odeint

Solving ordinary differential equations in C++

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

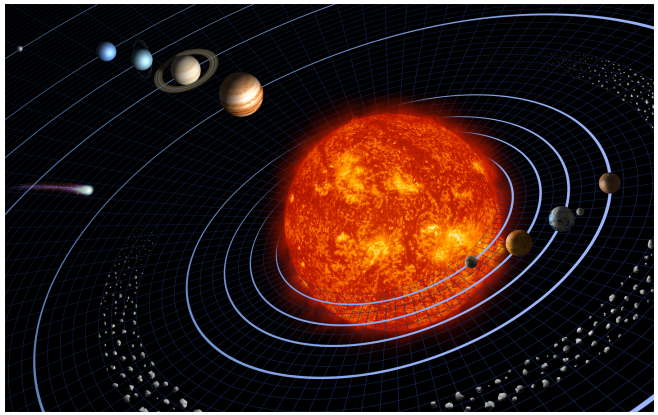
[1] Ambrosys GmbH, Potsdam
[2] Institut für Physik und Astronomie, Universität Potsdam

May 8, 2012

Outline

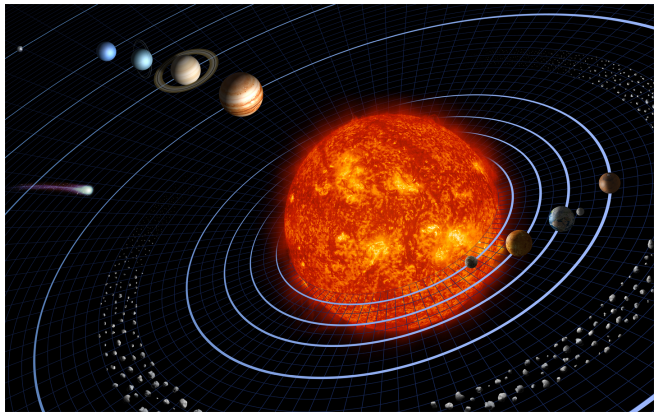
What is an ODE? – Examples

Newtons equations



What is an ODE? – Examples

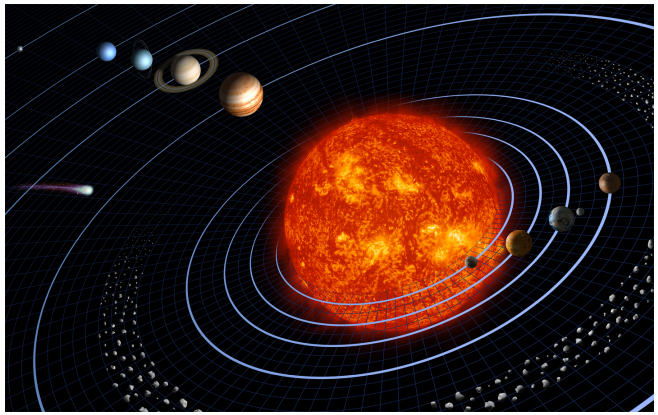
Newtons equations



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

What is an ODE? – Examples

Newtons equations

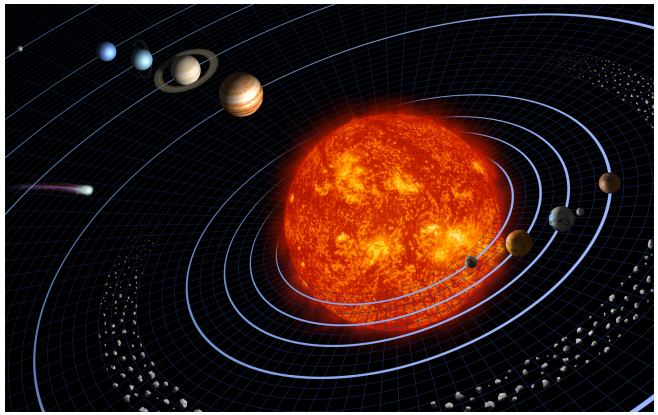


Granular systems

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

What is an ODE? – Examples

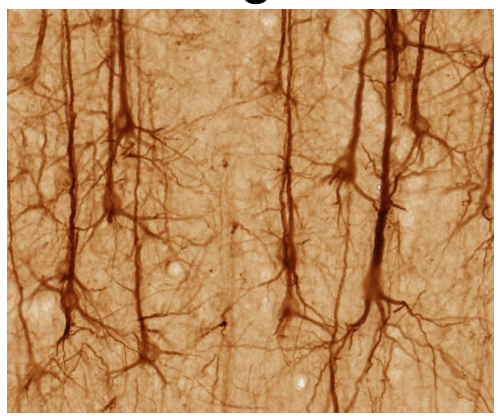
Newtons equations



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

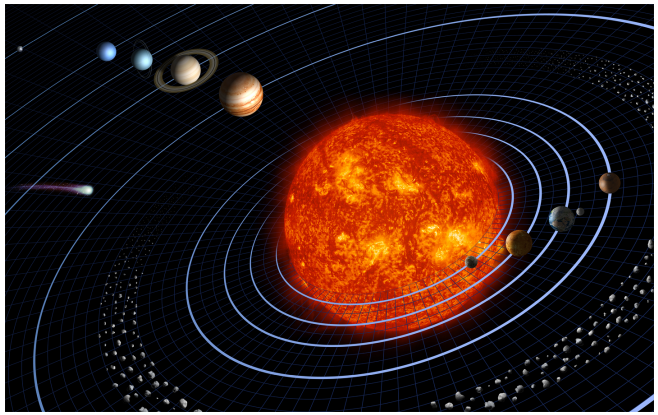
Granular systems

Interacting neurons



What is an ODE? – Examples

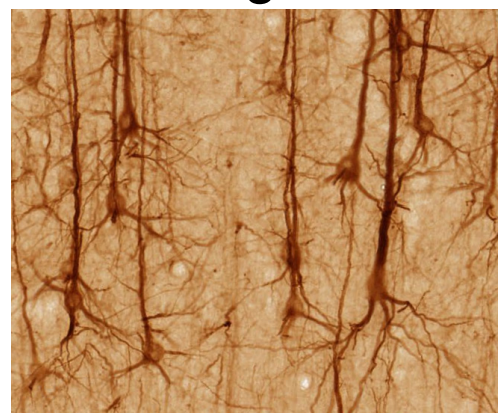
Newtons equations



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?

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

- $x(t)$ – 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

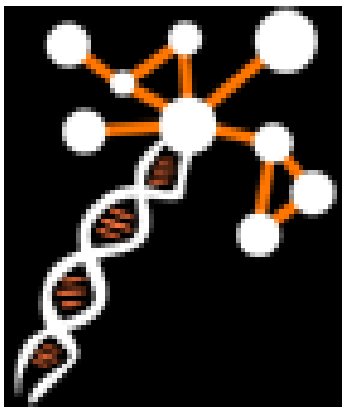
- **`www.odeint.com`**

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

We want to solve ODEs

- using `double`, `std::vector`, `std::array`, ... as state types.
- with complex numbers.
- on one, two and/or three-dimensional lattices.
- on Graphs.
- on graphic cards.
- with arbitrary precision types.

Container independent and **portable** algorithms are needed!

Motivation: The interface problem in C/C++, SKIP!!

- 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++, SKIP!!

- 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++, SKIP!!

- 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, SKIP!!

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, SKIP!!

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, SKIP!!

