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type,
            Err &);
  template<typename System, typename StateIn, typename StateOut, typename Err>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type,
                 Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename Err>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type, Err &);
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  template<typename StateIn> bool resize_impl(const StateIn &);
};
```

Description

The Runge-Kutta Cash-Karp method is one of the standard methods for solving ordinary differential equations, see en.wikipedia.org/wiki/Cash-Karp_method. The method is explicit and fulfills the Error Stepper concept. Step size control is provided but continuous output is not available for this method.

This class derives from explicit_error_stepper_base and inherits its interface via CRTP (current recurring template pattern). This class implements the method directly, hence the generic Runge-Kutta algorithm is not used.

Template Parameters

1. ```
   typename State
   ```

   The state type.

2. ```
   typename Value = double
   ```

   The value type.

3. ```
   typename Deriv = State
   ```

   The type representing the time derivative of the state.

4. ```
   typename Time = double
   ```

   The time representing the independent variable - the time.

5. ```
   typename Algebra = range_algebra
   ```

   The algebra type.

6. ```
   typename Operations = default_operations
   ```

The operations type.

7.
```
typename Resizer = initially_resizer
```

The resizer policy type.

`runge_kutta_cash_karp54_classic` public construct/copy/destruct

1.
```
runge_kutta_cash_karp54_classic(const algebra_type & algebra = algebra_type());
```

Constructs the `runge_kutta_cash_karp54_classic` class. This constructor can be used as a default constructor if the algebra has a default constructor.

Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`runge_kutta_cash_karp54_classic` public member functions

1.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename Err>
  void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                    time_type t, StateOut & out, time_type dt, Err & xerr);
```

This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method.

The result is updated out-of-place, hence the input is in `in` and the output in `out`. Futhermore, an estimation of the error is stored in `xerr`. Access to this step functionality is provided by `explicit_error_stepper_base` and `do_step_impl` should not be called directly.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The result of the error estimation is written in xerr.

2.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                    time_type t, StateOut & out, time_type dt);
```

This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out-of-place, hence the input is in `in` and the output in `out`. Access to this step functionality is provided by `explicit_error_stepper_base` and `do_step_impl` should not be called directly.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

3.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

4.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper if it used without error estimation.

5.
```
order_type stepper_order(void) const;
```

Returns: Returns the order of a step if the stepper is used without error estimation.

6.
```
order_type error_order(void) const;
```

Returns: Returns the order of an error step if the stepper is used without error estimation.

7.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

8.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

9.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

10.
```
template<typename System, typename StateIn, typename StateOut>
  boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
  do_step(System system, const StateIn & in, time_type t, StateOut & out,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. This method is disabled if StateIn and Time are the same type. In this case the method can not be distinguished from other `do_step` variants.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

11.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, const StateIn & in, const DerivIn & dxdt,
          time_type t, StateOut & out, time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

This method is disabled if DerivIn and Time are of same type.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

12.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

The method performs one step with the stepper passed by Stepper and estimates the error. The state of the ODE is updated in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. x is updated by this method.

`xerr`	The estimation of the error is stored in xerr.

13.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

14.
```
template<typename System, typename StateInOut, typename DerivIn, typename Err>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt , xerr );
```

The result is updated in place in x. This method is disabled if Time and DerivIn are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.
`xerr`	The error estimate is stored in xerr.

15.
```
template<typename System, typename StateIn, typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the error is estimated.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

16.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper and the error is estimated. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

This method is disabled if DerivIn and Time are of same type.

Note

This method does not solve the forwarding problem.

Parameters:		
	`dt`	The step size.
	`dxdt`	The derivative of x at t.
	`in`	The state of the ODE which should be solved. in is not modified in this method
	`out`	The result of the step is written in out.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time, at which the step should be performed.
	`xerr`	The error estimate.

17.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

18.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`runge_kutta_cash_karp54_classic` private member functions

1.
```
template<typename StateIn> bool resize_impl(const StateIn & x);
```

Header <boost/numeric/odeint/stepper/runge_kutta_dopri5.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class runge_kutta_dopri5;
    }
  }
}
```

Class template runge_kutta_dopri5

boost::numeric::odeint::runge_kutta_dopri5 — The Runge-Kutta Dormand-Prince 5 method.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/runge_kutta_dopri5.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class runge_kutta_dopri5 : public boost::numeric::odeint::explicit_error_stepper_fsal_base< Step↵
per, Order, StepperOrder, ErrorOrder, State, Value, Deriv, Time, Algebra, Operations, Resizer >
{
public:
  // types
  typedef explicit_error_stepper_fsal_base< runge_kutta_dopri5< ... >,... > stepper_base_type;
  typedef stepper_base_type::state_type                                      state_type;
  typedef stepper_base_type::value_type                                      value_type;
  typedef stepper_base_type::deriv_type                                      deriv_type;
  typedef stepper_base_type::time_type                                       time_type;
  typedef stepper_base_type::algebra_type                                    algebra_type;
  typedef stepper_base_type::operations_type                                 operations_type;
  typedef stepper_base_type::resizer_type                                    resizer_type;

  // construct/copy/destruct
  runge_kutta_dopri5(const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename DerivOut>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, DerivOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename DerivOut, typename Err>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, DerivOut &, time_type, Err &);
  template<typename StateOut, typename StateIn1, typename DerivIn1,
           typename StateIn2, typename DerivIn2>
    void calc_state(time_type, StateOut &, const StateIn1 &, const DerivIn1 &,
                    time_type, const StateIn2 &, const DerivIn2 &, time_type) const;
  template<typename StateIn> void adjust_size(const StateIn &);
  order_type order(void) const;
  order_type stepper_order(void) const;
  order_type error_order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivInOut>
    boost::disable_if< boost::is_same< StateInOut, time_type >, void >::type
    do_step(System, StateInOut &, DerivInOut &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
    do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename DerivOut>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, DerivOut &, time_type);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, StateInOut &, time_type, time_type, Err &);
```

```
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, const StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename DerivInOut,
           typename Err>
    boost::disable_if< boost::is_same< StateInOut, time_type >, void >::type
    do_step(System, StateInOut &, DerivInOut &, time_type, time_type, Err &);
  template<typename System, typename StateIn, typename StateOut, typename Err>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type,
                 Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename DerivOut, typename Err>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, DerivOut &, time_type, Err &);
  void reset(void);
  template<typename DerivIn> void initialize(const DerivIn &);
  template<typename System, typename StateIn>
    void initialize(System, const StateIn &, time_type);
  bool is_initialized(void) const;
  algebra_type & algebra();
  const algebra_type & algebra() const;

  // private member functions
  template<typename StateIn> bool resize_k_x_tmp_impl(const StateIn &);
  template<typename StateIn> bool resize_dxdt_tmp_impl(const StateIn &);
};
```

Description

The Runge-Kutta Dormand-Prince 5 method is a very popular method for solving ODEs, see . The method is explicit and fulfills the Error Stepper concept. Step size control is provided but continuous output is available which make this method favourable for many applications.

This class derives from explicit_error_stepper_fsal_base and inherits its interface via CRTP (current recurring template pattern). The method possesses the FSAL (first-same-as-last) property. See explicit_error_stepper_fsal_base for more details.

Template Parameters

1. `typename State`

   The state type.

2. `typename Value = double`

   The value type.

3. `typename Deriv = State`

   The type representing the time derivative of the state.

4. `typename Time = Value`

   The time representing the independent variable - the time.

5. `typename Algebra = range_algebra`

   The algebra type.

6.
```
typename Operations = default_operations
```

The operations type.

7.
```
typename Resizer = initially_resizer
```

The resizer policy type.

`runge_kutta_dopri5` public construct/copy/destruct

1.
```
runge_kutta_dopri5(const algebra_type & algebra = algebra_type());
```

Constructs the `runge_kutta_dopri5` class. This constructor can be used as a default constructor if the algebra has a default constructor.

Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`runge_kutta_dopri5` public member functions

1.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename DerivOut>
  void do_step_impl(System system, const StateIn & in,
                    const DerivIn & dxdt_in, time_type t, StateOut & out,
                    DerivOut & dxdt_out, time_type dt);
```

This method performs one step. The derivative `dxdt_in` of `in` at the time `t` is passed to the method. The result is updated out-of-place, hence the input is in `in` and the output in `out`. Furthermore, the derivative is update out-of-place, hence the input is assumed to be in `dxdt_in` and the output in `dxdt_out`. Access to this step functionality is provided by `explicit_error_stepper_fsal_base` and `do_step_impl` should not be called directly.

Parameters:		
	`dt`	The step size.
	`dxdt_in`	The derivative of x at t. dxdt_in is not modified by this method
	`dxdt_out`	The result of the new derivative at time t+dt.
	`in`	The state of the ODE which should be solved. in is not modified in this method
	`out`	The result of the step is written in out.
	`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
	`t`	The value of the time, at which the step should be performed.

2.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename DerivOut, typename Err>
  void do_step_impl(System system, const StateIn & in,
                    const DerivIn & dxdt_in, time_type t, StateOut & out,
                    DerivOut & dxdt_out, time_type dt, Err & xerr);
```

This method performs one step. The derivative `dxdt_in` of `in` at the time `t` is passed to the method. The result is updated out-of-place, hence the input is in `in` and the output in `out`. Furthermore, the derivative is update out-of-place, hence the input is assumed to be in `dxdt_in` and the output in `dxdt_out`. Access to this step functionality is provided by `explicit_error_stepper_fsal_base` and `do_step_impl` should not be called directly. An estimation of the error is calculated.

Parameters:		
	`dt`	The step size.
	`dxdt_in`	The derivative of x at t. dxdt_in is not modified by this method
	`dxdt_out`	The result of the new derivative at time t+dt.
	`in`	The state of the ODE which should be solved. in is not modified in this method
	`out`	The result of the step is written in out.

system	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
t	The value of the time, at which the step should be performed.
xerr	An estimation of the error.

3.
```
template<typename StateOut, typename StateIn1, typename DerivIn1,
         typename StateIn2, typename DerivIn2>
  void calc_state(time_type t, StateOut & x, const StateIn1 & x_old,
                  const DerivIn1 & deriv_old, time_type t_old,
                  const StateIn2 &, const DerivIn2 & deriv_new,
                  time_type t_new) const;
```

This method is used for continuous output and it calculates the state `x` at a time `t` from the knowledge of two states `old_state` and `current_state` at time points `t_old` and `t_new`. It also uses internal variables to calculate the result. Hence this method must be called after two successful `do_step` calls.

4.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

5.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper if it used without error estimation.

6.
```
order_type stepper_order(void) const;
```

Returns: Returns the order of a step if the stepper is used without error estimation.

7.
```
order_type error_order(void) const;
```

Returns: Returns the order of an error step if the stepper is used without error estimation.

8.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Note

This method uses the internal state of the stepper.

Parameters:

dt	The step size.
system	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
t	The value of the time, at which the step should be performed.
x	The state of the ODE which should be solved. After calling do_step the result is updated in x.

9.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

10.
```
template<typename System, typename StateInOut, typename DerivInOut>
  boost::disable_if< boost::is_same< StateInOut, time_type >, void >::type
  do_step(System system, StateInOut & x, DerivInOut & dxdt, time_type t,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other methods the derivative of x is also passed to this method. Therefore, dxdt must be evaluated initially:

```
ode( x , dxdt , t );
for( ... )
{
    stepper.do_step( ode , x , dxdt , t , dt );
    t += dt;
}
```

Note

This method does NOT use the initial state, since the first derivative is explicitly passed to this method.

The result is updated in place in x as well as the derivative dxdt. This method is disabled if Time and StateInOut are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t. After calling `do_step` dxdt is updated to the new value.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

11.
```
template<typename System, typename StateIn, typename StateOut>
  boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
  do_step(System system, const StateIn & in, time_type t, StateOut & out,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. This method is disabled if StateIn and Time are the same type. In this case the method can not be distinguished from other `do_step` variants.

Note

This method uses the internal state of the stepper.

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

12.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename DerivOut>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt_in,
               time_type t, StateOut & out, DerivOut & dxdt_out,
               time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper and updated by the stepper to its new value at t+dt.

Note

This method does not solve the forwarding problem.

This method does NOT use the internal state of the stepper.

Parameters:

`dt`	The step size.
`dxdt_in`	The derivative of x at t.
`dxdt_out`	The updated derivative of `out` at `t+dt`.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

13.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

The method performs one step with the stepper passed by Stepper and estimates the error. The state of the ODE is updated in-place.

Note

This method uses the internal state of the stepper.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. x is updated by this method.
`xerr`	The estimation of the error is stored in xerr.

14.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

15.
```
template<typename System, typename StateInOut, typename DerivInOut,
         typename Err>
  boost::disable_if< boost::is_same< StateInOut, time_type >, void >::type
  do_step(System system, StateInOut & x, DerivInOut & dxdt, time_type t,
          time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method and updated by this method.

Note

This method does NOT use the internal state of the stepper.

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions. This method is disabled if StateInOut and Time are of the same type.

Note

This method does NOT use the internal state of the stepper.

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t. After calling `do_step` this value is updated to the new value at `t+dt`.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.
`xerr`	The error estimate is stored in xerr.

