odeint
Solving ordinary differential equations in C++

Karsten Ahnert[1,2] and Mario Mulansky[2]

[1] Ambrosys GmbH, Potsdam
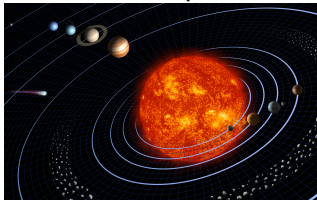[2] Institut für Physik und Astronomie, Universität Potsdam
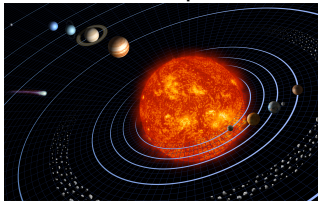
April 24, 2012

Outline

What is an ODE? – Examples

Newtons equations
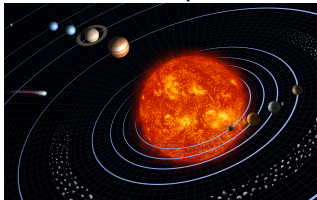
What is an ODE? – Examples

Newtons equations

Reaction and relaxation equations (i.e. blood alcohol content)

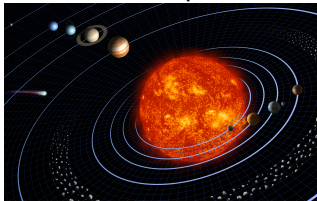# What is an ODE? – Examples

Newtons equations

Reaction and relaxation equations (i.e. blood alcohol content)

Granular systems

What is an ODE? – Examples
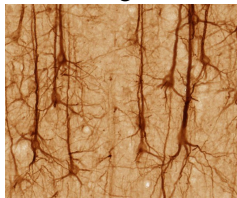
Newtons equations

Reaction and relaxation equations (i.e. blood alcohol content)

Granular systems

Interacting neurons

What is an ODE? – Examples
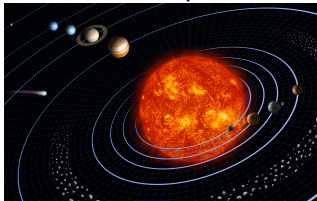
Newtons equations

Reaction and relaxation equations (i.e. blood alcohol content)

Granular systems

Interacting neurons
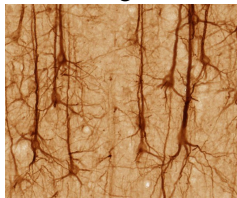
- Many examples in physics, biology, chemistry, social sciences
- Fundamental in mathematical modelling

What is an ODE?

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = f(x(t), t) \qquad \text{short form} \qquad \dot{x} = f(x, t)$$

- $x(t)$ – dependent variable
- t – indenpendent variable (time)
- $f(x, t)$ – defines the ODE

Initial Value Problem (IVP):

$$\dot{x} = f(x, t), \qquad x(t = 0) = x_0$$

Numerical integration of ODEs

Find a numerical solution of an ODE an its initial value problem

$$\dot{x} = f(x, t) , \qquad x(t = 0) = x_0$$

Example: Explicit Euler

$$x(t + \Delta t) = x(t) + \Delta t \, f(x(t), t) + \mathcal{O}(\Delta t^2)$$

General scheme of order s

$$x(t) \mapsto x(t + \Delta t) \qquad , \text{ or}$$

$$x(t + \Delta t) = \mathcal{F}_t x(t) + \mathcal{O}(\Delta t^{s+1})$$

odeint

Solving ordinary differential equations in C++

Open source
- Boost license – do whatever you want do to with it

odeint

Solving ordinary differential equations in C++

Open source

- Boost license – do whatever you want do to with it

Download

- **www.odeint.com**

odeint

Solving ordinary differential equations in C++

Open source

- Boost license – do whatever you want do to with it

Download

- **www.odeint.com**

Modern C++

- Generic programming, functional programming
- Fast, easy-to-use and extendable.
- Container independent
- Portable

Who uses odeint

NetEvo

OMPL – Open Motion
Planning Library

Motivation: The interface problem in C/C++

- Many frameworks exist to do numerical computations.
- Data has to be stored in containers or collections.
- GSL: `gsl_vector`, `gsl_matrix`
- NR: pointers with Fortran-style indexing
- Blitz++, MTL4, boost::ublas
- QT: `QVector`, wxWidgets: `wxArray`, MFC: `CArray`

But: All books on C++ recommend the use of the STL containers `std::vector`, `std::list`, ...