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

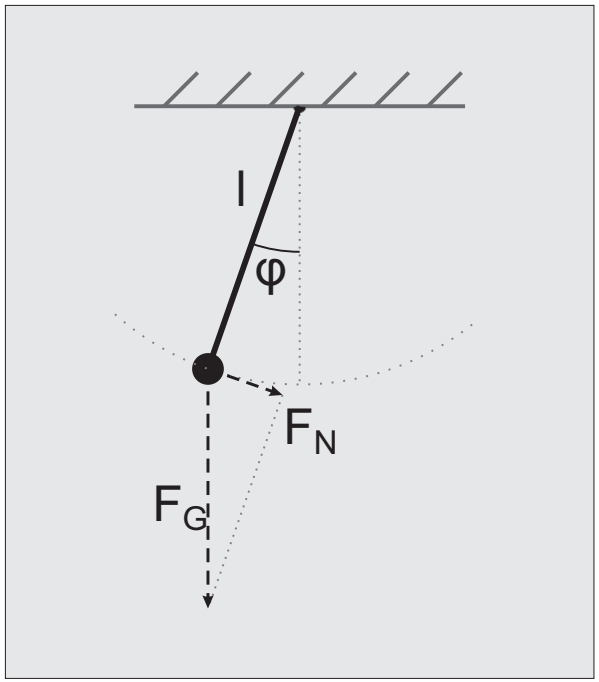
GoF Design Pattern: Strategy, also known as Policy

Alternative

Generic algorithms

Lets step into odeint

Example – Pendulum



Newtons law: $ma = F$

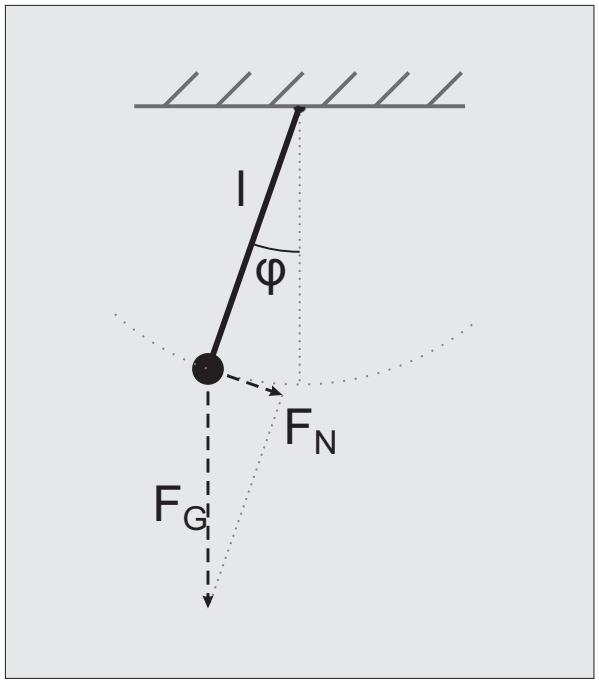
Acceleration: $a = l\ddot{\varphi}$

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

$\Longrightarrow$ **ODE for** φ

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

Example – Pendulum



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

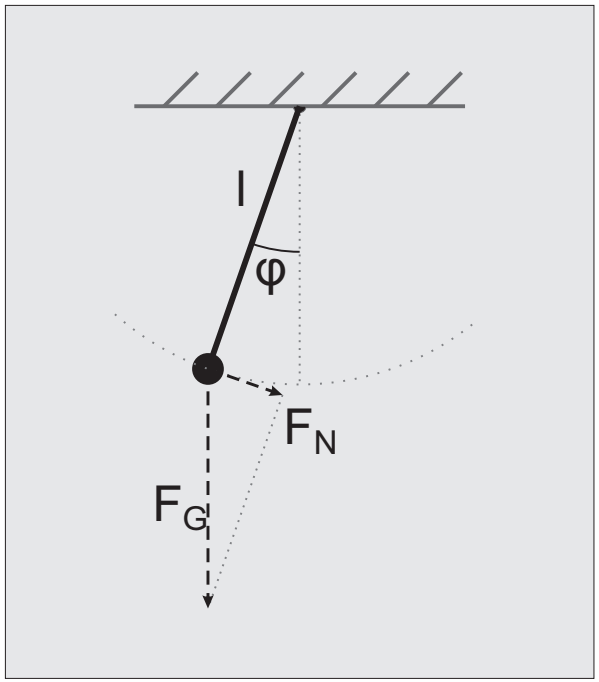
Small angle: $\sin\varphi \approx \varphi$