16.
```
template<typename System, typename StateIn, typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the error is estimated.

Note

This method uses the internal state of the stepper.

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

17.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename DerivOut, typename Err>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt_in,
               time_type t, StateOut & out, DerivOut & dxdt_out,
               time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper and the error is estimated.

Note

This method does NOT use the internal state of the stepper.

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt_in`	The derivative of x at t.
`dxdt_out`	The new derivative at `t+dt` is written into this variable.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

18.
```
void reset(void);
```

Resets the internal state of this stepper. After calling this method it is safe to use all `do_step` method without explicitly initializing the stepper.

19.
```
template<typename DerivIn> void initialize(const DerivIn & deriv);
```

Initializes the internal state of the stepper.

Parameters:

`deriv`	The derivative of x. The next call of `do_step` expects that the derivative of `x` passed to `do_step` has the value of `deriv`.

20.
```
template<typename System, typename StateIn>
  void initialize(System system, const StateIn & x, time_type t);
```

Initializes the internal state of the stepper.

This method is equivalent to

```
Deriv dxdt;
system( x , dxdt , t );
stepper.initialize( dxdt );
```

Parameters:

`system`	The system function for the next calls of `do_step`.
`t`	The current time of the ODE.
`x`	The current state of the ODE.

21.
```
bool is_initialized(void) const;
```

Returns if the stepper is already initialized. If the stepper is not initialized, the first call of `do_step` will initialize the state of the stepper. If the stepper is already initialized the system function can not be safely exchanged between consecutive `do_step` calls.

22.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

23.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

`runge_kutta_dopri5` private member functions

1.
```
template<typename StateIn> bool resize_k_x_tmp_impl(const StateIn & x);
```

2.
```
template<typename StateIn> bool resize_dxdt_tmp_impl(const StateIn & x);
```

Header <boost/numeric/odeint/stepper/runge_kutta_fehlberg78.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename State, typename Value = double,
               typename Deriv = State, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class runge_kutta_fehlberg78;
    }
  }
}
```

Class template runge_kutta_fehlberg78

boost::numeric::odeint::runge_kutta_fehlberg78 — The Runge-Kutta Fehlberg 78 method.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/runge_kutta_fehlberg78.hpp>

template<typename State, typename Value = double, typename Deriv = State,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class runge_kutta_fehlberg78 : public boost::numeric::odeint::explicit_error_generic_rk< Stage↵
Count, Order, StepperOrder, ErrorOrder, State, Value, Deriv, Time, Algebra, Operations, Resizer >
{
public:
  // types
  typedef stepper_base_type::state_type      state_type;
  typedef stepper_base_type::value_type      value_type;
  typedef stepper_base_type::deriv_type      deriv_type;
  typedef stepper_base_type::time_type       time_type;
  typedef stepper_base_type::algebra_type    algebra_type;
  typedef stepper_base_type::operations_type operations_type;
  typedef stepper_base_type::resizer_type    resizer_type;

  // construct/copy/destruct
  runge_kutta_fehlberg78(const algebra_type & = algebra_type());

  // public member functions
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename Err>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type, Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    void do_step_impl(System, const StateIn &, const DerivIn &, time_type,
                      StateOut &, time_type);
  template<typename StateIn> void adjust_size(const StateIn &);
  order_type order(void) const;
  order_type stepper_order(void) const;
  order_type error_order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut, typename DerivIn>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type);
  template<typename System, typename StateIn, typename StateOut>
    boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
    do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, const StateIn &, const DerivIn &, time_type, StateOut &,
            time_type);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename Err>
    void do_step(System, const StateInOut &, time_type, time_type, Err &);
  template<typename System, typename StateInOut, typename DerivIn,
           typename Err>
    boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
    do_step(System, StateInOut &, const DerivIn &, time_type, time_type,
            Err &);
  template<typename System, typename StateIn, typename StateOut, typename Err>
```

```
    void do_step(System, const StateIn &, time_type, StateOut &, time_type,
                 Err &);
  template<typename System, typename StateIn, typename DerivIn,
           typename StateOut, typename Err>
    void do_step(System, const StateIn &, const DerivIn &, time_type,
                 StateOut &, time_type, Err &);
  algebra_type & algebra();
  const algebra_type & algebra() const;
};
```

Description

The Runge-Kutta Fehlberg 78 method is a standard method for high-precision applications. The method is explicit and fulfills the Error Stepper concept. Step size control is provided but continuous output is not available for this method.

This class derives from explicit_error_stepper_base and inherits its interface via CRTP (current recurring template pattern). Furthermore, it derivs from explicit_error_generic_rk which is a generic Runge-Kutta algorithm with error estimation. For more details see explicit_error_stepper_base and explicit_error_generic_rk.

Template Parameters

1. `typename State`

   The state type.

2. `typename Value = double`

   The value type.

3. `typename Deriv = State`

   The type representing the time derivative of the state.

4. `typename Time = Value`

   The time representing the independent variable - the time.

5. `typename Algebra = range_algebra`

   The algebra type.

6. `typename Operations = default_operations`

   The operations type.

7. `typename Resizer = initially_resizer`

   The resizer policy type.

`runge_kutta_fehlberg78` public construct/copy/destruct

1. `runge_kutta_fehlberg78(const algebra_type & algebra = algebra_type());`

Constructs the runge_kutta_cash_fehlberg78 class. This constructor can be used as a default constructor if the algebra has a default constructor.

Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`runge_kutta_fehlberg78` public member functions

1.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename Err>
  void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                    time_type t, StateOut & out, time_type dt, Err & xerr);
```

This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out-of-place, hence the input is in `in` and the output in `out`. Futhermore, an estimation of the error is stored in `xerr`. `do_step_impl` is used by `explicit_error_stepper_base`.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The result of the error estimation is written in xerr.

2.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  void do_step_impl(System system, const StateIn & in, const DerivIn & dxdt,
                    time_type t, StateOut & out, time_type dt);
```

This method performs one step. The derivative `dxdt` of `in` at the time `t` is passed to the method. The result is updated out-of-place, hence the input is in `in` and the output in `out`. Access to this step functionality is provided by `explicit_stepper_base` and `do_step_impl` should not be called directly.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

3.
```
template<typename StateIn> void adjust_size(const StateIn & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

4.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper if it used without error estimation.

5.
```
order_type stepper_order(void) const;
```

Returns: Returns the order of a step if the stepper is used without error estimation.

6.
```
order_type error_order(void) const;
```

Returns: Returns the order of an error step if the stepper is used without error estimation.

7.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & x, time_type t, time_type dt);
```

This method performs one step. It transforms the result in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ordinary differential equation. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

8.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

9.
```
template<typename System, typename StateInOut, typename DerivIn>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt );
```

The result is updated in place in x. This method is disabled if Time and Deriv are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.

10.
```
template<typename System, typename StateIn, typename StateOut>
  boost::disable_if< boost::is_same< StateIn, time_type >, void >::type
  do_step(System system, const StateIn & in, time_type t, StateOut & out,
          time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. This method is disabled if StateIn and Time are the same type. In this case the method can not be distinguished from other `do_step` variants.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

11.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, const StateIn & in, const DerivIn & dxdt,
          time_type t, StateOut & out, time_type dt);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

This method is disabled if DerivIn and Time are of same type.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.

12.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

The method performs one step with the stepper passed by Stepper and estimates the error. The state of the ODE is updated in-place.

Parameters:

`dt`	The step size.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. x is updated by this method.
`xerr`	The estimation of the error is stored in xerr.

13.
```
template<typename System, typename StateInOut, typename Err>
  void do_step(System system, const StateInOut & x, time_type t, time_type dt,
               Err & xerr);
```

Second version to solve the forwarding problem, can be called with Boost.Range as StateInOut.

14.
```
template<typename System, typename StateInOut, typename DerivIn, typename Err>
  boost::disable_if< boost::is_same< DerivIn, time_type >, void >::type
  do_step(System system, StateInOut & x, const DerivIn & dxdt, time_type t,
          time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. Additionally to the other method the derivative of x is also passed to this method. It is supposed to be used in the following way:

```
sys( x , dxdt , t );
stepper.do_step( sys , x , dxdt , t , dt , xerr );
```

The result is updated in place in x. This method is disabled if Time and DerivIn are of the same type. In this case the method could not be distinguished from other `do_step` versions.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`x`	The state of the ODE which should be solved. After calling do_step the result is updated in x.
`xerr`	The error estimate is stored in xerr.

15.
```
template<typename System, typename StateIn, typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the error is estimated.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

16.
```
template<typename System, typename StateIn, typename DerivIn,
         typename StateOut, typename Err>
  void do_step(System system, const StateIn & in, const DerivIn & dxdt,
               time_type t, StateOut & out, time_type dt, Err & xerr);
```

The method performs one step with the stepper passed by Stepper. The state of the ODE is updated out-of-place. Furthermore, the derivative of x at t is passed to the stepper and the error is estimated. It is supposed to be used in the following way:

```
sys( in , dxdt , t );
stepper.do_step( sys , in , dxdt , t , out , dt );
```

This method is disabled if DerivIn and Time are of same type.

Note

This method does not solve the forwarding problem.

Parameters:

`dt`	The step size.
`dxdt`	The derivative of x at t.
`in`	The state of the ODE which should be solved. in is not modified in this method
`out`	The result of the step is written in out.
`system`	The system function to solve, hence the r.h.s. of the ODE. It must fulfill the Simple System concept.
`t`	The value of the time, at which the step should be performed.
`xerr`	The error estimate.

17.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

18.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

Header <boost/numeric/odeint/stepper/stepper_categories.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      struct stepper_tag;
      struct error_stepper_tag;
      struct explicit_error_stepper_tag;
      struct explicit_error_stepper_fsal_tag;
      struct controlled_stepper_tag;
      struct explicit_controlled_stepper_tag;
      struct explicit_controlled_stepper_fsal_tag;
      struct dense_output_stepper_tag;
      template<typename tag> struct base_tag;

      template<> struct base_tag<stepper_tag>;
      template<> struct base_tag<error_stepper_tag>;
      template<> struct base_tag<explicit_error_stepper_tag>;
      template<> struct base_tag<explicit_error_stepper_fsal_tag>;
      template<> struct base_tag<controlled_stepper_tag>;
      template<> struct base_tag<explicit_controlled_stepper_tag>;
      template<> struct base_tag<explicit_controlled_stepper_fsal_tag>;
      template<> struct base_tag<dense_output_stepper_tag>;
    }
  }
}
```

Struct stepper_tag

boost::numeric::odeint::stepper_tag

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct stepper_tag {
};
```

Struct error_stepper_tag

boost::numeric::odeint::error_stepper_tag

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct error_stepper_tag : public boost::numeric::odeint::stepper_tag {
};
```

Struct explicit_error_stepper_tag

boost::numeric::odeint::explicit_error_stepper_tag

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct explicit_error_stepper_tag :
  public boost::numeric::odeint::error_stepper_tag
{
};
```

Struct explicit_error_stepper_fsal_tag

boost::numeric::odeint::explicit_error_stepper_fsal_tag

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct explicit_error_stepper_fsal_tag :
  public boost::numeric::odeint::error_stepper_tag
{
};
```

Struct controlled_stepper_tag

boost::numeric::odeint::controlled_stepper_tag

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct controlled_stepper_tag {
};
```

Struct explicit_controlled_stepper_tag

boost::numeric::odeint::explicit_controlled_stepper_tag

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct explicit_controlled_stepper_tag :
  public boost::numeric::odeint::controlled_stepper_tag
{
};
```

Struct explicit_controlled_stepper_fsal_tag

boost::numeric::odeint::explicit_controlled_stepper_fsal_tag

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct explicit_controlled_stepper_fsal_tag :
  public boost::numeric::odeint::controlled_stepper_tag
{
};
```

Struct dense_output_stepper_tag

boost::numeric::odeint::dense_output_stepper_tag

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct dense_output_stepper_tag {
};
```

Struct template base_tag

boost::numeric::odeint::base_tag

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>

template<typename tag>
struct base_tag {
};
```

Struct base_tag<stepper_tag>

boost::numeric::odeint::base_tag<stepper_tag>

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct base_tag<stepper_tag> {
  // types
  typedef stepper_tag type;
};
```

Struct base_tag<error_stepper_tag>

boost::numeric::odeint::base_tag<error_stepper_tag>

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct base_tag<error_stepper_tag> {
  // types
  typedef stepper_tag type;
};
```

Struct base_tag<explicit_error_stepper_tag>

boost::numeric::odeint::base_tag<explicit_error_stepper_tag>

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct base_tag<explicit_error_stepper_tag> {
  // types
  typedef stepper_tag type;
};
```

Struct base_tag<explicit_error_stepper_fsal_tag>

boost::numeric::odeint::base_tag<explicit_error_stepper_fsal_tag>

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct base_tag<explicit_error_stepper_fsal_tag> {
  // types
  typedef stepper_tag type;
};
```

Struct base_tag<controlled_stepper_tag>

boost::numeric::odeint::base_tag<controlled_stepper_tag>

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct base_tag<controlled_stepper_tag> {
  // types
  typedef controlled_stepper_tag type;
};
```

Struct base_tag<explicit_controlled_stepper_tag>

boost::numeric::odeint::base_tag<explicit_controlled_stepper_tag>

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct base_tag<explicit_controlled_stepper_tag> {
  // types
  typedef controlled_stepper_tag type;
};
```

Struct base_tag<explicit_controlled_stepper_fsal_tag>

boost::numeric::odeint::base_tag<explicit_controlled_stepper_fsal_tag>

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct base_tag<explicit_controlled_stepper_fsal_tag> {
  // types
  typedef controlled_stepper_tag type;
};
```

Struct base_tag<dense_output_stepper_tag>

boost::numeric::odeint::base_tag<dense_output_stepper_tag>

Synopsis

```
// In header: <boost/numeric/odeint/stepper/stepper_categories.hpp>


struct base_tag<dense_output_stepper_tag> {
  // types
  typedef dense_output_stepper_tag type;
};
```

Header <boost/numeric/odeint/stepper/symplectic_euler.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Coor, typename Momentum = Coor,
               typename Value = double, typename CoorDeriv = Coor,
               typename MomentumDeriv = Coor, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class symplectic_euler;
    }
  }
}
```

Class template symplectic_euler

boost::numeric::odeint::symplectic_euler — Implementation of the symplectic Euler method.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/symplectic_euler.hpp>

template<typename Coor, typename Momentum = Coor, typename Value = double,
         typename CoorDeriv = Coor, typename MomentumDeriv = Coor,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class symplectic_euler : public boost::numeric::odeint::symplectic_nystroem_stepper_base< NumOf↵
Stages, Order, Coor, Momentum, Value, CoorDeriv, MomentumDeriv, Time, Algebra, Operations, Res↵
izer >
{
public:
  // types
  typedef stepper_base_type::algebra_type algebra_type;
  typedef stepper_base_type::value_type   value_type;

  // construct/copy/destruct
  symplectic_euler(const algebra_type & = algebra_type());

  // public member functions
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename CoorInOut, typename MomentumInOut>
    void do_step(System, CoorInOut &, MomentumInOut &, time_type, time_type);
  template<typename System, typename CoorInOut, typename MomentumInOut>
    void do_step(System, const CoorInOut &, const MomentumInOut &, time_type,
                 time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename StateType> void adjust_size(const StateType &);
  const coef_type & coef_a(void) const;
  const coef_type & coef_b(void) const;
  algebra_type & algebra();
  const algebra_type & algebra() const;
};
```

Description

The method is of first order and has one stage. It is described HERE.

Template Parameters

1. `typename Coor`

   The type representing the coordinates q.

2. `typename Momentum = Coor`

   The type representing the coordinates p.

3. `typename Value = double`

   The basic value type. Should be something like float, double or a high-precision type.

4.
```
typename CoorDeriv = Coor
```

The type representing the time derivative of the coordinate dq/dt.

5.
```
typename MomentumDeriv = Coor
```

6.
```
typename Time = Value
```

The type representing the time t.

7.
```
typename Algebra = range_algebra
```

The algebra.

8.
```
typename Operations = default_operations
```

The operations.

9.
```
typename Resizer = initially_resizer
```

The resizer policy.

`symplectic_euler` public construct/copy/destruct

1.
```
symplectic_euler(const algebra_type & algebra = algebra_type());
```

Constructs the `symplectic_euler`. This constructor can be used as a default constructor if the algebra has a default constructor.

Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`symplectic_euler` public member functions

1.
```
order_type order(void) const;
```

Returns: Returns the order of the stepper.

2.
```
template<typename System, typename StateInOut>
  void do_step(System system, const StateInOut & state, time_type t,
               time_type dt);
```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated in-place.

Note

boost::ref or std::ref can be used for the system as well as for the state. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , make_pair( std::ref( q ) , std::ref( p ) ) , t , dt )`.

This method solves the forwarding problem.

Parameters:

`dt`	The time step.
`state`	The state of the ODE. It is a pair of Coor and Momentum. The state is updated in-place, therefore, the new value of the state will be written into this variable.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

3.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & state, time_type t, time_type dt);
```

Same function as above. It differs only in a different const specifier in order to solve the forwarding problem, can be used with Boost.Range.