Motivation: The interface problem in C/C++

- Many frameworks exist to do numerical computations.
- Data has to be stored in containers or collections.
- GSL: `gsl_vector`, `gsl_matrix`
- NR: pointers with Fortran-style indexing
- Blitz++, MTL4, boost::ublas
- QT: `QVector`, wxWidgets: `wxArray`, MFC: `CArray`

But: All books on C++ recommend the use of the STL containers `std::vector`, `std::list`, ...

Theoretical solution of the interface mess

GoF Design Pattern: Adaptor, also known as Wrapper

Motivation: The interface problem in C/C++

- Many frameworks exist to do numerical computations.
- Data has to be stored in containers or collections.
- GSL: `gsl_vector`, `gsl_matrix`
- NR: pointers with Fortran-style indexing
- Blitz++, MTL4, boost::ublas
- QT: `QVector`, wxWidgets: `wxArray`, MFC: `CArray`

But: All books on C++ recommend the use of the STL
containers `std::vector`, `std::list`, ...

Theoretical solution of the interface mess

GoF Design Pattern: Adaptor, also known as Wrapper

Alternative

Generic, container independent algorithms

Portability of your algorithm

How to run your algorithm?

- Single machine, single CPU
- Single machine, multiple CPU's (OpenMP, threads, ...)
- Multiple machines (MPI)
- GPU (Cuda, Thrust, OpenCL)

Portability of your algorithm

How to run your algorithm?

- Single machine, single CPU
- Single machine, multiple CPU's (OpenMP, threads, ...)
- Multiple machines (MPI)
- GPU (Cuda, Thrust, OpenCL)

Which data types are used by your algorithm?

- Build-in data types – `double`, `complex<double>`
- Arbitrary precision types – GMP, MPFR
- Vectorial data types `float2d`, `float3d`

Portability of your algorithm

How to run your algorithm?

- Single machine, single CPU
- Single machine, multiple CPU's (OpenMP, threads, ...)
- Multiple machines (MPI)
- GPU (Cuda, Thrust, OpenCL)

Which data types are used by your algorithm?

- Build-in data types – `double`, `complex<double>`
- Arbitrary precision types – GMP, MPFR
- Vectorial data types `float2d`, `float3d`

Theoretical solution

GoF Design Pattern: Strategy, also known as Policy

Alternative

Generic algorithms

Lets step into odeint
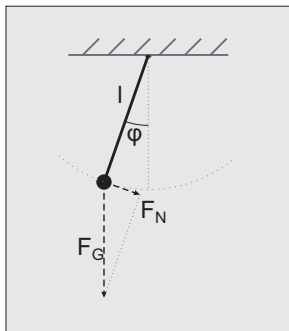
Example – Pendulum

Newtons law: $ma = F$

Acceleration: $a = l\ddot{\varphi}$

Force: $F = F_N = -mg\sin\varphi$

$\Longrightarrow$ **ODE for** φ

$$\ddot{\varphi} = -g/l\sin\varphi = -\omega_0^2\sin\varphi$$

Example – Pendulum
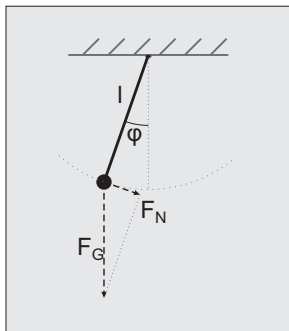
$$\ddot{\varphi} = -\omega_0^2 \sin \varphi$$

Small angle: $\sin \varphi \approx \varphi$

Harmonic oscillator $\ddot{\varphi} = -\omega_0^2 \varphi$

Analytic solution:
$\varphi = A \cos \omega_0 t + B \sin \omega_0 t$

Determine A and B from initial condition

Example – Pendulum
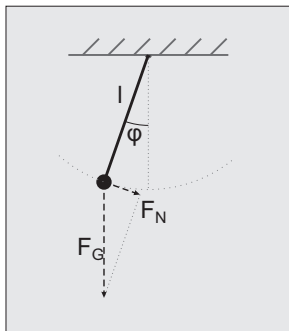
Full equation: $\ddot{\varphi} = -\omega_0^2 \sin\varphi$