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: $\ddot{\varphi} = -\omega_0^2 \sin\varphi$

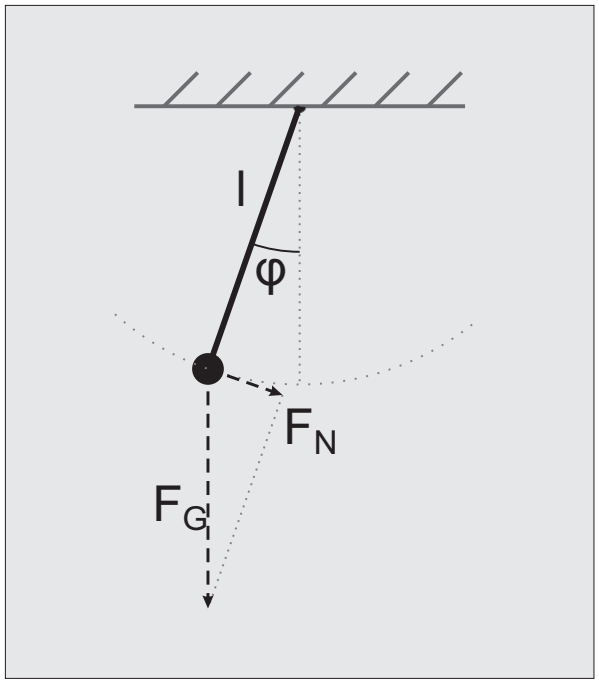
Pendulum with friction and external driving:

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

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

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

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

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

```
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

$$\varphi(0) = x_1(0) = 1 \ , \quad \dot{\varphi}(0) = x_2(0) = 0$$

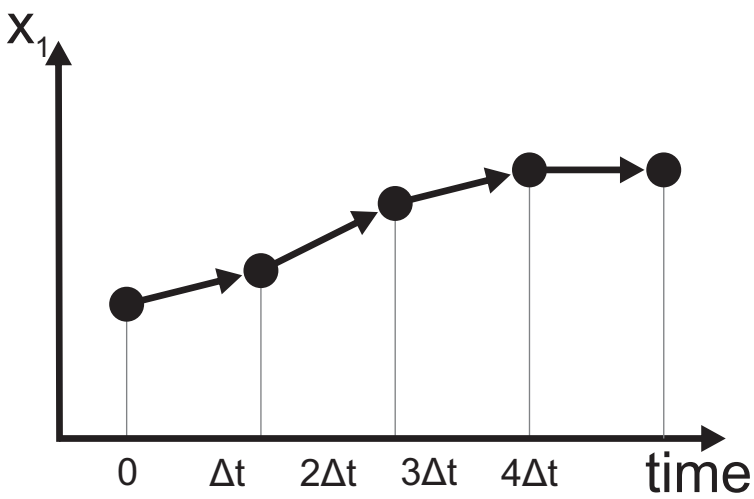
```
odeint::rk4< state_type > rk4;
pendulum p( 0.1 , 1.05 , 1.5 );

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;
```

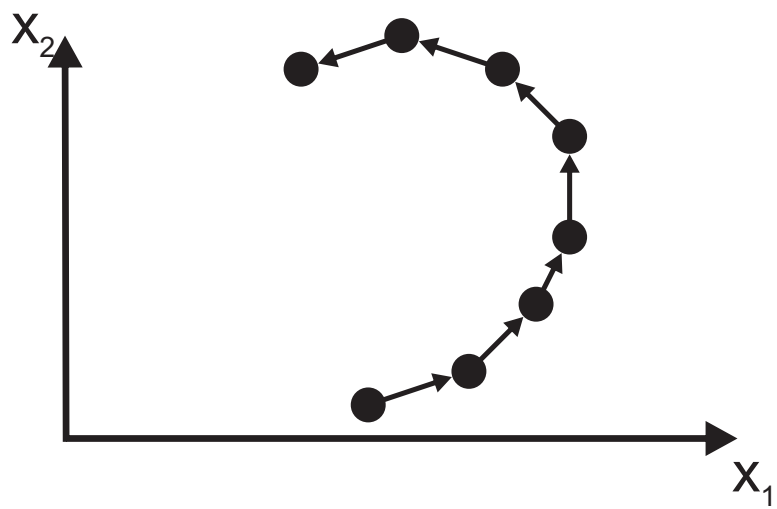
$$x(0) \mapsto x(\Delta t)$$

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";
}
```