4.
```
template<typename System, typename CoorInOut, typename MomentumInOut>
  void do_step(System system, CoorInOut & q, MomentumInOut & p, time_type t,
               time_type dt);
```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated in-place.

Note

boost::ref or std::ref can be used for the system. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , q , p , t , dt )`.

This method solves the forwarding problem.

Parameters:

`dt`	The time step.
`p`	The momentum of the ODE. It is updated in-place. Therefore, the new value of the momentum will be written info this variable.
`q`	The coordinate of the ODE. It is updated in-place. Therefore, the new value of the coordinate will be written into this variable.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

5.
```
template<typename System, typename CoorInOut, typename MomentumInOut>
  void do_step(System system, const CoorInOut & q, const MomentumInOut & p,
               time_type t, time_type dt);
```

Same function as do_step(system , q , p , t , dt). It differs only in a different const specifier in order to solve the forwarding problem, can be called with Boost.Range.

6.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated out-of-place.

Note

boost::ref or std::ref can be used for the system. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , x_in , t , x_out , dt )`.

This method NOT solve the forwarding problem.

Parameters:

`dt`	The time step.
`in`	The state of the ODE, which is a pair of coordinate and momentum. The state is updated out-of-place, therefore the new value is written into out
`out`	The new state of the ODE.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

7.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

8.
```
const coef_type & coef_a(void) const;
```

Returns the coefficients a.

9.
```
const coef_type & coef_b(void) const;
```

Returns the coefficients b.

10.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

11.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

Header <boost/numeric/odeint/stepper/symplectic_rkn_sb3a_m4_mclachlan.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Coor, typename Momentum = Coor,
               typename Value = double, typename CoorDeriv = Coor,
               typename MomentumDeriv = Coor, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class symplectic_rkn_sb3a_m4_mclachlan;
    }
  }
}
```

Class template symplectic_rkn_sb3a_m4_mclachlan

boost::numeric::odeint::symplectic_rkn_sb3a_m4_mclachlan — Implementation of the symmetric B3A Runge-Kutta Nystroem method of fifth order.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/symplectic_rkn_sb3a_m4_mclachlan.hpp>

template<typename Coor, typename Momentum = Coor, typename Value = double,
         typename CoorDeriv = Coor, typename MomentumDeriv = Coor,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class symplectic_rkn_sb3a_m4_mclachlan : public boost::numeric::odeint::symplectic_nystroem_step↵
per_base< NumOfStages, Order, Coor, Momentum, Value, CoorDeriv, MomentumDeriv, Time, Algebra, ↵
Operations, Resizer >
{
public:
  // types
  typedef stepper_base_type::algebra_type algebra_type;
  typedef stepper_base_type::value_type   value_type;

  // construct/copy/destruct
  symplectic_rkn_sb3a_m4_mclachlan(const algebra_type & = algebra_type());

  // public member functions
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename CoorInOut, typename MomentumInOut>
    void do_step(System, CoorInOut &, MomentumInOut &, time_type, time_type);
  template<typename System, typename CoorInOut, typename MomentumInOut>
    void do_step(System, const CoorInOut &, const MomentumInOut &, time_type,
                 time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename StateType> void adjust_size(const StateType &);
  const coef_type & coef_a(void) const;
  const coef_type & coef_b(void) const;
  algebra_type & algebra();
  const algebra_type & algebra() const;
};
```

Description

The method is of fourth order and has five stages. It is described HERE. This method can be used with multiprecision types since the coefficients are defined analytically.

ToDo: add reference to paper.

Template Parameters

1. `typename Coor`

   The type representing the coordinates q.

2. `typename Momentum = Coor`

   The type representing the coordinates p.

3. `typename Value = double`

   The basic value type. Should be something like float, double or a high-precision type.

4. `typename CoorDeriv = Coor`

   The type representing the time derivative of the coordinate dq/dt.

5. `typename MomentumDeriv = Coor`

6. `typename Time = Value`

   The type representing the time t.

7. `typename Algebra = range_algebra`

   The algebra.

8. `typename Operations = default_operations`

   The operations.

9. `typename Resizer = initially_resizer`

   The resizer policy.

`symplectic_rkn_sb3a_m4_mclachlan` public construct/copy/destruct

1. `symplectic_rkn_sb3a_m4_mclachlan(const algebra_type & algebra = algebra_type());`

   Constructs the `symplectic_rkn_sb3a_m4_mclachlan`. This constructor can be used as a default constructor if the algebra has a default constructor.

   Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`symplectic_rkn_sb3a_m4_mclachlan` public member functions

1. `order_type order(void) const;`

   Returns: Returns the order of the stepper.

2. ```
   template<typename System, typename StateInOut>
     void do_step(System system, const StateInOut & state, time_type t,
                  time_type dt);
   ```
   ```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated in-place.

Note

boost::ref or std::ref can be used for the system as well as for the state. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , make_pair( std::ref( q ) , std::ref( p ) ) , t , dt )`.

This method solves the forwarding problem.

Parameters:

`dt`	The time step.
`state`	The state of the ODE. It is a pair of Coor and Momentum. The state is updated in-place, therefore, the new value of the state will be written into this variable.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

3.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & state, time_type t, time_type dt);
```

Same function as above. It differs only in a different const specifier in order to solve the forwarding problem, can be used with Boost.Range.

4.
```
template<typename System, typename CoorInOut, typename MomentumInOut>
  void do_step(System system, CoorInOut & q, MomentumInOut & p, time_type t,
               time_type dt);
```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated in-place.

Note

boost::ref or std::ref can be used for the system. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , q , p , t , dt )`.

This method solves the forwarding problem.

Parameters:

`dt`	The time step.
`p`	The momentum of the ODE. It is updated in-place. Therefore, the new value of the momentum will be written info this variable.
`q`	The coordinate of the ODE. It is updated in-place. Therefore, the new value of the coordinate will be written into this variable.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

5.
```
template<typename System, typename CoorInOut, typename MomentumInOut>
  void do_step(System system, const CoorInOut & q, const MomentumInOut & p,
               time_type t, time_type dt);
```

Same function as do_step(system , q , p , t , dt). It differs only in a different const specifier in order to solve the forwarding problem, can be called with Boost.Range.

6.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated out-of-place.

Note

boost::ref or std::ref can be used for the system. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , x_in , t , x_out , dt )`.

This method NOT solve the forwarding problem.

Parameters:

`dt`	The time step.
`in`	The state of the ODE, which is a pair of coordinate and momentum. The state is updated out-of-place, therefore the new value is written into out
`out`	The new state of the ODE.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

7.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

8.
```
const coef_type & coef_a(void) const;
```

Returns the coefficients a.

9.
```
const coef_type & coef_b(void) const;
```

Returns the coefficients b.

10.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

11.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

Header <boost/numeric/odeint/stepper/symplectic_rkn_sb3a_mclachlan.hpp>

```
namespace boost {
  namespace numeric {
    namespace odeint {
      template<typename Coor, typename Momentum = Coor,
               typename Value = double, typename CoorDeriv = Coor,
               typename MomentumDeriv = Coor, typename Time = Value,
               typename Algebra = range_algebra,
               typename Operations = default_operations,
               typename Resizer = initially_resizer>
        class symplectic_rkn_sb3a_mclachlan;
    }
  }
}
```

Class template symplectic_rkn_sb3a_mclachlan

boost::numeric::odeint::symplectic_rkn_sb3a_mclachlan — Implement of the symmetric B3A method of Runge-Kutta-Nystroem method of sixth order.

Synopsis

```
// In header: <boost/numeric/odeint/stepper/symplectic_rkn_sb3a_mclachlan.hpp>

template<typename Coor, typename Momentum = Coor, typename Value = double,
         typename CoorDeriv = Coor, typename MomentumDeriv = Coor,
         typename Time = Value, typename Algebra = range_algebra,
         typename Operations = default_operations,
         typename Resizer = initially_resizer>
class symplectic_rkn_sb3a_mclachlan : public boost::numeric::odeint::symplectic_nystroem_step↵
per_base< NumOfStages, Order, Coor, Momentum, Value, CoorDeriv, MomentumDeriv, Time, Algebra, ↵
Operations, Resizer >
{
public:
  // types
  typedef stepper_base_type::algebra_type algebra_type;
  typedef stepper_base_type::value_type   value_type;

  // construct/copy/destruct
  symplectic_rkn_sb3a_mclachlan(const algebra_type & = algebra_type());

  // public member functions
  order_type order(void) const;
  template<typename System, typename StateInOut>
    void do_step(System, const StateInOut &, time_type, time_type);
  template<typename System, typename StateInOut>
    void do_step(System, StateInOut &, time_type, time_type);
  template<typename System, typename CoorInOut, typename MomentumInOut>
    void do_step(System, CoorInOut &, MomentumInOut &, time_type, time_type);
  template<typename System, typename CoorInOut, typename MomentumInOut>
    void do_step(System, const CoorInOut &, const MomentumInOut &, time_type,
                 time_type);
  template<typename System, typename StateIn, typename StateOut>
    void do_step(System, const StateIn &, time_type, StateOut &, time_type);
  template<typename StateType> void adjust_size(const StateType &);
  const coef_type & coef_a(void) const;
  const coef_type & coef_b(void) const;
  algebra_type & algebra();
  const algebra_type & algebra() const;
};
```

Description

The method is of fourth order and has six stages. It is described HERE. This method cannot be used with multiprecision types since the coefficients are not defined analytically.

ToDo Add reference to the paper.

Template Parameters

1. ```
   typename Coor
   ```

   The type representing the coordinates q.

2. ```
   typename Momentum = Coor
   ```

   The type representing the coordinates p.

3. ```
   typename Value = double
   ```

   The basic value type. Should be something like float, double or a high-precision type.

4. ```
   typename CoorDeriv = Coor
   ```

   The type representing the time derivative of the coordinate dq/dt.

5. ```
   typename MomentumDeriv = Coor
   ```

6. ```
   typename Time = Value
   ```

   The type representing the time t.

7. ```
   typename Algebra = range_algebra
   ```

   The algebra.

8. ```
   typename Operations = default_operations
   ```

   The operations.

9. ```
   typename Resizer = initially_resizer
   ```

   The resizer policy.

`symplectic_rkn_sb3a_mclachlan` public construct/copy/destruct

1. ```
   symplectic_rkn_sb3a_mclachlan(const algebra_type & algebra = algebra_type());
   ```

   Constructs the `symplectic_rkn_sb3a_mclachlan`. This constructor can be used as a default constructor if the algebra has a default constructor.

   Parameters: `algebra` A copy of algebra is made and stored inside `explicit_stepper_base`.

`symplectic_rkn_sb3a_mclachlan` public member functions

1. ```
   order_type order(void) const;
   ```

   Returns: Returns the order of the stepper.