Pendulum with friction and external driving:

$$\ddot{\varphi} = -\omega_0^2 \sin\varphi - \mu\dot{\varphi} + \varepsilon \sin\omega_E t$$

No analytic solution is known

$\Longrightarrow$ **Solve this equation numerically.**

Example – Pendulum
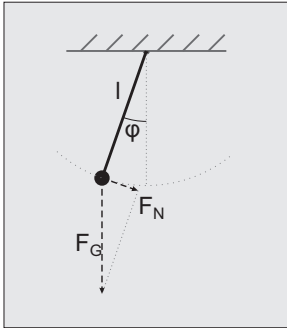
$$\ddot{\varphi} = -\omega_0^2 \sin\varphi - \mu\dot{\varphi} + \varepsilon \sin\omega_E t$$

Create a first order ODE

$$x_1 = \varphi \,, \quad x_2 = \dot{\varphi}$$

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = -\omega_0 \sin x_1 - \mu x_2 + \varepsilon \sin\omega_E t$$

x_1 and x_2 are the state space variables

Let's solve the pendulum example numerically

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

namespace odeint = boost::numeric::odeint;
```

$$\dot{x}_1 = x_2 \,, \quad \dot{x}_2 = -\omega_0 \sin x_1 - \mu x_2 + \varepsilon \sin \omega_E t$$

```cpp
typedef std::array<double,2> state_type;
```

Let's solve the pendulum example numerically

$$\dot{x}_1 = x_2, \; \dot{x}_2 = -\omega_0^2 \sin x_1 - \mu x_2 + \varepsilon \sin \omega_E t$$

```
struct pendulum
{
  double m_mu, m_omega, m_eps;

  pendulum(double mu,double omega,double eps)
  : m_mu(mu),m_omega(omega),m_eps(eps) { }

  void operator()(const state_type &x,
      state_type &dxdt,double t) const
  {
    dxdt[0] = x[1];
    dxdt[1] = -sin(x[0]) - m_mu * x[1] +
        m_eps * sin(m_omega*t);
  }
};
```

Let's solve the pendulum example numerically

$\varphi(0) = 1 , \quad \dot{\varphi}(0) = 0$

```cpp
odeint::rk4< state_type > rk4;
pendulum p( 0.1 , 1.05 , 1.5 );

state_type x = {{ 1.0 , 0.0 }};
double t = 0.0;

const double dt = 0.01;
rk4.do_step( p , x , t , dt );
t += dt;
```

$x(0) \mapsto x(\Delta t)$

Let's solve the pendulum example numerically

```
std::cout<<t<<" "<< x[0]<<" "<<x[1]<<"\n";
for( size_t i=0 ; i<10 ; ++i )
{
  rk4.do_step( p , x , t , dt );
  t += dt;
  std::cout<<t<<" "<< x[0]<<" "<<x[1]<<"\n";
}
```

$x(0) \mapsto x(\Delta t) \mapsto x(2\Delta t) \mapsto x(3\Delta) \mapsto \dots$
Grafik einfuegen

Simulation

Oscillator: $\mu = 0$, $\omega_E = 0$, $\varepsilon = 0$

Damped oscillator: $\mu = 0.1$, $\omega_E = 0$, $\varepsilon = 0$

Damped, driven oscillator: $\mu = 0.1$, $\omega_E = 1.05$, $\varepsilon = 1.5$

Different Steppers

```
runge_kutta_fehlberg78< state_type > s;
```

```
runge_kutta_dopri5< state_type > s;
```

Symplectic steppers (for Hamiltonian systems)

```
symplectic_rkn_sb3a_mclachlan< state_type > s;
```

Implicit steppers (for stiff systems)

```
rosenbrock4< double > s;
```

These steppers perform one step with constant step size!

Controlled steppers – Step size control

insert graphic

Controlled steppers

```
auto s = make_controlled(1.0e-6,1.0e6,
  runge_kutta_fehlberg78<state_type>() );
controlled_step_result r =
  s.try_step(ode,x,t,dt);
```

Tries to perform the step and updates *x*, *t*, and *dt*!

It works because Runge-Kutta-Fehlberg has error estimation:

```
runge_kutta_fehlberg78<state_type> s;
s.do_step(ode,x,t,dt,xerr);
```

Controlled steppers

```
auto s = make_controlled(1.0e-6,1.0e6,
  runge_kutta_fehlberg78<state_type>() );
while( t < t_end )
{
  controlled_step_result res
    = s.try_step(ode,x,t,dt);
  while( res != success )
  {
    res = s.try_step(ode,x,t,dt);
  }
}
```

Non-trivial time-stepping logic

Use integrate functions!