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





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)

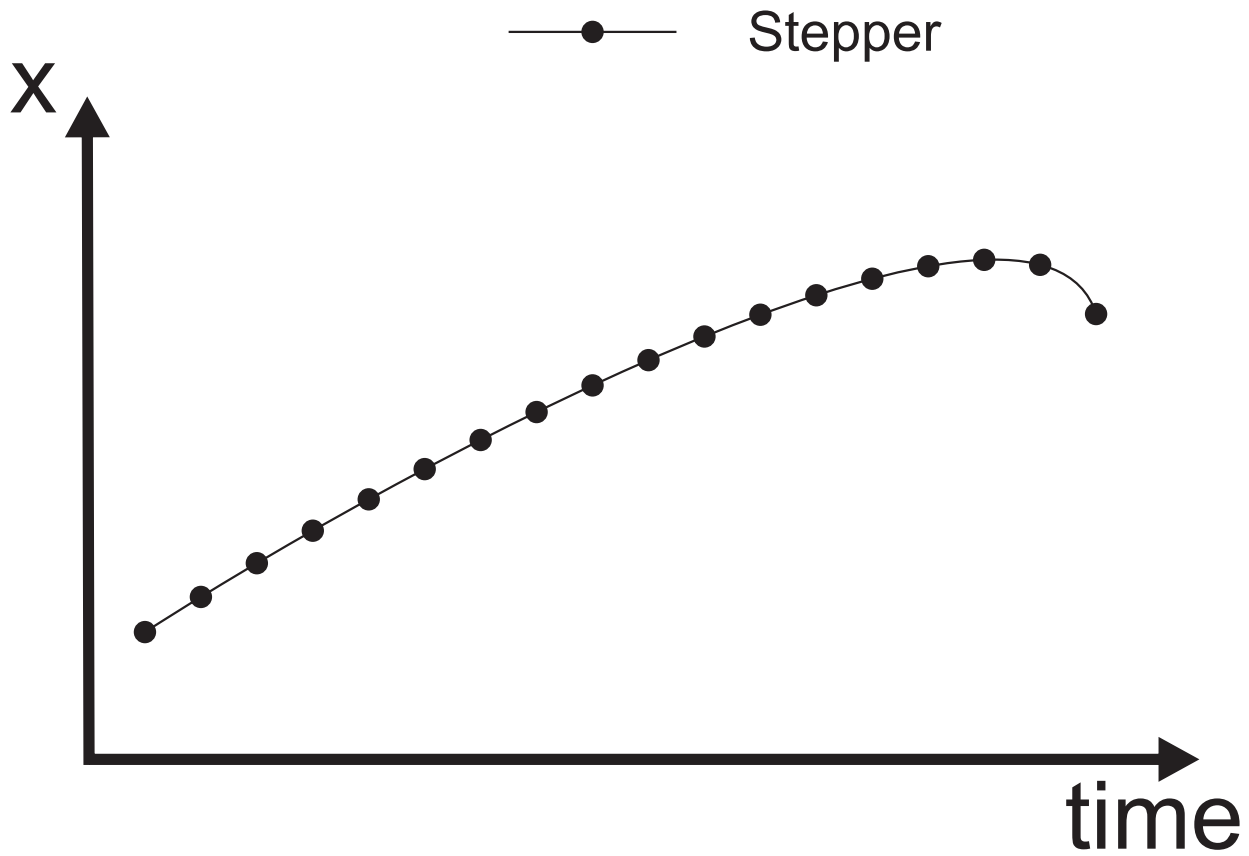
```
symplectic_rkn_sb3a_mclachlan< state_type > s;
```