2. ```
   template<typename System, typename StateInOut>
     void do_step(System system, const StateInOut & state, time_type t,
                  time_type dt);
   ```

   This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated in-place.

Note

boost::ref or std::ref can be used for the system as well as for the state. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , make_pair( std::ref( q ) , std::ref( p ) ) , t , dt )`.

This method solves the forwarding problem.

Parameters:

`dt`	The time step.
`state`	The state of the ODE. It is a pair of Coor and Momentum. The state is updated in-place, therefore, the new value of the state will be written into this variable.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

3.
```
template<typename System, typename StateInOut>
  void do_step(System system, StateInOut & state, time_type t, time_type dt);
```

Same function as above. It differs only in a different const specifier in order to solve the forwarding problem, can be used with Boost.Range.

4.
```
template<typename System, typename CoorInOut, typename MomentumInOut>
  void do_step(System system, CoorInOut & q, MomentumInOut & p, time_type t,
               time_type dt);
```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated in-place.

Note

boost::ref or std::ref can be used for the system. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , q , p , t , dt )`.

This method solves the forwarding problem.

Parameters:

`dt`	The time step.
`p`	The momentum of the ODE. It is updated in-place. Therefore, the new value of the momentum will be written info this variable.
`q`	The coordinate of the ODE. It is updated in-place. Therefore, the new value of the coordinate will be written into this variable.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

5.
```
template<typename System, typename CoorInOut, typename MomentumInOut>
  void do_step(System system, const CoorInOut & q, const MomentumInOut & p,
               time_type t, time_type dt);
```

Same function as do_step(system , q , p , t , dt). It differs only in a different const specifier in order to solve the forwarding problem, can be called with Boost.Range.

6.
```
template<typename System, typename StateIn, typename StateOut>
  void do_step(System system, const StateIn & in, time_type t, StateOut & out,
               time_type dt);
```

This method performs one step. The system can be either a pair of two function object describing the momentum part and the coordinate part or one function object describing only the momentum part. In this case the coordinate is assumed to be trivial dq/dt = p. The state is updated out-of-place.

Note

boost::ref or std::ref can be used for the system. So, it is correct to write `stepper.do_step( make_pair( std::ref( fq ) , std::ref( fp ) ) , x_in , t , x_out , dt )`.

This method NOT solve the forwarding problem.

Parameters:

`dt`	The time step.
`in`	The state of the ODE, which is a pair of coordinate and momentum. The state is updated out-of-place, therefore the new value is written into out
`out`	The new state of the ODE.
`system`	The system, can be represented as a pair of two function object or one function object. See above.
`t`	The time of the ODE. It is not advanced by this method.

7.
```
template<typename StateType> void adjust_size(const StateType & x);
```

Adjust the size of all temporaries in the stepper manually.

Parameters: `x` A state from which the size of the temporaries to be resized is deduced.

8.
```
const coef_type & coef_a(void) const;
```

Returns the coefficients a.

9.
```
const coef_type & coef_b(void) const;
```

Returns the coefficients b.

10.
```
algebra_type & algebra();
```

Returns: A reference to the algebra which is held by this class.

11.
```
const algebra_type & algebra() const;
```

Returns: A const reference to the algebra which is held by this class.

Indexes

Class Index

A

adams_bashforth
Class template adams_bashforth, 110
adams_bashforth_moulton
Class template adams_bashforth_moulton, 116
adams_moulton
Class template adams_moulton, 120
algebra_stepper_base
Class template adams_bashforth, 110
Class template algebra_stepper_base, 122
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_stepper_base, 142
Class template symplectic_nystroem_stepper_base, 148

B

base_tag
Struct base_tag<controlled_stepper_tag>, 261
Struct base_tag<dense_output_stepper_tag>, 262
Struct base_tag<error_stepper_tag>, 260
Struct base_tag<explicit_controlled_stepper_fsal_tag>, 262
Struct base_tag<explicit_controlled_stepper_tag>, 261
Struct base_tag<explicit_error_stepper_fsal_tag>, 261
Struct base_tag<explicit_error_stepper_tag>, 260
Struct base_tag<stepper_tag>, 260
Struct template base_tag, 260
bulirsch_stoer
Class template bulirsch_stoer, 154
bulirsch_stoer_dense_out
Class template bulirsch_stoer_dense_out, 159

C

controlled_runge_kutta
Class template controlled_runge_kutta, 166
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
controlled_stepper_tag
Struct base_tag<controlled_stepper_tag>, 261
Struct controlled_stepper_tag, 259
Struct explicit_controlled_stepper_fsal_tag, 259
Struct explicit_controlled_stepper_tag, 259
controller_factory
Generation functions, 67

D

default_error_checker
Class template default_error_checker, 165
default_operations
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54_classic, 235
Custom Runge-Kutta steppers, 65

default_rosenbrock_coefficients
Struct template default_rosenbrock_coefficients, 209
dense_output_runge_kutta
Class template dense_output_runge_kutta, 176
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
dense_output_stepper_tag
Struct base_tag<dense_output_stepper_tag>, 262
Struct dense_output_stepper_tag, 259

E

error_stepper_tag
Struct base_tag<error_stepper_tag>, 260
Struct error_stepper_tag, 258
Struct explicit_error_stepper_fsal_tag, 258
Struct explicit_error_stepper_tag, 258
euler
Class template euler, 183
explicit_controlled_stepper_fsal_tag
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Struct base_tag<explicit_controlled_stepper_fsal_tag>, 262
Struct explicit_controlled_stepper_fsal_tag, 259
explicit_controlled_stepper_tag
Struct base_tag<explicit_controlled_stepper_tag>, 261
Struct explicit_controlled_stepper_tag, 259
explicit_error_generic_rk
Class template explicit_error_generic_rk, 188
Class template runge_kutta_cash_karp54, 228
Class template runge_kutta_fehlberg78, 251
explicit_error_stepper_base
Class template explicit_error_generic_rk, 188
Class template explicit_error_stepper_base, 124
Class template runge_kutta_cash_karp54_classic, 235
explicit_error_stepper_fsal_base
Class template explicit_error_stepper_fsal_base, 132
Class template runge_kutta_dopri5, 242
explicit_error_stepper_fsal_tag
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Struct base_tag<explicit_error_stepper_fsal_tag>, 261
Struct explicit_error_stepper_fsal_tag, 258
explicit_error_stepper_tag
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Struct base_tag<explicit_error_stepper_tag>, 260
Struct explicit_error_stepper_tag, 258
explicit_generic_rk
Class template explicit_generic_rk, 196
Class template runge_kutta4, 218
Custom Runge-Kutta steppers, 65
explicit_stepper_base
Class template euler, 183
Class template explicit_generic_rk, 196
Class template explicit_stepper_base, 142
Class template modified_midpoint, 203
Class template runge_kutta4_classic, 223

G

get_controller

Generation functions, 67
gsl_vector_iterator
GSL Vector, 74

I

implicit_euler
Class template implicit_euler, 201
initially_resizer
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54_classic, 235
Custom Runge-Kutta steppers, 65
is_resizeable
Boost.Ublas, 78
GSL Vector, 74
std::list, 73
Using the container interface, 72

M

modified_midpoint
Class template modified_midpoint, 203
modified_midpoint_dense_out
Class template modified_midpoint_dense_out, 207

N

null_observer
Struct null_observer, 107

O

observer_collection
Class template observer_collection, 108

R

range_algebra
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54_classic, 235
Custom Runge-Kutta steppers, 65
resize_impl
GSL Vector, 74
std::list, 73
rosenbrock4
Class template rosenbrock4, 210
rosenbrock4_controller
Class template rosenbrock4_controller, 213
rosenbrock4_dense_output
Class template rosenbrock4_dense_output, 215
runge_kutta4
Class template runge_kutta4, 218
runge_kutta4_classic
Class template runge_kutta4_classic, 223
runge_kutta_cash_karp54
Class template runge_kutta_cash_karp54, 228
runge_kutta_cash_karp54_classic
Class template runge_kutta_cash_karp54_classic, 235
runge_kutta_fehlberg78
Class template runge_kutta_fehlberg78, 251

S

same_size_impl
GSL Vector, 74
std::list, 73
state_wrapper
GSL Vector, 74
stepper_tag
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Struct base_tag<stepper_tag>, 260
Struct error_stepper_tag, 258
Struct stepper_tag, 258
symplectic_euler
Class template symplectic_euler, 263
symplectic_nystroem_stepper_base
Class template symplectic_euler, 263
Class template symplectic_nystroem_stepper_base, 148
Class template symplectic_rkn_sb3a_m4_mclachlan, 267
Class template symplectic_rkn_sb3a_mclachlan, 272
symplectic_rkn_sb3a_m4_mclachlan
Class template symplectic_rkn_sb3a_m4_mclachlan, 267
symplectic_rkn_sb3a_mclachlan
Class template symplectic_rkn_sb3a_mclachlan, 272

V

vector_space_reduce
Point type, 79

Function Index

A

adjust_size
Class template adams_bashforth, 110, 113
Class template adams_bashforth_moulton, 116, 118
Class template adams_moulton, 120-121
Class template bulirsch_stoer, 154, 157
Class template bulirsch_stoer_dense_out, 159, 162
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 174
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177-178
Class template euler, 183, 185-186
Class template explicit_error_generic_rk, 188, 191
Class template explicit_error_stepper_base, 124, 130
Class template explicit_error_stepper_fsal_base, 132, 139
Class template explicit_generic_rk, 196, 198
Class template explicit_stepper_base, 142, 146
Class template implicit_euler, 201
Class template modified_midpoint, 203-204
Class template modified_midpoint_dense_out, 207-208
Class template rosenbrock4, 210-211
Class template rosenbrock4_controller, 213-214
Class template rosenbrock4_dense_output, 215-216
Class template runge_kutta4, 218, 220
Class template runge_kutta4_classic, 223, 225-226
Class template runge_kutta_cash_karp54, 228, 230
Class template runge_kutta_cash_karp54_classic, 235, 237

Class template runge_kutta_dopri5, 242, 245
Class template runge_kutta_fehlberg78, 251, 253
Class template symplectic_euler, 263, 266
Class template symplectic_nystroem_stepper_base, 148, 152
Class template symplectic_rkn_sb3a_m4_mclachlan, 267, 270
Class template symplectic_rkn_sb3a_mclachlan, 272, 275
advance
GSL Vector, 74
algebra
Class template adams_bashforth, 110, 114
Class template adams_moulton, 120-122
Class template algebra_stepper_base, 122-123
Class template euler, 183, 186-187
Class template explicit_error_generic_rk, 188, 194
Class template explicit_error_stepper_base, 124, 130-131
Class template explicit_error_stepper_fsal_base, 132, 140
Class template explicit_generic_rk, 196, 200
Class template explicit_stepper_base, 142, 146
Class template modified_midpoint, 203, 206
Class template runge_kutta4, 218, 221
Class template runge_kutta4_classic, 223, 226
Class template runge_kutta_cash_karp54, 228, 234
Class template runge_kutta_cash_karp54_classic, 235, 241
Class template runge_kutta_dopri5, 242, 249
Class template runge_kutta_fehlberg78, 251, 257
Class template symplectic_euler, 263, 266
Class template symplectic_nystroem_stepper_base, 148, 152
Class template symplectic_rkn_sb3a_m4_mclachlan, 267, 270
Class template symplectic_rkn_sb3a_mclachlan, 272, 275

C

calculate_finite_difference
Class template bulirsch_stoer_dense_out, 159, 163
calc_state
Class template bulirsch_stoer_dense_out, 159, 162
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177-178
Class template euler, 183, 185
Class template rosenbrock4, 210-211
Class template rosenbrock4_dense_output, 215-216
Class template runge_kutta_dopri5, 242, 245

D

decrement
GSL Vector, 74
do_step
Class template adams_bashforth, 110, 112-113
Class template adams_bashforth_moulton, 116, 118
Class template adams_moulton, 120-121
Class template bulirsch_stoer_dense_out, 159, 161
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177-178
Class template euler, 183, 185-186
Class template explicit_error_generic_rk, 188, 191-194
Class template explicit_error_stepper_base, 124, 127-130
Class template explicit_error_stepper_fsal_base, 132-133, 136-139
Class template explicit_generic_rk, 196, 198-199

Class template explicit_stepper_base, 142, 145-146
Class template implicit_euler, 201
Class template modified_midpoint, 203-205
Class template modified_midpoint_dense_out, 207-208
Class template rosenbrock4, 210-211
Class template rosenbrock4_dense_output, 215-216
Class template runge_kutta4, 218, 220-221
Class template runge_kutta4_classic, 223, 225-226
Class template runge_kutta_cash_karp54, 228, 231-233
Class template runge_kutta_cash_karp54_classic, 235, 238-241
Class template runge_kutta_dopri5, 242, 245-248
Class template runge_kutta_fehlberg78, 251, 254-256
Class template symplectic_euler, 263-265
Class template symplectic_nystroem_stepper_base, 148, 150-151
Class template symplectic_rkn_sb3a_m4_mclachlan, 267-270
Class template symplectic_rkn_sb3a_mclachlan, 272-274
Custom steppers, 63
Error Stepper, 89
Explicit steppers, 50
do_step_impl
Class template adams_bashforth, 110, 114
Class template euler, 183-184
Class template explicit_error_generic_rk, 188, 190-191
Class template explicit_error_stepper_base, 125
Class template explicit_error_stepper_fsal_base, 133
Class template explicit_generic_rk, 196, 198
Class template explicit_stepper_base, 142
Class template modified_midpoint, 203-204
Class template runge_kutta4, 218-219
Class template runge_kutta4_classic, 223-224
Class template runge_kutta_cash_karp54, 228, 230
Class template runge_kutta_cash_karp54_classic, 235, 237
Class template runge_kutta_dopri5, 242, 244
Class template runge_kutta_fehlberg78, 251, 253
Class template symplectic_nystroem_stepper_base, 148, 152
do_step_v1
Class template explicit_error_stepper_base, 124, 131
Class template explicit_error_stepper_fsal_base, 132, 140
Class template explicit_stepper_base, 142, 147
do_step_v5
Class template explicit_error_stepper_base, 124, 131
Class template explicit_error_stepper_fsal_base, 132, 140

E

end_iterator
GSL Vector, 74
error
Class template default_error_checker, 165-166
Class template rosenbrock4_controller, 213

F

f
Custom steppers, 63
Define the ODE, 10
Define the system function, 14
Explicit steppers, 50
Implicit solvers, 53

Implicit System, 87
Overview, 3
Short Example, 8
Stiff systems, 20
Symplectic solvers, 51
Symplectic System, 85
System, 85

G

get_current_deriv
Class template bulirsch_stoer_dense_out, 159, 164
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180, 182
get_current_state
Class template bulirsch_stoer_dense_out, 159, 163
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177, 179
Class template rosenbrock4_dense_output, 215-216
get_old_deriv
Class template bulirsch_stoer_dense_out, 159, 164
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180, 182
get_old_state