```
integrate_adaptive(s,ode,x,t_start,t_end,dt);
integrate_adaptive(s,ode,x,t_start,t_end,dt,
    observer);
```

Observer: Callable object `obs(x,t)`

Example (using Boost.Phoenix):

```
integrate_adaptive(s,ode,x,t_start,t_end,dt,
  cout<< arg1[0] << " " << arg1[1] << "\n" );
```

More integrate versions:
`integrate_const`, `integrate_times`,...

```
integrate_const(s,ode,x,t,dt,obs);
```

Grafik with problem and solution

Dense output

```
auto s = make_dense_output( 1.0e-6 , 1.0e-6 ,
    runge_kutta_dopri5< state_type >() );
integrate_const( s , p , x , t , dt );
```

Interpolation between two steps with same precision as the
original stepper!
Grafik!

More steppers

Stepper Concepts: Stepper, ErrorStepper, ControlledStepper, DenseOutputStepper

Stepper types:

- Implicit – `implicit_euler`, `rosenbrock4`
- Symplectic – `symplectic_rkn_sb3a_mclachlan`
- Predictor-Corrector – `adams_bashforth_moulton`
- Extrapolation – `bulirsch_stoer`
- Multistep methods – `adams_bashforth_moulton`

Some of them have step-size control and dense-output!

Small summary

- Very easy example – harmonic oscillator
- Basic features of odeint
- Different stepper – Controlled steppers, Dense output steppers
- Integrate functions

Small summary

- Very easy example – harmonic oscillator
- Basic features of odeint
- Different stepper – Controlled steppers, Dense output steppers
- Integrate functions

Now, lets look at the advanced features!

Extended systems

Lattice systems

Extended systems

Lattice systems

Discretiztations of PDEs

Extended systems

Lattice systems

Discretiztations of PDEs

Granular systems

Extended systems

Lattice systems

Discretiztations of PDEs

Granular systems

ODEs on graphs
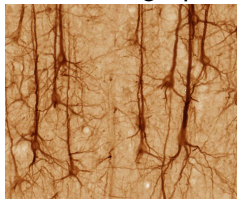
High-Performance-Computing

Phase oscillator lattices
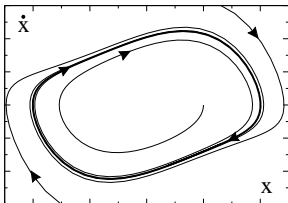
Any oscillator can be described by one variable, its phase.
Trivial dynamics: $\dot{\varphi} = \omega\varphi$
Coupled phase oscillators
Neurosciences
Heart dynamics
Synchronization
Any weakly perturbed oscillator system
$\dot{\varphi}_k = \omega_k\varphi_k + q(\varphi_{k+1}, \varphi_k) + q(\varphi_k, \varphi_{k-1})$

Phase compacton lattice

$$\dot{\varphi}_k = \cos\varphi_{k+1} - \cos\varphi_{k-1}$$

State space contains *N* variables

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

Animation
Space-time plot for visualization of compactons and chaos

Ensemble of phase oscillators

$$\dot{\varphi}_k = \omega_k + \sum_l \sin(\varphi_l - \varphi_k)$$

Synchronization – all oscillator oscillates with the same frequency

Synchronized state $\varphi_k = \omega_S t + \varphi_{0,k}$

Ensemble of phase oscillators

```cpp
typedef std::vector<double> state_type;

struct ensemble
{
    state_type m_omega,m_eps;

    ensemble(size_t n,double eps)
    : m_omega(n,0.0),m_eps(eps)
    {
        create_frequencies();
    }

    void create_frequencies() { ... }

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

Solving ODEs with CUDA using Thrust

Thrust is a parallel algorithms library which resembles the C++ Standard Template Library (STL). Thrust's high-level interface greatly enhances developer productivity while enabling performance portability between GPUs and multicore CPUs. Interoperability with established technologies (such as CUDA, TBB and OpenMP) facilitates integration with existing software. Develop high-performance applications rapidly with Thrust!