Implicit steppers (for stiff systems)

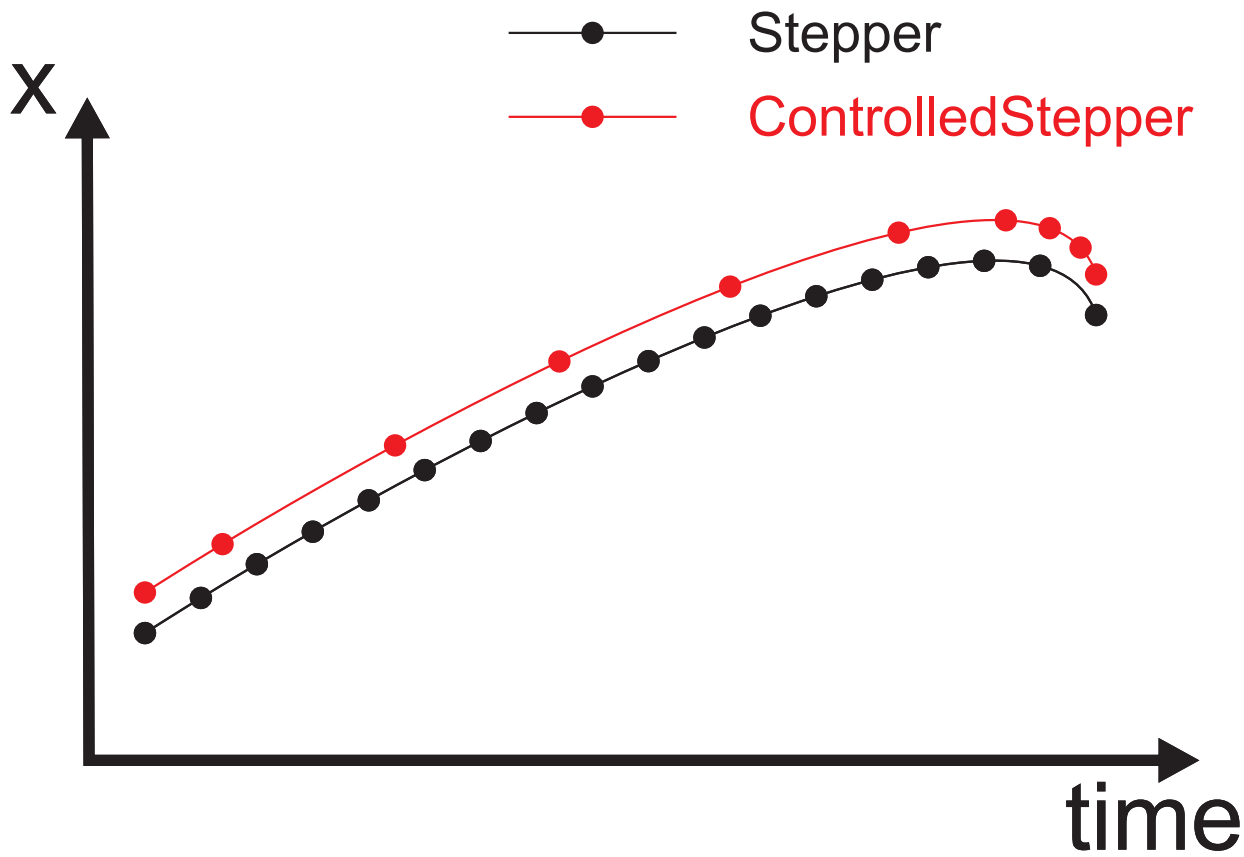
```
rosenbrock4< double > s;
```

These steppers perform one step with constant step size!

Controlled steppers – Step size control



Controlled steppers – Step size control



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`, ...