Class template bulirsch_stoer_dense_out, 159, 163-164
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180, 182
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177, 179
Class template rosenbrock4_dense_output, 215-216
gsl_vector_iterator
GSL Vector, 74

I

increment
GSL Vector, 74
initialize
Class template adams_bashforth, 110, 113
Class template adams_bashforth_moulton, 116, 118-119
Class template bulirsch_stoer_dense_out, 159, 161
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 174
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177-178
Class template explicit_error_stepper_fsal_base, 132, 139
Class template rosenbrock4_dense_output, 215
Class template runge_kutta_dopri5, 242, 249
Using boost::range, 81
Using steppers, 56
initializing_stepper
Class template adams_bashforth, 110, 114
integrate
Function template integrate, 101-102
Header < boost/numeric/odeint/integrate/integrate.hpp >, 101
integrate_adaptive
Function template integrate_adaptive, 103
Header < boost/numeric/odeint/integrate/integrate_adaptive.hpp >, 102
Parameter studies, 42
integrate_const
Ensembles of oscillators, 25
Function template integrate_const, 104
Header < boost/numeric/odeint/integrate/integrate_const.hpp >, 104
Integration with Constant Step Size, 11

Large oscillator chains, 40
Using OpenCL via VexCL, 45
integrate_n_steps
Chaotic systems and Lyapunov exponents, 17
Function template integrate_n_steps, 105
Header < boost/numeric/odeint/integrate/integrate_n_steps.hpp >, 105
integrate_times
Function template integrate_times, 107
Header < boost/numeric/odeint/integrate/integrate_times.hpp >, 106
iter
GSL Vector, 74

M

max
Adaptive step size algorithms, 54
Controlled steppers, 54

O

observers
Class template observer_collection, 108
ode
Binding member functions, 83

P

prepare_dense_output
Class template bulirsch_stoer_dense_out, 159, 163
Class template rosenbrock4, 210-211

R

range_begin
GSL Vector, 74
range_end
GSL Vector, 74
reset
Class template adams_bashforth, 110, 114
Class template bulirsch_stoer, 154, 157
Class template bulirsch_stoer_dense_out, 159, 162
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 174
Class template explicit_error_stepper_fsal_base, 132, 139
Class template runge_kutta_dopri5, 242, 249
Ensembles of oscillators, 25
resize
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template modified_midpoint_dense_out, 207-208
GSL Vector, 74
std::list, 73
resize_dpdt
Class template symplectic_nystroem_stepper_base, 148, 152
resize_dqdt
Class template symplectic_nystroem_stepper_base, 148, 152
resize_dxdt_tmp_impl
Class template runge_kutta_dopri5, 242, 250
resize_impl
Class template adams_bashforth, 110, 114
Class template adams_moulton, 120, 122
Class template bulirsch_stoer, 154, 157
Class template bulirsch_stoer_dense_out, 159, 163

Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177, 179
Class template explicit_error_generic_rk, 188, 195
Class template explicit_error_stepper_base, 124, 131
Class template explicit_error_stepper_fsal_base, 132, 140
Class template explicit_generic_rk, 196, 200
Class template explicit_stepper_base, 142, 147
Class template implicit_euler, 201
Class template modified_midpoint, 203, 206
Class template rosenbrock4, 210-211
Class template rosenbrock4_dense_output, 215-216
Class template runge_kutta4_classic, 223, 227
Class template runge_kutta_cash_karp54_classic, 235, 241
resize_k_x_tmp_impl
Class template runge_kutta_dopri5, 242, 250
resize_m_dxdt
Class template bulirsch_stoer, 154, 157
resize_m_dxdt_impl
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 175
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
resize_m_dxdt_new_impl
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 175
resize_m_xerr
Class template rosenbrock4_controller, 213-214
resize_m_xerr_impl
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 175
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
resize_m_xnew
Class template bulirsch_stoer, 154, 157
Class template rosenbrock4_controller, 213-214
resize_m_xnew_impl
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 175
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
resize_x_err
Class template rosenbrock4, 210, 212

S

same_size
GSL Vector, 74
std::list, 73
set_steps
Class template modified_midpoint, 203-204
Class template modified_midpoint_dense_out, 207-208
solve
Class template implicit_euler, 201
stepper
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 174
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
Class template explicit_error_stepper_base, 124, 131
Class template explicit_error_stepper_fsal_base, 132, 140
Class template explicit_stepper_base, 142, 146
Class template rosenbrock4_controller, 213-214
Short Example, 8
step_storage
Class template adams_bashforth, 110, 113
sys
Large oscillator chains, 40
Steppers, 50

system
Parameter studies, 42

T

toggle_current_state
Class template bulirsch_stoer_dense_out, 159, 164
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180, 182
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177, 179
Class template rosenbrock4_dense_output, 215-216
try_step
Class template bulirsch_stoer, 154, 156-157
Class template bulirsch_stoer_dense_out, 159, 161
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171-173
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167-169
Class template rosenbrock4_controller, 213-214
try_step_v1
Class template bulirsch_stoer, 154, 158
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 175
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170

Index

A

Acknowledgments
example, 100
adams_bashforth
Class template adams_bashforth, 110
adams_bashforth_moulton
Class template adams_bashforth_moulton, 116
adams_moulton
Class template adams_moulton, 120
Adaptive step size algorithms
max, 54
adjust_size
Class template adams_bashforth, 110, 113
Class template adams_bashforth_moulton, 116, 118
Class template adams_moulton, 120-121
Class template bulirsch_stoer, 154, 157
Class template bulirsch_stoer_dense_out, 159, 162
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 174
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177-178
Class template euler, 183, 185-186
Class template explicit_error_generic_rk, 188, 191
Class template explicit_error_stepper_base, 124, 130
Class template explicit_error_stepper_fsal_base, 132, 139
Class template explicit_generic_rk, 196, 198
Class template explicit_stepper_base, 142, 146
Class template implicit_euler, 201
Class template modified_midpoint, 203-204
Class template modified_midpoint_dense_out, 207-208
Class template rosenbrock4, 210-211
Class template rosenbrock4_controller, 213-214
Class template rosenbrock4_dense_output, 215-216
Class template runge_kutta4, 218, 220
Class template runge_kutta4_classic, 223, 225-226

Class template runge_kutta_cash_karp54, 228, 230
Class template runge_kutta_cash_karp54_classic, 235, 237
Class template runge_kutta_dopri5, 242, 245
Class template runge_kutta_fehlberg78, 251, 253
Class template symplectic_euler, 263, 266
Class template symplectic_nystroem_stepper_base, 148, 152
Class template symplectic_rkn_sb3a_m4_mclachlan, 267, 270
Class template symplectic_rkn_sb3a_mclachlan, 272, 275
advance
GSL Vector, 74
algebra
Class template adams_bashforth, 110, 114
Class template adams_moulton, 120-122
Class template algebra_stepper_base, 122-123
Class template euler, 183, 186-187
Class template explicit_error_generic_rk, 188, 194
Class template explicit_error_stepper_base, 124, 130-131
Class template explicit_error_stepper_fsal_base, 132, 140
Class template explicit_generic_rk, 196, 200
Class template explicit_stepper_base, 142, 146
Class template modified_midpoint, 203, 206
Class template runge_kutta4, 218, 221
Class template runge_kutta4_classic, 223, 226
Class template runge_kutta_cash_karp54, 228, 234
Class template runge_kutta_cash_karp54_classic, 235, 241
Class template runge_kutta_dopri5, 242, 249
Class template runge_kutta_fehlberg78, 251, 257
Class template symplectic_euler, 263, 266
Class template symplectic_nystroem_stepper_base, 148, 152
Class template symplectic_rkn_sb3a_m4_mclachlan, 267, 270
Class template symplectic_rkn_sb3a_mclachlan, 272, 275
algebra_stepper_base
Class template adams_bashforth, 110
Class template algebra_stepper_base, 122
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_stepper_base, 142
Class template symplectic_nystroem_stepper_base, 148
algebra_stepper_base_type
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_stepper_base, 142
Class template symplectic_nystroem_stepper_base, 148
algebra_type
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116
Class template adams_moulton, 120
Class template algebra_stepper_base, 122
Class template bulirsch_stoer, 154
Class template bulirsch_stoer_dense_out, 159
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Class template default_error_checker, 165
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template euler, 183
Class template explicit_error_generic_rk, 188
Class template explicit_error_stepper_base, 124

Class template explicit_error_stepper_fsal_base, 132
Class template explicit_generic_rk, 196
Class template explicit_stepper_base, 142
Class template modified_midpoint, 203
Class template modified_midpoint_dense_out, 207
Class template runge_kutta4, 218
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54, 228
Class template runge_kutta_cash_karp54_classic, 235
Class template runge_kutta_dopri5, 242
Class template runge_kutta_fehlberg78, 251
Class template symplectic_euler, 263
Class template symplectic_nystroem_stepper_base, 148
Class template symplectic_rkn_sb3a_m4_mclachlan, 267
Class template symplectic_rkn_sb3a_mclachlan, 272
Custom Runge-Kutta steppers, 65
All examples
example, 47

B

base_tag
Struct base_tag<controlled_stepper_tag>, 261
Struct base_tag<dense_output_stepper_tag>, 262
Struct base_tag<error_stepper_tag>, 260
Struct base_tag<explicit_controlled_stepper_fsal_tag>, 262
Struct base_tag<explicit_controlled_stepper_tag>, 261
Struct base_tag<explicit_error_stepper_fsal_tag>, 261
Struct base_tag<explicit_error_stepper_tag>, 260
Struct base_tag<stepper_tag>, 260
Struct template base_tag, 260
base_type
Class template explicit_stepper_base, 142
Binding member functions
ode, 83
book
Define the system function, 14
Boost.Ublas
example, 78
is_resizeable, 78
state_type, 78
type, 78
bulirsch_stoer
Class template bulirsch_stoer, 154
bulirsch_stoer_dense_out
Class template bulirsch_stoer_dense_out, 159

C

calculate_finite_difference
Class template bulirsch_stoer_dense_out, 159, 163
calc_state
Class template bulirsch_stoer_dense_out, 159, 162
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177-178
Class template euler, 183, 185
Class template rosenbrock4, 210-211
Class template rosenbrock4_dense_output, 215-216
Class template runge_kutta_dopri5, 242, 245

Chaotic systems and Lyapunov exponents
equations, 17
example, 17
integrate_n_steps, 17
pre-conditions, 17
snippet, 17
state_type, 17
Class template adams_bashforth
adams_bashforth, 110
adjust_size, 110, 113
algebra, 110, 114
algebra_stepper_base, 110
algebra_type, 110
deriv_type, 110
do_step, 110, 112-113
do_step_impl, 110, 114
equations, 112-114
initialize, 110, 113
initializing_stepper, 110, 114
initializing_stepper_type, 110
operations_type, 110
order_type, 110
pre-conditions, 111, 113
reset, 110, 114
resizer_type, 110
resize_impl, 110, 114
state_type, 110
stepper_category, 110
step_storage, 110, 113
step_storage_type, 110
time_type, 110
value_type, 110
wrapped_deriv_type, 110
wrapped_state_type, 110
Class template adams_bashforth_moulton
adams_bashforth_moulton, 116
adjust_size, 116, 118
algebra_type, 116
deriv_type, 116
do_step, 116, 118
equations, 118-119
initialize, 116, 118-119
operations_type, 116
order_type, 116
pre-conditions, 116, 118
resizer_type, 116
state_type, 116
stepper_category, 116
time_type, 116
value_type, 116
wrapped_deriv_type, 116
wrapped_state_type, 116
Class template adams_moulton
adams_moulton, 120
adjust_size, 120-121
algebra, 120-122
algebra_type, 120
deriv_type, 120

do_step, 120-121
operations_type, 120
order_type, 120
resizer_type, 120
resize_impl, 120, 122
state_type, 120
stepper_category, 120
stepper_type, 120
step_storage_type, 120
time_type, 120
value_type, 120
wrapped_deriv_type, 120
wrapped_state_type, 120
Class template algebra_stepper_base
algebra, 122-123
algebra_stepper_base, 122
algebra_type, 122
operations_type, 122
Class template bulirsch_stoer
adjust_size, 154, 157
algebra_type, 154
bulirsch_stoer, 154
deriv_type, 154
operations_type, 154
reset, 154, 157
resizer_type, 154
resize_impl, 154, 157
resize_m_dxdt, 154, 157
resize_m_xnew, 154, 157
state_type, 154
time_type, 154
try_step, 154, 156-157
try_step_v1, 154, 158
value_type, 154
Class template bulirsch_stoer_dense_out
adjust_size, 159, 162
algebra_type, 159
bulirsch_stoer_dense_out, 159
calculate_finite_difference, 159, 163
calc_state, 159, 162
deriv_type, 159
do_step, 159, 161
equations, 162
get_current_deriv, 159, 164
get_current_state, 159, 163
get_old_deriv, 159, 164
get_old_state, 159, 163-164
initialize, 159, 161
operations_type, 159
pre-conditions, 159, 162-163
prepare_dense_output, 159, 163
reset, 159, 162
resizer_type, 159
resize_impl, 159, 163
state_type, 159
stepper_category, 159
time_type, 159
toggle_current_state, 159, 164

try_step, 159, 161
value_type, 159
Class template controlled_runge_kutta
controlled_runge_kutta, 166
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>
adjust_size, 171, 174
algebra_type, 171
controlled_runge_kutta, 171
deriv_type, 171
error_checker_type, 171
explicit_error_stepper_fsal_tag, 171
initialize, 171, 174
operations_type, 171
reset, 171, 174
resizer_type, 171
resize_m_dxdt_impl, 171, 175
resize_m_dxdt_new_impl, 171, 175
resize_m_xerr_impl, 171, 175
resize_m_xnew_impl, 171, 175
state_type, 171
stepper, 171, 174
stepper_category, 171
stepper_type, 171
time_type, 171
try_step, 171-173
try_step_v1, 171, 175
value_type, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>
adjust_size, 167, 170
algebra_type, 167
controlled_runge_kutta, 167
deriv_type, 167
error_checker_type, 167
explicit_error_stepper_tag, 167
operations_type, 167
resizer_type, 167
resize_m_dxdt_impl, 167, 170
resize_m_xerr_impl, 167, 170
resize_m_xnew_impl, 167, 170
state_type, 167
stepper, 167, 170
stepper_category, 167
stepper_type, 167
time_type, 167
try_step, 167-169
try_step_v1, 167, 170
value_type, 167
Class template default_error_checker
algebra_type, 165
default_error_checker, 165
error, 165-166
operations_type, 165
value_type, 165
Class template dense_output_runge_kutta
dense_output_runge_kutta, 176
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>
adjust_size, 180-181
algebra_type, 180

calc_state, 180-181
controlled_stepper_type, 180
dense_output_runge_kutta, 180
dense_output_stepper_type, 180
deriv_type, 180
do_step, 180-181
explicit_controlled_stepper_fsal_tag, 180
get_current_deriv, 180, 182
get_current_state, 180-181
get_old_deriv, 180, 182
get_old_state, 180, 182
initialize, 180-181
operations_type, 180
pre-conditions, 180-181
resize, 180-181
resizer_type, 180
state_type, 180
stepper_category, 180
stepper_type, 180
time_type, 180
toggle_current_state, 180, 182
value_type, 180
wrapped_deriv_type, 180
wrapped_state_type, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>
adjust_size, 177-178
algebra_type, 177
calc_state, 177-178
dense_output_runge_kutta, 177