Solving ODEs with CUDA using thrust

Applications and use cases for GPUs:

- Large systems, discretizations of PDEs, lattice systems, granular systems, etc.
- Parameter studies, solve many ODEs in parallel with different parameters
- Initial value studies, solve the same ODE with many different initial conditions in parallel

Lorenz system – Parameter study

$$\dot{x} = \sigma(y - x) \qquad \dot{y} = Rx - y - xz \qquad \dot{z} = -bz + xy$$

Standard parameters $\sigma = 10$, $R = 28$, $b = 8/3$

Deterministic chaos – Butterfly effect
Picture of Lorenz system

Lorenz system – Parameter study

$$\dot{x} = \sigma(y - x) \qquad \dot{y} = Rx - y - xz \qquad \dot{z} = -bz + xy$$

Does one observ chaos over the whole parameter range?

Lyapunov exponents:

- Measure of chaos
- Perturbations of the original system

Vary R from 0 to 50 and calculate the Lyapunov exponents!

Use CUDA and Thrust!

odeint's algebras and operations

Euler method
$$x_i(t + \Delta t) = x_i(t) + \Delta t * f_i(x)$$
Algebras perform the iteration over *i* and operation the
elementary addition.
Algebras and operations enter the stepper as template
parameters

```
typedef runge_kutta4<state_type,value_type,
   deriv_type,time_type,
    algebra,operations,resize_policy> stepper;
```

- `default_operations`
- `range_algebra` – Boost.Ranges
- `vector_space_algebra` – Passes the state directly to
  the operations
- `fusion_algebra` – For compile time sequences, like
  std :: *tuple* < *double*, *double* >
- `thrust_algebra` and `thrust_algebra` – for thrust

Thrust example for Lorenz system,
Implementation of the system function

More advanced features, die themen können auch auf mehreren folien zusammengefasst werden

Boost::ref

boost::range

complex state types, vielleicht auch nicht

arbitrary precision types

matrices as state types

graph as state types

self expanding lattices

Old Stuff

First example – Lorenz system

```cpp
#include <iostream>
#include <boost/array.hpp>
#include <boost/numeric/odeint.hpp>

using namespace std;
using namespace boost::numeric::odeint;

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

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

void lorenz( const state_type &x , 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];
}
```

Different steppers:

```
runge_kutta4< state_type > stepper;
```

Different steppers:

```
runge_kutta4< state_type > stepper;
```

```
controlled_runge_kutta< runge_kutta_cash_karp54
    < state_type > > stepper;
```

Different steppers:

```
runge_kutta4< state_type > stepper;
```

```
controlled_runge_kutta< runge_kutta_cash_karp54
    < state_type > > stepper;
```

```
dense_output_runge_kutta<
    controlled_runge_kutta<
     runge_kutta_dopri5< state_type > > >
        stepper;
```

Lorenz system continued

Different steppers:

```
runge_kutta4< state_type > stepper;
```

```
controlled_runge_kutta< runge_kutta_cash_karp54
    < state_type > > stepper;
```

```
dense_output_runge_kutta<
    controlled_runge_kutta<
     runge_kutta_dopri5< state_type > > >
        stepper;
```

```
runge_kutta_dopri5< state_type > stepper;
make_dense_output( 1.0e-6 , 1.0e-6 , stepper );
        // incomplete
```

Different steppers:

```
runge_kutta4< state_type > stepper;
```

```
controlled_runge_kutta< runge_kutta_cash_karp54
    < state_type > > stepper;
```

```
dense_output_runge_kutta<
    controlled_runge_kutta<
     runge_kutta_dopri5< state_type > > >
        stepper;
```

```
runge_kutta_dopri5< state_type > stepper;
make_dense_output( 1.0e-6 , 1.0e-6 , stepper );
        // incomplete
```

All together:

```
int main( int argc , char **argv )
{
    state type x = {{ 10.0 , 1.0 , 1.0 }};
```

$$\dot{q}_k = p_k$$
$$\dot{p}_k = -q_k^2 + \Delta q_k + \beta\{(q_{k+1} - q_k)^3 - (q_k - q_{k-1})^3\}$$

$$\Delta q_k = q_{k+1} - 2q_k + q_{k-1}$$

Second example – Fermi-Pasta-Ulam lattice

$$\dot{q}_k = p_k$$
$$\dot{p}_k = -q_k^2 + \Delta q_k + \beta\{(q_{k+1} - q_k)^3 - (q_k - q_{k-1})^3\}$$
$$\Delta q_k = q_{k+1} - 2q_k + q_{k-1}$$