Adaptive step size vs. constant step size

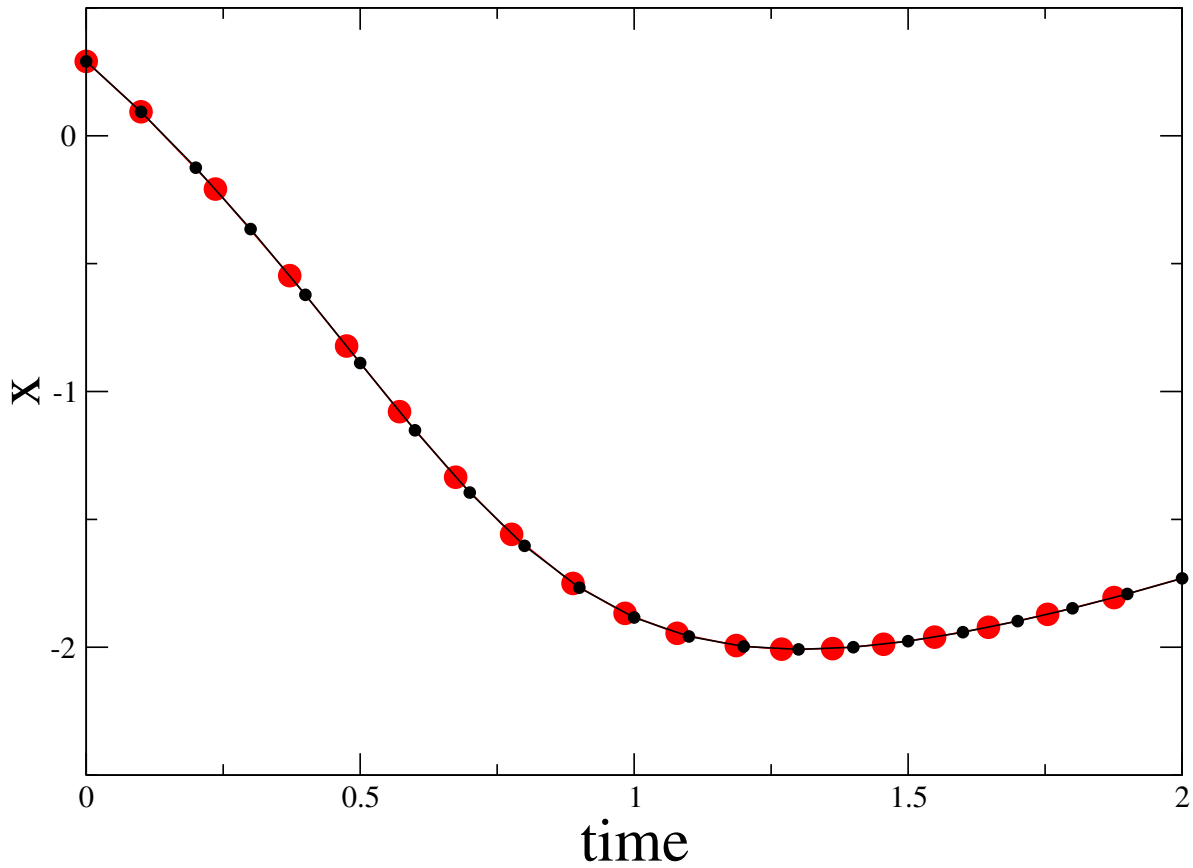
```
integrate_const(s,ode,x,t,dt,obs);
```