dense_output_stepper_type, 177
deriv_type, 177
do_step, 177-178
equations, 178
get_current_state, 177, 179
get_old_state, 177, 179
initialize, 177-178
operations_type, 177
pre-conditions, 177, 179
resizer_type, 177
resize_impl, 177, 179
state_type, 177
stepper_category, 177
stepper_tag, 177
stepper_type, 177
time_type, 177
toggle_current_state, 177, 179
value_type, 177
wrapped_deriv_type, 177
wrapped_state_type, 177
Class template euler
adjust_size, 183, 185-186
algebra, 183, 186-187
algebra_type, 183
calc_state, 183, 185
deriv_type, 183
do_step, 183, 185-186
do_step_impl, 183-184
equations, 183, 185

euler, 183
explicit_stepper_base, 183
operations_type, 183
resizer_type, 183
state_type, 183
stepper_base_type, 183
time_type, 183
value_type, 183
Class template explicit_error_generic_rk
adjust_size, 188, 191
algebra, 188, 194
algebra_type, 188
coef_a_type, 188
coef_b_type, 188
coef_c_type, 188
deriv_type, 188
do_step, 188, 191-194
do_step_impl, 188, 190-191
equations, 191
example, 189-190
explicit_error_generic_rk, 188
explicit_error_stepper_base, 188
operations_type, 188
resizer_type, 188
resize_impl, 188, 195
rk_algorithm_type, 188
state_type, 188
stepper_base_type, 188
time_type, 188
value_type, 188
wrapped_deriv_type, 188
wrapped_state_type, 188
Class template explicit_error_stepper_base
adjust_size, 124, 130
algebra, 124, 130-131
algebra_stepper_base, 124
algebra_stepper_base_type, 124
algebra_type, 124
deriv_type, 124
do_step, 124, 127-130
do_step_impl, 125
do_step_v1, 124, 131
do_step_v5, 124, 131
equations, 127
example, 125-126
explicit_error_stepper_base, 124
order_type, 124
resizer_type, 124
resize_impl, 124, 131
state_type, 124
stepper, 124, 131
stepper_category, 124
stepper_type, 124
time_type, 124
value_type, 124
Class template explicit_error_stepper_fsal_base
adjust_size, 132, 139
algebra, 132, 140

algebra_stepper_base, 132
algebra_stepper_base_type, 132
algebra_type, 132
deriv_type, 132
do_step, 132-133, 136-139
do_step_impl, 133
do_step_v1, 132, 140
do_step_v5, 132, 140
equations, 136
example, 133, 135
explicit_error_stepper_fsal_base, 132
initialize, 132, 139
order_type, 132
reset, 132, 139
resizer_type, 132
resize_impl, 132, 140
state_type, 132
stepper, 132, 140
stepper_category, 132
stepper_type, 132
time_type, 132
value_type, 132
Class template explicit_generic_rk
adjust_size, 196, 198
algebra, 196, 200
algebra_type, 196
coef_a_type, 196
coef_b_type, 196
coef_c_type, 196
deriv_type, 196
do_step, 196, 198-199
do_step_impl, 196, 198
equations, 198
example, 197
explicit_generic_rk, 196
explicit_stepper_base, 196
operations_type, 196
resizer_type, 196
resize_impl, 196, 200
rk_algorithm_type, 196
state_type, 196
stepper_base_type, 196
time_type, 196
value_type, 196
wrapped_deriv_type, 196
wrapped_state_type, 196
Class template explicit_stepper_base
adjust_size, 142, 146
algebra, 142, 146
algebra_stepper_base, 142
algebra_stepper_base_type, 142
algebra_type, 142
base_type, 142
deriv_type, 142
do_step, 142, 145-146
do_step_impl, 142
do_step_v1, 142, 147
equations, 145

example, 142, 144
explicit_stepper_base, 142
operations_type, 142
order_type, 142
resizer_type, 142
resize_impl, 142, 147
snippet, 142
state_type, 142
stepper, 142, 146
stepper_category, 142
stepper_type, 142
time_type, 142
value_type, 142
Class template implicit_euler
adjust_size, 201
deriv_type, 201
do_step, 201
implicit_euler, 201
matrix_type, 201
pmatrix_type, 201
resizer_type, 201
resize_impl, 201
solve, 201
state_type, 201
stepper_category, 201
stepper_type, 201
time_type, 201
value_type, 201
wrapped_deriv_type, 201
wrapped_matrix_type, 201
wrapped_pmatrix_type, 201
wrapped_state_type, 201
Class template modified_midpoint
adjust_size, 203-204
algebra, 203, 206
algebra_type, 203
deriv_type, 203
do_step, 203-205
do_step_impl, 203-204
equations, 204
explicit_stepper_base, 203
modified_midpoint, 203
operations_type, 203
resizer_type, 203
resize_impl, 203, 206
set_steps, 203-204
state_type, 203
stepper_base_type, 203
stepper_type, 203
time_type, 203
value_type, 203
wrapped_deriv_type, 203
wrapped_state_type, 203
Class template modified_midpoint_dense_out
adjust_size, 207-208
algebra_type, 207
deriv_table_type, 207
deriv_type, 207

do_step, 207-208
modified_midpoint_dense_out, 207
operations_type, 207
resize, 207-208
resizer_type, 207
set_steps, 207-208
state_type, 207
stepper_type, 207
time_type, 207
value_type, 207
wrapped_deriv_type, 207
wrapped_state_type, 207
Class template observer_collection
collection_type, 108
observers, 108
observer_collection, 108
observer_type, 108
Class template rosenbrock4
adjust_size, 210-211
calc_state, 210-211
deriv_type, 210
do_step, 210-211
matrix_type, 210
order_type, 210
pmatrix_type, 210
pre-conditions, 210-211
prepare_dense_output, 210-211
resizer_type, 210
resize_impl, 210-211
resize_x_err, 210, 212
rosenbrock4, 210
rosenbrock_coefficients, 210
state_type, 210
stepper_category, 210
stepper_type, 210
time_type, 210
value_type, 210
wrapped_deriv_type, 210
wrapped_matrix_type, 210
wrapped_pmatrix_type, 210
wrapped_state_type, 210
Class template rosenbrock4_controller
adjust_size, 213-214
controller_type, 213
deriv_type, 213
error, 213
resizer_type, 213
resize_m_xerr, 213-214
resize_m_xnew, 213-214
rosenbrock4_controller, 213
state_type, 213
stepper, 213-214
stepper_category, 213
stepper_type, 213
time_type, 213
try_step, 213-214
value_type, 213
wrapped_deriv_type, 213

wrapped_state_type, 213
Class template rosenbrock4_dense_output
adjust_size, 215-216
calc_state, 215-216
controlled_stepper_type, 215
dense_output_stepper_type, 215
deriv_type, 215
do_step, 215-216
get_current_state, 215-216
get_old_state, 215-216
initialize, 215
pre-conditions, 215-216
resizer_type, 215
resize_impl, 215-216
rosenbrock4_dense_output, 215
state_type, 215
stepper_category, 215
stepper_type, 215
time_type, 215
toggle_current_state, 215-216
value_type, 215
wrapped_deriv_type, 215
wrapped_state_type, 215
Class template runge_kutta4
adjust_size, 218, 220
algebra, 218, 221
algebra_type, 218
deriv_type, 218
do_step, 218, 220-221
do_step_impl, 218-219
equations, 218, 220
explicit_generic_rk, 218
operations_type, 218
resizer_type, 218
runge_kutta4, 218
state_type, 218
time_type, 218
value_type, 218
Class template runge_kutta4_classic
adjust_size, 223, 225-226
algebra, 223, 226
algebra_type, 223
default_operations, 223
deriv_type, 223
do_step, 223, 225-226
do_step_impl, 223-224
equations, 223, 225
explicit_stepper_base, 223
initially_resizer, 223
operations_type, 223
range_algebra, 223
resizer_type, 223
resize_impl, 223, 227
runge_kutta4_classic, 223
state_type, 223
stepper_base_type, 223
time_type, 223
value_type, 223

Class template runge_kutta_cash_karp54
adjust_size, 228, 230
algebra, 228, 234
algebra_type, 228
deriv_type, 228
do_step, 228, 231-233
do_step_impl, 228, 230
equations, 229, 231
explicit_error_generic_rk, 228
operations_type, 228
resizer_typ, 228
runge_kutta_cash_karp54, 228
state_type, 228
time_type, 228
value_type, 228
Class template runge_kutta_cash_karp54_classic
adjust_size, 235, 237
algebra, 235, 241
algebra_type, 235
default_operations, 235
deriv_type, 235
do_step, 235, 238-241
do_step_impl, 235, 237
equations, 236, 238
explicit_error_stepper_base, 235
initially_resizer, 235
operations_type, 235
range_algebra, 235
resizer_type, 235
resize_impl, 235, 241
runge_kutta_cash_karp54_classic, 235
state_type, 235
stepper_base_type, 235
time_type, 235
value_type, 235
Class template runge_kutta_dopri5
adjust_size, 242, 245
algebra, 242, 249
algebra_type, 242
calc_state, 242, 245
deriv_type, 242
do_step, 242, 245-248
do_step_impl, 242, 244
equations, 245
explicit_error_stepper_fsal_base, 242
initialize, 242, 249
operations_type, 242
reset, 242, 249
resizer_type, 242
resize_dxdt_tmp_impl, 242, 250
resize_k_x_tmp_impl, 242, 250
state_type, 242
stepper_base_type, 242
time_type, 242
value_type, 242
Class template runge_kutta_fehlberg78
adjust_size, 251, 253
algebra, 251, 257

algebra_type, 251
deriv_type, 251
do_step, 251, 254-256
do_step_impl, 251, 253
equations, 254
explicit_error_generic_rk, 251
operations_type, 251
pre-conditions, 252
resizer_type, 251
runge_kutta_fehlberg78, 251
state_type, 251
time_type, 251
value_type, 251
Class template symplectic_euler
adjust_size, 263, 266
algebra, 263, 266
algebra_type, 263
do_step, 263-265
pre-conditions, 263
symplectic_euler, 263
symplectic_nystroem_stepper_base, 263
value_type, 263
Class template symplectic_nystroem_stepper_base
adjust_size, 148, 152
algebra, 148, 152
algebra_stepper_base, 148
algebra_stepper_base_type, 148
algebra_type, 148
coef_type, 148
coor_deriv_type, 148
coor_type, 148
deriv_type, 148
do_step, 148, 150-151
do_step_impl, 148, 152
equations, 149
momentum_deriv_type, 148
momentum_type, 148
operations_type, 148
order_type, 148
pre-conditions, 149
resizer_type, 148
resize_dpdt, 148, 152
resize_dqdt, 148, 152
state_type, 148
stepper_category, 148
symplectic_nystroem_stepper_base, 148
time_type, 148
value_type, 148
wrapped_coor_deriv_type, 148
wrapped_momentum_deriv_type, 148
Class template symplectic_rkn_sb3a_m4_mclachlan
adjust_size, 267, 270
algebra, 267, 270
algebra_type, 267
do_step, 267-270
pre-conditions, 268
symplectic_nystroem_stepper_base, 267
symplectic_rkn_sb3a_m4_mclachlan, 267

value_type, 267
Class template symplectic_rkn_sb3a_mclachlan
adjust_size, 272, 275
algebra, 272, 275
algebra_type, 272
do_step, 272-274
pre-conditions, 273
symplectic_nystroem_stepper_base, 272
symplectic_rkn_sb3a_mclachlan, 272
value_type, 272
coef_a_type
Class template explicit_error_generic_rk, 188
Class template explicit_generic_rk, 196
coef_b_type
Class template explicit_error_generic_rk, 188
Class template explicit_generic_rk, 196
coef_c_type
Class template explicit_error_generic_rk, 188
Class template explicit_generic_rk, 196
coef_type
Class template symplectic_nystroem_stepper_base, 148
collection_type
Class template observer_collection, 108
Complex state types
example, 22
pre-conditions, 22
state_type, 22
stepper_type, 22
Construction/Resizing
example, 72
Controlled steppers
example, 54
max, 54
pre-conditions, 54
controlled_runge_kutta
Class template controlled_runge_kutta, 166
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
controlled_stepper_tag
Struct base_tag<controlled_stepper_tag>, 261
Struct controlled_stepper_tag, 259
Struct explicit_controlled_stepper_fsal_tag, 259
Struct explicit_controlled_stepper_tag, 259
controlled_stepper_type
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template rosenbrock4_dense_output, 215
Integration with Adaptive Step Size, 11
controller_factory
Generation functions, 67
controller_type
Class template rosenbrock4_controller, 213
coor_deriv_type
Class template symplectic_nystroem_stepper_base, 148
coor_type
Class template symplectic_nystroem_stepper_base, 148
Custom Runge-Kutta steppers
algebra_type, 65
default_operations, 65

deriv_type, 65
example, 65
explicit_generic_rk, 65
initially_resizer, 65
operations_type, 65
pre-conditions, 65
range_algebra, 65
resizer_type, 65
state_type, 65
stepper_base_type, 65
stepper_type, 65
time_type, 65
value_type, 65
wrapped_deriv_type, 65
wrapped_state_type, 65
Custom steppers
deriv_type, 63
do_step, 63
equations, 63
example, 63
f, 63
order_type, 63
state_type, 63
stepper_category, 63
time_type, 63
value_type, 63

D

decrement
GSL Vector, 74
default_error_checker
Class template default_error_checker, 165
default_operations
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54_classic, 235
Custom Runge-Kutta steppers, 65
default_rosenbrock_coefficients
Struct template default_rosenbrock_coefficients, 209
Define the ODE
equations, 10
example, 10
f, 10
state_type, 10
Define the system function
book, 14
equations, 14
example, 14
f, 14
pre-conditions, 14
stepper_type, 14
value_type, 14
Dense output steppers
example, 55
dense_output_runge_kutta
Class template dense_output_runge_kutta, 176
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177

dense_output_stepper_tag
Struct base_tag<dense_output_stepper_tag>, 262
Struct dense_output_stepper_tag, 259
dense_output_stepper_type
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template rosenbrock4_dense_output, 215
deriv_table_type
Class template modified_midpoint_dense_out, 207
deriv_type
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116
Class template adams_moulton, 120
Class template bulirsch_stoer, 154
Class template bulirsch_stoer_dense_out, 159
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template euler, 183
Class template explicit_error_generic_rk, 188
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_generic_rk, 196
Class template explicit_stepper_base, 142
Class template implicit_euler, 201
Class template modified_midpoint, 203
Class template modified_midpoint_dense_out, 207
Class template rosenbrock4, 210
Class template rosenbrock4_controller, 213
Class template rosenbrock4_dense_output, 215
Class template runge_kutta4, 218
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54, 228
Class template runge_kutta_cash_karp54_classic, 235
Class template runge_kutta_dopri5, 242
Class template runge_kutta_fehlberg78, 251
Class template symplectic_nystroem_stepper_base, 148
Custom Runge-Kutta steppers, 65
Custom steppers, 63
Using boost::units, 28
do_step
Class template adams_bashforth, 110, 112-113
Class template adams_bashforth_moulton, 116, 118
Class template adams_moulton, 120-121
Class template bulirsch_stoer_dense_out, 159, 161
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177-178
Class template euler, 183, 185-186
Class template explicit_error_generic_rk, 188, 191-194
Class template explicit_error_stepper_base, 124, 127-130
Class template explicit_error_stepper_fsal_base, 132-133, 136-139
Class template explicit_generic_rk, 196, 198-199
Class template explicit_stepper_base, 142, 145-146
Class template implicit_euler, 201
Class template modified_midpoint, 203-205
Class template modified_midpoint_dense_out, 207-208
Class template rosenbrock4, 210-211

Class template rosenbrock4_dense_output, 215-216
Class template runge_kutta4, 218, 220-221
Class template runge_kutta4_classic, 223, 225-226
Class template runge_kutta_cash_karp54, 228, 231-233
Class template runge_kutta_cash_karp54_classic, 235, 238-241
Class template runge_kutta_dopri5, 242, 245-248
Class template runge_kutta_fehlberg78, 251, 254-256
Class template symplectic_euler, 263-265
Class template symplectic_nystroem_stepper_base, 148, 150-151
Class template symplectic_rkn_sb3a_m4_mclachlan, 267-270
Class template symplectic_rkn_sb3a_mclachlan, 272-274
Custom steppers, 63
Error Stepper, 89
Explicit steppers, 50
do_step_impl
Class template adams_bashforth, 110, 114
Class template euler, 183-184
Class template explicit_error_generic_rk, 188, 190-191
Class template explicit_error_stepper_base, 125
Class template explicit_error_stepper_fsal_base, 133
Class template explicit_generic_rk, 196, 198
Class template explicit_stepper_base, 142
Class template modified_midpoint, 203-204
Class template runge_kutta4, 218-219
Class template runge_kutta4_classic, 223-224
Class template runge_kutta_cash_karp54, 228, 230
Class template runge_kutta_cash_karp54_classic, 235, 237
Class template runge_kutta_dopri5, 242, 244
Class template runge_kutta_fehlberg78, 251, 253
Class template symplectic_nystroem_stepper_base, 148, 152
do_step_v1
Class template explicit_error_stepper_base, 124, 131