State type consists of coordinates *q* and momentas *p*

```
typedef std::vector<double> vector_type;
vector_type q( 256 ) , p( 256 );
// initialize q,p
std::pair< state_type , state_type > state =
    std::make_pair( q , p );
```

Second example – Fermi-Pasta-Ulam lattice

$$\dot{q}_k = p_k$$
$$\dot{p}_k = -q_k^2 + \Delta q_k + \beta\{(q_{k+1} - q_k)^3 - (q_k - q_{k-1})^3\}$$
$$\Delta q_k = q_{k+1} - 2q_k + q_{k-1}$$

State type consists of coordinates q and momentas p

```
typedef std::vector<double> vector_type;
vector_type q( 256 ) , p( 256 );
// initialize q,p
std::pair< state_type , state_type > state =
    std::make_pair( q , p );
```

Hamiltonian system $\implies$ Symplectic solvers needed

```
symplectic_rkn_sb3a_mclachlan< vector_type >
    stepper;
```

Fermi-Pasta-Ulam lattice continued

Trivial first component $\dot{q}_k = p_k$

```
struct fpu {
    double m_beta;
    fpu(double beta) : m_beta(beta) { }

    void operator()(const vector_type &q,
       vector_type &dpdt) const {
        // ...
    }
};
```

Trivial first component $\dot{q}_k = p_k$

```cpp
struct fpu {
    double m_beta;
    fpu(double beta) : m_beta(beta) { }

    void operator()(const vector_type &q,
       vector_type &dpdt) const {
        // ...
    }
};
```

All together

```cpp
struct statistics_observer {
    void operator()( const state_type &x ,
       double t ) const {
    // write the statistics
    }
};
```

Stepper Classes
- o—o + Stepper
- o—o + ErrorStepper
- o—o + ControlledStepper
- o—o + DenseOutputStepper

Integrate Functions
- + integrate()
- + integrate_adaptive()
- + integrate_const()
- + integrate_n_steps
- + integrate_times

Utils
- + state_wrapper
- + resize()
- + ...

Operations
- + default_operations
- + mkl_operations
- + thrust_operations

Algebra
- + range_algebra
- + fusion_algebra
- + thrust_algebra
- + vector_space_algebra
- + ...

User provides

$$y_i = f_i(x(t), t)$$

odeint provides

$$x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i$$

(In general vector operations like $z_i = a_1 x_{1,i} + a_2 x_{2,i} + \dots$)

Instantiation

```
euler<state_type, value_type, deriv_type,
    time_type, algebra, operations > stepper;
```

**All elements for container independence and portability
are already included in this line!**

$$y_i = f_i(x(t))$$
$$x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i$$

```
euler<state_type, value_type, deriv_type,
    time_type, algebra, operations > stepper;
```

Data types

- state_type – the type of *x*
- value_type – the basic numeric type, e.g. double
- deriv_type – the type of *y*
- time_type – the type of *t*, Δt

$$y_i = f_i(x(t))$$
$$x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i$$

```
euler<state_type, value_type, deriv_type,
    time_type, algebra, operations > stepper;
```

Algebra policies, perform the iteration

Algebra must be a class with public methods

- `for_each1(x,op)` – Performs $op(x_i)$ for all i
- `for_each2(x1,x2,op)` – Performs $op(x1_i, x2_i)$ for all i
- ...

$$y_i = f_i(x(t))$$
$$x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i$$

```
euler<state_type, value_type, deriv_type,
    time_type, algebra, operations > stepper;
```

Operations do the basic computation

Operations must be a class with the public classes (functors)

- scale_sum1 – Calculates $x = a1 \cdot y1$
- scale_sum2 – Calculates $x = a1 \cdot y1 + a2 \cdot y2$
- ...

$$y_i = f_i(x(t))$$
$$x_i(t + \Delta t) = x_i(t) + \Delta t \cdot y_i$$

```
euler<state_type, value_type, deriv_type,
    time_type, algebra, operations > stepper;
```

All together

```
m_algebra.for_each3(xnew ,xold, y ,
   operations_type::scale_sum2<value_type,time_type>(1.0,dt));
```

Concepts

"... In generic programming, a concept is a description of supported operations on a type..."