Class template explicit_error_stepper_fsal_base, 132, 140
Class template explicit_stepper_base, 142, 147
do_step_v5
Class template explicit_error_stepper_base, 124, 131
Class template explicit_error_stepper_fsal_base, 132, 140

E

end_iterator
GSL Vector, 74
Ensembles of oscillators
equations, 25
example, 25
integrate_const, 25
reset, 25
equations
Chaotic systems and Lyapunov exponents, 17
Class template adams_bashforth, 112-114
Class template adams_bashforth_moulton, 118-119
Class template bulirsch_stoer_dense_out, 162
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 178
Class template euler, 183, 185
Class template explicit_error_generic_rk, 191
Class template explicit_error_stepper_base, 127
Class template explicit_error_stepper_fsal_base, 136
Class template explicit_generic_rk, 198

Class template explicit_stepper_base, 145
Class template modified_midpoint, 204
Class template runge_kutta4, 218, 220
Class template runge_kutta4_classic, 223, 225
Class template runge_kutta_cash_karp54, 229, 231
Class template runge_kutta_cash_karp54_classic, 236, 238
Class template runge_kutta_dopri5, 245
Class template runge_kutta_fehlberg78, 254
Class template symplectic_nystroem_stepper_base, 149
Custom steppers, 63
Define the ODE, 10
Define the system function, 14
Ensembles of oscillators, 25
Examples Overview, 47
Explicit steppers, 50
Function template integrate, 101-102
Gravitation and energy conservation, 13
Implicit solvers, 53
Large oscillator chains, 40
Lattice systems, 23
Literature, 99
Overview, 3
Parameter studies, 42
Short Example, 8
Simple Symplectic System, 86
State Algebra Operations, 94
Steppers, 50
Stiff systems, 20
Symplectic solvers, 51
Symplectic System, 85
Using boost::units, 28
Using CUDA (or OpenMP, TBB, ...) via Thrust, 36
Using matrices as state types, 30
Using steppers, 56
error
Class template default_error_checker, 165-166
Class template rosenbrock4_controller, 213
Error Stepper
do_step, 89
error_checker_type
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
error_stepper_tag
Struct base_tag<error_stepper_tag>, 260
Struct error_stepper_tag, 258
Struct explicit_error_stepper_fsal_tag, 258
Struct explicit_error_stepper_tag, 258
euler
Class template euler, 183
example
Acknowledgments, 100
All examples, 47
Boost.Ublas, 78
Chaotic systems and Lyapunov exponents, 17
Class template explicit_error_generic_rk, 189-190
Class template explicit_error_stepper_base, 125-126
Class template explicit_error_stepper_fsal_base, 133, 135
Class template explicit_generic_rk, 197

Class template explicit_stepper_base, 142, 144
Complex state types, 22
Construction/Resizing, 72
Controlled steppers, 54
Custom Runge-Kutta steppers, 65
Custom steppers, 63
Define the ODE, 10
Define the system function, 14
Dense output steppers, 55
Ensembles of oscillators, 25
Example expressions, 98
Examples Overview, 47
Explicit steppers, 50
Gravitation and energy conservation, 13
GSL Vector, 74
Harmonic oscillator, 10
Integration with Adaptive Step Size, 11
Integration with Constant Step Size, 11
Large oscillator chains, 40
Lattice systems, 23
Multistep methods, 53
Overview, 3
Parameter studies, 42
Phase oscillator ensemble, 37
Point type, 79
Self expanding lattices, 33
Short Example, 8
State Algebra Operations, 94
State types, algebras and operations, 71
std::list, 73
Stepper Types, 10
Stiff systems, 20
Symplectic solvers, 51
Using arbitrary precision floating point types, 32
Using boost::range, 81
Using boost::ref, 81
Using boost::units, 28
Using CUDA (or OpenMP, TBB, ...) via Thrust, 36
Using matrices as state types, 30
Using OpenCL via VexCL, 45
Using steppers, 56
Using the container interface, 72
Example expressions
example, 98
Examples Overview
equations, 47
example, 47
graphics, 47
pre-conditions, 47
Explicit steppers
do_step, 50
equations, 50
example, 50
f, 50
explicit_controlled_stepper_fsal_tag
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Struct base_tag<explicit_controlled_stepper_fsal_tag>, 262
Struct explicit_controlled_stepper_fsal_tag, 259

explicit_controlled_stepper_tag
Struct base_tag<explicit_controlled_stepper_tag>, 261
Struct explicit_controlled_stepper_tag, 259
explicit_error_generic_rk
Class template explicit_error_generic_rk, 188
Class template runge_kutta_cash_karp54, 228
Class template runge_kutta_fehlberg78, 251
explicit_error_stepper_base
Class template explicit_error_generic_rk, 188
Class template explicit_error_stepper_base, 124
Class template runge_kutta_cash_karp54_classic, 235
explicit_error_stepper_fsal_base
Class template explicit_error_stepper_fsal_base, 132
Class template runge_kutta_dopri5, 242
explicit_error_stepper_fsal_tag
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Struct base_tag<explicit_error_stepper_fsal_tag>, 261
Struct explicit_error_stepper_fsal_tag, 258
explicit_error_stepper_tag
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Struct base_tag<explicit_error_stepper_tag>, 260
Struct explicit_error_stepper_tag, 258
explicit_generic_rk
Class template explicit_generic_rk, 196
Class template runge_kutta4, 218
Custom Runge-Kutta steppers, 65
explicit_stepper_base
Class template euler, 183
Class template explicit_generic_rk, 196
Class template explicit_stepper_base, 142
Class template modified_midpoint, 203
Class template runge_kutta4_classic, 223

F

f
Custom steppers, 63
Define the ODE, 10
Define the system function, 14
Explicit steppers, 50
Implicit solvers, 53
Implicit System, 87
Overview, 3
Short Example, 8
Stiff systems, 20
Symplectic solvers, 51
Symplectic System, 85
System, 85
Function template integrate
equations, 101-102
integrate, 101-102
Function template integrate_adaptive
integrate_adaptive, 103
Function template integrate_const
integrate_const, 104
Function template integrate_n_steps
integrate_n_steps, 105
Function template integrate_times

integrate_times, 107

G

Generation functions
controller_factory, 67
get_controller, 67
type, 67
Generation functions make_controlled(abs_error , rel_error , stepper)
remark, 11, 67
Generation functions make_dense_output(abs_error , rel_error , stepper)
remark, 11, 67
get_controller
Generation functions, 67
get_current_deriv
Class template bulirsch_stoer_dense_out, 159, 164
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180, 182
get_current_state
Class template bulirsch_stoer_dense_out, 159, 163
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177, 179
Class template rosenbrock4_dense_output, 215-216
get_old_deriv
Class template bulirsch_stoer_dense_out, 159, 164
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180, 182
get_old_state
Class template bulirsch_stoer_dense_out, 159, 163-164
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180, 182
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177, 179
Class template rosenbrock4_dense_output, 215-216
graphics
Examples Overview, 47
Using CUDA (or OpenMP, TBB, ...) via Thrust, 36
Gravitation and energy conservation
equations, 13
example, 13
GSL Vector
advance, 74
decrement, 74
end_iterator, 74
example, 74
gsl_vector_iterator, 74
increment, 74
is_resizeable, 74
iter, 74
pre-conditions, 74
range_begin, 74
range_end, 74
resize, 74
resize_impl, 74
same_size, 74
same_size_impl, 74
state_type, 74
state_wrapper, 74
state_wrapper_type, 74
type, 74
value_type, 74
gsl_vector_iterator

GSL Vector, 74

H

Harmonic oscillator
example, 10
Header < boost/numeric/odeint/integrate/integrate.hpp >
integrate, 101
Header < boost/numeric/odeint/integrate/integrate_adaptive.hpp >
integrate_adaptive, 102
Header < boost/numeric/odeint/integrate/integrate_const.hpp >
integrate_const, 104
Header < boost/numeric/odeint/integrate/integrate_n_steps.hpp >
integrate_n_steps, 105
Header < boost/numeric/odeint/integrate/integrate_times.hpp >
integrate_times, 106

I

Implicit solvers
equations, 53
f, 53
Implicit System
f, 87
implicit_euler
Class template implicit_euler, 201
increment
GSL Vector, 74
index
Indexes, 276
Large oscillator chains, 40
Self expanding lattices, 33
Indexes
index, 276
initialize
Class template adams_bashforth, 110, 113
Class template adams_bashforth_moulton, 116, 118-119
Class template bulirsch_stoer_dense_out, 159, 161
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 174
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177-178
Class template explicit_error_stepper_fsal_base, 132, 139
Class template rosenbrock4_dense_output, 215
Class template runge_kutta_dopri5, 242, 249
Using boost::range, 81
Using steppers, 56
initializing_stepper
Class template adams_bashforth, 110, 114
initializing_stepper_type
Class template adams_bashforth, 110
initially_resizer
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54_classic, 235
Custom Runge-Kutta steppers, 65
integrate
Function template integrate, 101-102
Header < boost/numeric/odeint/integrate/integrate.hpp >, 101
Integrate functions
pre-conditions, 69

integrate_adaptive
Function template integrate_adaptive, 103
Header < boost/numeric/odeint/integrate/integrate_adaptive.hpp >, 102
Parameter studies, 42
integrate_const
Ensembles of oscillators, 25
Function template integrate_const, 104
Header < boost/numeric/odeint/integrate/integrate_const.hpp >, 104
Integration with Constant Step Size, 11
Large oscillator chains, 40
Using OpenCL via VexCL, 45
integrate_n_steps
Chaotic systems and Lyapunov exponents, 17
Function template integrate_n_steps, 105
Header < boost/numeric/odeint/integrate/integrate_n_steps.hpp >, 105
integrate_times
Function template integrate_times, 107
Header < boost/numeric/odeint/integrate/integrate_times.hpp >, 106
Integration with Adaptive Step Size
controlled_stepper_type, 11
example, 11
Integration with Constant Step Size
example, 11
integrate_const, 11
is_resizeable
Boost.Ublas, 78
GSL Vector, 74
std::list, 73
Using the container interface, 72
iter
GSL Vector, 74

L

Large oscillator chains
equations, 40
example, 40
index, 40
integrate_const, 40
pre-conditions, 40
state_type, 40
sys, 40
Lattice systems
equations, 23
example, 23
remark, 23
stepper_type, 23
links
Usage, Compilation, Headers, 7
Literature
equations, 99
pre-conditions, 99

M

matrix_type
Class template implicit_euler, 201
Class template rosenbrock4, 210
Stiff systems, 20

max
Adaptive step size algorithms, 54
Controlled steppers, 54
modified_midpoint
Class template modified_midpoint, 203
modified_midpoint_dense_out
Class template modified_midpoint_dense_out, 207
momentum_deriv_type
Class template symplectic_nystroem_stepper_base, 148
momentum_type
Class template symplectic_nystroem_stepper_base, 148
Multistep methods
example, 53
pre-conditions, 53

N

null_observer
Struct null_observer, 107

O

observers
Class template observer_collection, 108
observer_collection
Class template observer_collection, 108
observer_type
Class template observer_collection, 108
ode
Binding member functions, 83
operations_type
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116
Class template adams_moulton, 120
Class template algebra_stepper_base, 122
Class template bulirsch_stoer, 154
Class template bulirsch_stoer_dense_out, 159
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Class template default_error_checker, 165
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template euler, 183
Class template explicit_error_generic_rk, 188
Class template explicit_generic_rk, 196
Class template explicit_stepper_base, 142
Class template modified_midpoint, 203
Class template modified_midpoint_dense_out, 207
Class template runge_kutta4, 218
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54, 228
Class template runge_kutta_cash_karp54_classic, 235
Class template runge_kutta_dopri5, 242
Class template runge_kutta_fehlberg78, 251
Class template symplectic_nystroem_stepper_base, 148
Custom Runge-Kutta steppers, 65
order_type
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116

Class template adams_moulton, 120
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_stepper_base, 142
Class template rosenbrock4, 210
Class template symplectic_nystroem_stepper_base, 148
Custom steppers, 63
Struct template default_rosenbrock_coefficients, 209
Overview
equations, 3
example, 3
f, 3

P

Parameter studies
equations, 42
example, 42
integrate_adaptive, 42
pre-conditions, 42
stepper_type, 42
system, 42
path
Usage, Compilation, Headers, 7
Phase oscillator ensemble
example, 37
pre-conditions, 37
state_type, 37
stepper_type, 37
value_type, 37
pmatrix_type
Class template implicit_euler, 201
Class template rosenbrock4, 210
Point type
example, 79
vector_space_reduce, 79
pre-conditions
Chaotic systems and Lyapunov exponents, 17
Class template adams_bashforth, 111, 113
Class template adams_bashforth_moulton, 116, 118
Class template bulirsch_stoer_dense_out, 159, 162-163
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177, 179
Class template rosenbrock4, 210-211
Class template rosenbrock4_dense_output, 215-216
Class template runge_kutta_fehlberg78, 252
Class template symplectic_euler, 263
Class template symplectic_nystroem_stepper_base, 149
Class template symplectic_rkn_sb3a_m4_mclachlan, 268
Class template symplectic_rkn_sb3a_mclachlan, 273
Complex state types, 22
Controlled steppers, 54
Custom Runge-Kutta steppers, 65
Define the system function, 14
Examples Overview, 47
GSL Vector, 74
Integrate functions, 69
Large oscillator chains, 40

Literature, 99
Multistep methods, 53
Parameter studies, 42
Phase oscillator ensemble, 37
Pre-Defined implementations, 97
Stepper Algorithms, 3, 59
Steppers, 50
Usage, Compilation, Headers, 7
Using arbitrary precision floating point types, 32
Using OpenCL via VexCL, 45
Using steppers, 58
Pre-Defined implementations
pre-conditions, 97
remark, 97
prepare_dense_output
Class template bulirsch_stoer_dense_out, 159, 163
Class template rosenbrock4, 210-211

R

range_algebra
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54_classic, 235
Custom Runge-Kutta steppers, 65
range_begin
GSL Vector, 74
range_end
GSL Vector, 74
remark
Generation functions make_controlled(abs_error , rel_error , stepper), 11, 67
Generation functions make_dense_output(abs_error , rel_error , stepper), 11, 67
Lattice systems, 23
Pre-Defined implementations, 97
Stepper Algorithms, 3, 59
reset
Class template adams_bashforth, 110, 114
Class template bulirsch_stoer, 154, 157
Class template bulirsch_stoer_dense_out, 159, 162
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 174
Class template explicit_error_stepper_fsal_base, 132, 139
Class template runge_kutta_dopri5, 242, 249
Ensembles of oscillators, 25
resize
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180-181
Class template modified_midpoint_dense_out, 207-208
GSL Vector, 74
std::list, 73
resizer_typ
Class template runge_kutta_cash_karp54, 228
resizer_type
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116
Class template adams_moulton, 120
Class template bulirsch_stoer, 154
Class template bulirsch_stoer_dense_out, 159
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180

Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template euler, 183
Class template explicit_error_generic_rk, 188
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_generic_rk, 196
Class template explicit_stepper_base, 142
Class template implicit_euler, 201
Class template modified_midpoint, 203
Class template modified_midpoint_dense_out, 207
Class template rosenbrock4, 210
Class template rosenbrock4_controller, 213
Class template rosenbrock4_dense_output, 215
Class template runge_kutta4, 218
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54_classic, 235
Class template runge_kutta_dopri5, 242
Class template runge_kutta_fehlberg78, 251
Class template symplectic_nystroem_stepper_base, 148
Custom Runge-Kutta steppers, 65
resize_dpdt
Class template symplectic_nystroem_stepper_base, 148, 152
resize_dqdt
Class template symplectic_nystroem_stepper_base, 148, 152
resize_dxdt_tmp_impl
Class template runge_kutta_dopri5, 242, 250
resize_impl
Class template adams_bashforth, 110, 114
Class template adams_moulton, 120, 122
Class template bulirsch_stoer, 154, 157
Class template bulirsch_stoer_dense_out, 159, 163
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177, 179
Class template explicit_error_generic_rk, 188, 195
Class template explicit_error_stepper_base, 124, 131
Class template explicit_error_stepper_fsal_base, 132, 140
Class template explicit_generic_rk, 196, 200
Class template explicit_stepper_base, 142, 147
Class template implicit_euler, 201
Class template modified_midpoint, 203, 206
Class template rosenbrock4, 210-211
Class template rosenbrock4_dense_output, 215-216
Class template runge_kutta4_classic, 223, 227
Class template runge_kutta_cash_karp54_classic, 235, 241
GSL Vector, 74
std::list, 73
resize_k_x_tmp_impl
Class template runge_kutta_dopri5, 242, 250
resize_m_dxdt
Class template bulirsch_stoer, 154, 157
resize_m_dxdt_impl
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 175
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
resize_m_dxdt_new_impl
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 175
resize_m_xerr
Class template rosenbrock4_controller, 213-214
resize_m_xerr_impl
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 175

Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
resize_m_xnew
Class template bulirsch_stoer, 154, 157
Class template rosenbrock4_controller, 213-214
resize_m_xnew_impl
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 175
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
resize_x_err
Class template rosenbrock4, 210, 212
rk_algorithm_type
Class template explicit_error_generic_rk, 188
Class template explicit_generic_rk, 196
rosenbrock4
Class template rosenbrock4, 210
rosenbrock4_controller
Class template rosenbrock4_controller, 213
rosenbrock4_dense_output
Class template rosenbrock4_dense_output, 215
rosenbrock_coefficients
Class template rosenbrock4, 210
runge_kutta4
Class template runge_kutta4, 218
runge_kutta4_classic
Class template runge_kutta4_classic, 223
runge_kutta_cash_karp54
Class template runge_kutta_cash_karp54, 228
runge_kutta_cash_karp54_classic
Class template runge_kutta_cash_karp54_classic, 235
runge_kutta_fehlberg78
Class template runge_kutta_fehlberg78, 251

S

same_size
GSL Vector, 74
std::list, 73
same_size_impl
GSL Vector, 74
std::list, 73
Self expanding lattices
example, 33
index, 33
state_type, 33
set_steps
Class template modified_midpoint, 203-204
Class template modified_midpoint_dense_out, 207-208
Short Example
equations, 8
example, 8
f, 8
state_type, 8
stepper, 8
Simple Symplectic System
equations, 86
snippet
Chaotic systems and Lyapunov exponents, 17
Class template explicit_stepper_base, 142
solve

Class template implicit_euler, 201
State Algebra Operations
equations, 94
example, 94
State types, algebras and operations
example, 71
state_type
Boost.Ublas, 78
Chaotic systems and Lyapunov exponents, 17
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116
Class template adams_moulton, 120
Class template bulirsch_stoer, 154
Class template bulirsch_stoer_dense_out, 159
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template euler, 183
Class template explicit_error_generic_rk, 188
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_generic_rk, 196
Class template explicit_stepper_base, 142
Class template implicit_euler, 201
Class template modified_midpoint, 203
Class template modified_midpoint_dense_out, 207
Class template rosenbrock4, 210
Class template rosenbrock4_controller, 213
Class template rosenbrock4_dense_output, 215
Class template runge_kutta4, 218
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54, 228
Class template runge_kutta_cash_karp54_classic, 235
Class template runge_kutta_dopri5, 242
Class template runge_kutta_fehlberg78, 251
Class template symplectic_nystroem_stepper_base, 148
Complex state types, 22
Custom Runge-Kutta steppers, 65
Custom steppers, 63
Define the ODE, 10
GSL Vector, 74
Large oscillator chains, 40
Phase oscillator ensemble, 37
Self expanding lattices, 33
Short Example, 8
std::list, 73
Using arbitrary precision floating point types, 32
Using boost::units, 28
Using matrices as state types, 30
Using OpenCL via VexCL, 45
state_wrapper
GSL Vector, 74
state_wrapper_type
GSL Vector, 74
std::list
example, 73
is_resizeable, 73

resize, 73
resize_impl, 73
same_size, 73
same_size_impl, 73
state_type, 73
type, 73
stepper
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 174
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
Class template explicit_error_stepper_base, 124, 131
Class template explicit_error_stepper_fsal_base, 132, 140
Class template explicit_stepper_base, 142, 146
Class template rosenbrock4_controller, 213-214
Short Example, 8
Stepper Algorithms
pre-conditions, 3, 59
remark, 3, 59
Stepper Types
example, 10
Steppers
equations, 50
pre-conditions, 50
sys, 50
stepper_base_type
Class template euler, 183
Class template explicit_error_generic_rk, 188
Class template explicit_generic_rk, 196
Class template modified_midpoint, 203
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54_classic, 235
Class template runge_kutta_dopri5, 242
Custom Runge-Kutta steppers, 65
stepper_category
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116
Class template adams_moulton, 120
Class template bulirsch_stoer_dense_out, 159
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_stepper_base, 142
Class template implicit_euler, 201
Class template rosenbrock4, 210
Class template rosenbrock4_controller, 213
Class template rosenbrock4_dense_output, 215
Class template symplectic_nystroem_stepper_base, 148
Custom steppers, 63
stepper_tag
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Struct base_tag<stepper_tag>, 260
Struct error_stepper_tag, 258
Struct stepper_tag, 258
stepper_type
Class template adams_moulton, 120
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171

Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_stepper_base, 142
Class template implicit_euler, 201
Class template modified_midpoint, 203
Class template modified_midpoint_dense_out, 207
Class template rosenbrock4, 210
Class template rosenbrock4_controller, 213
Class template rosenbrock4_dense_output, 215
Complex state types, 22
Custom Runge-Kutta steppers, 65
Define the system function, 14
Lattice systems, 23
Parameter studies, 42
Phase oscillator ensemble, 37
Using boost::units, 28
step_storage
Class template adams_bashforth, 110, 113
step_storage_type
Class template adams_bashforth, 110
Class template adams_moulton, 120
Stiff systems
equations, 20
example, 20
f, 20
matrix_type, 20
Struct base_tag<controlled_stepper_tag>
base_tag, 261
controlled_stepper_tag, 261
type, 261
Struct base_tag<dense_output_stepper_tag>
base_tag, 262
dense_output_stepper_tag, 262
type, 262
Struct base_tag<error_stepper_tag>
base_tag, 260
error_stepper_tag, 260
type, 260
Struct base_tag<explicit_controlled_stepper_fsal_tag>
base_tag, 262
explicit_controlled_stepper_fsal_tag, 262
type, 262
Struct base_tag<explicit_controlled_stepper_tag>
base_tag, 261
explicit_controlled_stepper_tag, 261
type, 261
Struct base_tag<explicit_error_stepper_fsal_tag>
base_tag, 261
explicit_error_stepper_fsal_tag, 261
type, 261
Struct base_tag<explicit_error_stepper_tag>
base_tag, 260
explicit_error_stepper_tag, 260
type, 260
Struct base_tag<stepper_tag>

base_tag, 260
stepper_tag, 260
type, 260
Struct controlled_stepper_tag
controlled_stepper_tag, 259
Struct dense_output_stepper_tag
dense_output_stepper_tag, 259
Struct error_stepper_tag
error_stepper_tag, 258
stepper_tag, 258
Struct explicit_controlled_stepper_fsal_tag
controlled_stepper_tag, 259
explicit_controlled_stepper_fsal_tag, 259
Struct explicit_controlled_stepper_tag
controlled_stepper_tag, 259
explicit_controlled_stepper_tag, 259
Struct explicit_error_stepper_fsal_tag
error_stepper_tag, 258
explicit_error_stepper_fsal_tag, 258
Struct explicit_error_stepper_tag
error_stepper_tag, 258
explicit_error_stepper_tag, 258
Struct null_observer
null_observer, 107
Struct stepper_tag
stepper_tag, 258
Struct template base_tag
base_tag, 260
Struct template default_rosenbrock_coefficients
default_rosenbrock_coefficients, 209
order_type, 209
value_type, 209
Symplectic solvers
equations, 51
example, 51
f, 51
Symplectic System
equations, 85
f, 85
symplectic_euler
Class template symplectic_euler, 263
symplectic_nystroem_stepper_base
Class template symplectic_euler, 263
Class template symplectic_nystroem_stepper_base, 148
Class template symplectic_rkn_sb3a_m4_mclachlan, 267
Class template symplectic_rkn_sb3a_mclachlan, 272
symplectic_rkn_sb3a_m4_mclachlan
Class template symplectic_rkn_sb3a_m4_mclachlan, 267
symplectic_rkn_sb3a_mclachlan
Class template symplectic_rkn_sb3a_mclachlan, 272
sys
Large oscillator chains, 40
Steppers, 50
system
Parameter studies, 42
System
f, 85

T

time_type
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116
Class template adams_moulton, 120
Class template bulirsch_stoer, 154
Class template bulirsch_stoer_dense_out, 159
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template euler, 183
Class template explicit_error_generic_rk, 188
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_generic_rk, 196
Class template explicit_stepper_base, 142
Class template implicit_euler, 201
Class template modified_midpoint, 203
Class template modified_midpoint_dense_out, 207
Class template rosenbrock4, 210
Class template rosenbrock4_controller, 213
Class template rosenbrock4_dense_output, 215
Class template runge_kutta4, 218
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54, 228
Class template runge_kutta_cash_karp54_classic, 235
Class template runge_kutta_dopri5, 242
Class template runge_kutta_fehlberg78, 251
Class template symplectic_nystroem_stepper_base, 148
Custom Runge-Kutta steppers, 65
Custom steppers, 63
Using boost::units, 28
toggle_current_state
Class template bulirsch_stoer_dense_out, 159, 164
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180, 182
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177, 179
Class template rosenbrock4_dense_output, 215-216
try_step
Class template bulirsch_stoer, 154, 156-157
Class template bulirsch_stoer_dense_out, 159, 161
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171-173
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167-169
Class template rosenbrock4_controller, 213-214
try_step_v1
Class template bulirsch_stoer, 154, 158
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171, 175
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167, 170
type
Boost.Ublas, 78
Generation functions, 67
GSL Vector, 74
std::list, 73
Struct base_tag<controlled_stepper_tag>, 261
Struct base_tag<dense_output_stepper_tag>, 262
Struct base_tag<error_stepper_tag>, 260
Struct base_tag<explicit_controlled_stepper_fsal_tag>, 262

Struct base_tag<explicit_controlled_stepper_tag>, 261
Struct base_tag<explicit_error_stepper_fsal_tag>, 261
Struct base_tag<explicit_error_stepper_tag>, 260
Struct base_tag<stepper_tag>, 260
Using the container interface, 72

U

Usage, Compilation, Headers
links, 7
path, 7
pre-conditions, 7
Using arbitrary precision floating point types
example, 32
pre-conditions, 32
state_type, 32
value_type, 32
Using boost::range
example, 81
initialize, 81
Using boost::ref
example, 81
Using boost::units
deriv_type, 28
equations, 28
example, 28
state_type, 28
stepper_type, 28
time_type, 28
Using CUDA (or OpenMP, TBB, ...) via Thrust
equations, 36
example, 36
graphics, 36
Using matrices as state types
equations, 30
example, 30
state_type, 30
Using OpenCL via VexCL
example, 45
integrate_const, 45
pre-conditions, 45
state_type, 45
Using steppers
equations, 56
example, 56
initialize, 56
pre-conditions, 58
Using the container interface
example, 72
is_resizeable, 72
type, 72

V

value_type
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116
Class template adams_moulton, 120
Class template bulirsch_stoer, 154

Class template bulirsch_stoer_dense_out, 159
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_fsal_tag>, 171
Class template controlled_runge_kutta<ErrorStepper, ErrorChecker, Resizer, explicit_error_stepper_tag>, 167
Class template default_error_checker, 165
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template euler, 183
Class template explicit_error_generic_rk, 188
Class template explicit_error_stepper_base, 124
Class template explicit_error_stepper_fsal_base, 132
Class template explicit_generic_rk, 196
Class template explicit_stepper_base, 142
Class template implicit_euler, 201
Class template modified_midpoint, 203
Class template modified_midpoint_dense_out, 207
Class template rosenbrock4, 210
Class template rosenbrock4_controller, 213
Class template rosenbrock4_dense_output, 215
Class template runge_kutta4, 218
Class template runge_kutta4_classic, 223
Class template runge_kutta_cash_karp54, 228
Class template runge_kutta_cash_karp54_classic, 235
Class template runge_kutta_dopri5, 242
Class template runge_kutta_fehlberg78, 251
Class template symplectic_euler, 263
Class template symplectic_nystroem_stepper_base, 148
Class template symplectic_rkn_sb3a_m4_mclachlan, 267
Class template symplectic_rkn_sb3a_mclachlan, 272
Custom Runge-Kutta steppers, 65
Custom steppers, 63
Define the system function, 14
GSL Vector, 74
Phase oscillator ensemble, 37
Struct template default_rosenbrock_coefficients, 209
Using arbitrary precision floating point types, 32
vector_space_reduce
Point type, 79

W

wrapped_coor_deriv_type
Class template symplectic_nystroem_stepper_base, 148
wrapped_deriv_type
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116
Class template adams_moulton, 120
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template explicit_error_generic_rk, 188
Class template explicit_generic_rk, 196
Class template implicit_euler, 201
Class template modified_midpoint, 203
Class template modified_midpoint_dense_out, 207
Class template rosenbrock4, 210
Class template rosenbrock4_controller, 213
Class template rosenbrock4_dense_output, 215
Custom Runge-Kutta steppers, 65
wrapped_matrix_type

Class template implicit_euler, 201
Class template rosenbrock4, 210
wrapped_momentum_deriv_type
Class template symplectic_nystroem_stepper_base, 148
wrapped_pmatrix_type
Class template implicit_euler, 201
Class template rosenbrock4, 210
wrapped_state_type
Class template adams_bashforth, 110
Class template adams_bashforth_moulton, 116
Class template adams_moulton, 120
Class template dense_output_runge_kutta<Stepper, explicit_controlled_stepper_fsal_tag>, 180
Class template dense_output_runge_kutta<Stepper, stepper_tag>, 177
Class template explicit_error_generic_rk, 188
Class template explicit_generic_rk, 196
Class template implicit_euler, 201
Class template modified_midpoint, 203
Class template modified_midpoint_dense_out, 207
Class template rosenbrock4, 210
Class template rosenbrock4_controller, 213
Class template rosenbrock4_dense_output, 215
Custom Runge-Kutta steppers, 65