Stepper concepts

Concepts

"... In generic programming, a concept is a description of supported operations on a type..."

odeint provides

- Stepper concept
  ```
  stepper.do_step(sys,x,t,dt);
  ```
- ErrorStepper concept
  ```
  stepper.do_step(sys,x,t,dt,xerr);
  ```
- ControlledStepper concept
  ```
  stepper.try_step(sys,x,t,dt);
  ```
- DenseOutputStepper concept
  ```
  stepper.do_step(sys);
  stepper.calc_state(t,x);
  ```

Supported methods

Method	Class name	Concept
Euler	`euler`	SD
Runge-Kutta 4	`runge_kutta4`	S
Runge-Kutta Cash-Karp	`runge_kutta_cash_karp54`	SE
Runge-Kutta Fehlberg	`runge_kutta_runge_fehlberg78`	SE
Runge-Kutta Dormand-Prince	`runge_kutta_dopri5`	SED
Runge-Kutta controller	`controlled_runge_kutta`	C
Runge-Kutta dense output	`dense_output_runge_kutta`	D
Symplectic Euler	`symplectic_euler`	S
Symplectic RKN	`symplectic_rkn_sb3a_mclachlan`	S
Rosenbrock 4	`rosenbrock4`	ECD
Implicit Euler	`implicit_euler`	S
Adams-Bashforth-Moulton	`adams_bashforth_moulton`	S
Bulirsch-Stoer	`bulirsch_stoer`	CD

S – fulfills stepper concept

E – fulfills error stepper concept

C – fulfills controlled stepper concept

D – fulfills dense output stepper concept

- `integrate_const`
- `integrate_adaptive`
- `integrate_times`
- `integrate_n_steps`

Perform many steps, use all features of the underlying method

Integrate functions

- integrate_const
- integrate_adaptive
- integrate_times
- integrate_n_steps

Perform many steps, use all features of the underlying method

An additional observer can be called

```
integrate_const(stepper, sys, x, t_start,
t_end, dt, obs);
```

More internals

- Header-only, no linking $\rightarrow$ powerful compiler optimization
- Memory allocation is managed internally
- No virtual inheritance, no virtual functions are called
- Different container types are supported, for example
    - STL containers (`vector`, `list`, `map`, `tr1::array`)
    - MTL4 matrix types, blitz++ arrays, Boost.Ublas matrix types
    - `thrust::device_vector`
    - Fancy types, like Boost.Units
    - ANY type you like
- Explicit Runge-Kutta-steppers are implemented with a new template-metaprogramming method
- Different operations and algebras are supported
    - MKL
    - Thrust
    - gsl

Graphical processing units (GPUs) are able to perform up to 10^6 operations at once in parallel

Frameworks

- CUDA from NVIDIA
- OpenCL
- Thrust a STL-like library for CUDA and OpenMP

Applications:

- Parameter studies
- Large systems, like ensembles or one- or two dimensional lattices
- Discretizations of PDEs

odeint supports CUDA, through Thrust

```cpp
typedef thrust::device_vector<double>
   state_type;
typedef runge_kutta4<state_type ,value_type ,
   state_type ,value_type ,thrust_algebra ,
   thrust_operations > stepper_type;

struct lorenz_system {

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

    void operator()( const state_type &x ,
       state_type &dxdt , double t ){
        // ..
    }

    size_t m_N;
    const state_type &m_beta;
};
```

- odeint provides a fast, flexible and easy-to-use C++ library for numerical integration of ODEs.
- Its container independence is a large advantage over existing libraries.
- Portable
- Generic programming is the main programming technique.

Outlook

- Submission to the boost libraries
- Dynamical system classes for easy implementation of interacting dynamical systems
- More methods: implicit methods and multistep methods.
- Implementation of the Taylor series method

```
taylor_fixed_order< 25 , 3 > taylor_type
   stepper;

stepper.do_step(
    fusion::make_vector
    (
        sigma * ( arg2 - arg1 ) ,
        R * arg1 - arg2 - arg1 * arg3 ,
        arg1 * arg2 - b * arg3
    ) , x , t , dt );
```

Download and documentation
`odeint.com`

An article about the used techniques exists at
`http://www.codeproject.com/KB/recipes/odeint-v2.aspx`

Development
`https://github.com/headmyshoulder/odeint-v2`

Contributions and feedback
are highly welcome