Performance loss when constant step size integration is needed

Adaptive step size vs. constant step size

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

Performance loss when constant step size integration is needed

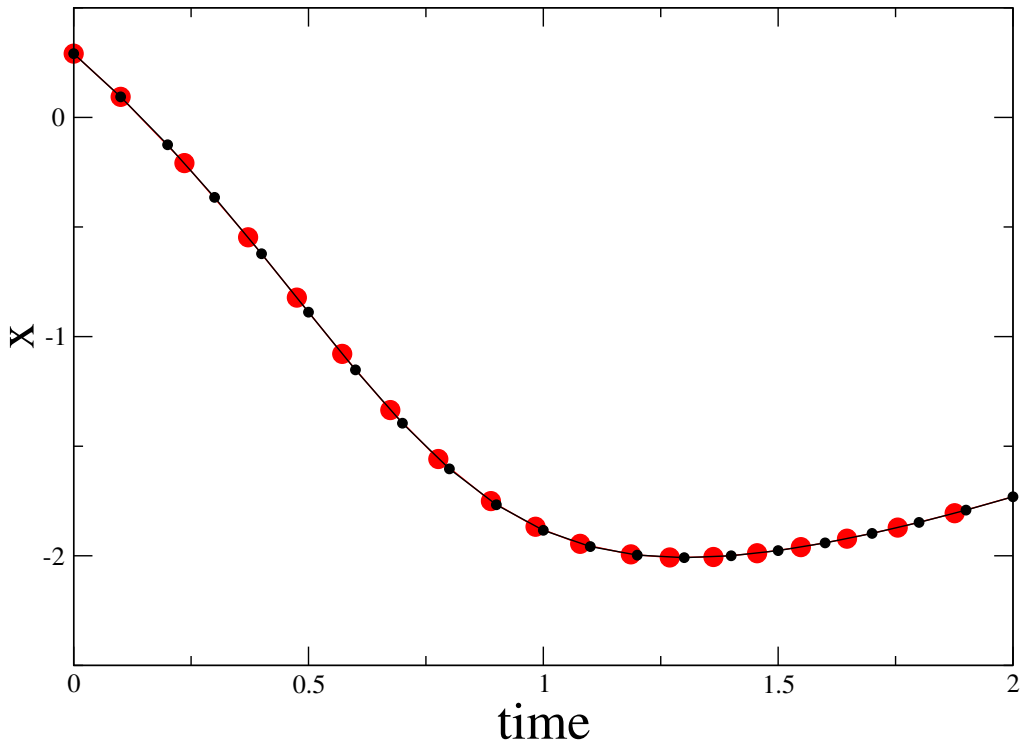


Dense output stepper

Problem: Adaptive step size vs. constant step size integration

```
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 the same precision as the stepper!



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 steppers – Steppers, Error steppers, Controlled steppers, Dense output steppers
- Integrate functions

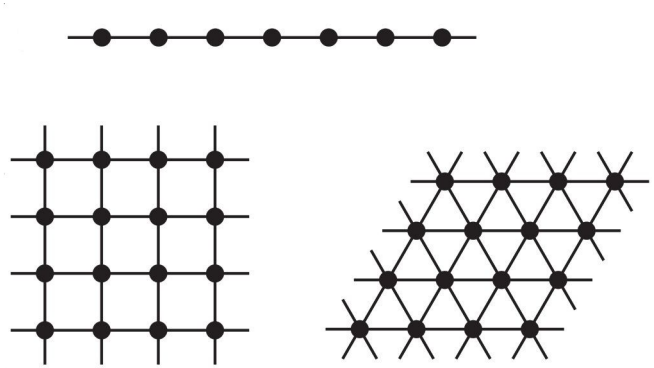
Small summary

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

Now, lets look at some advanced features!

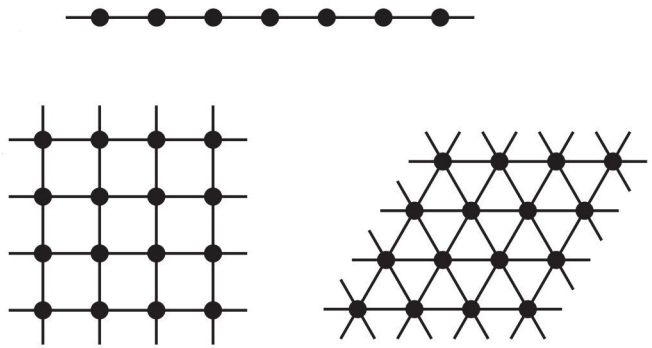
Large systems

Lattice systems



Large systems

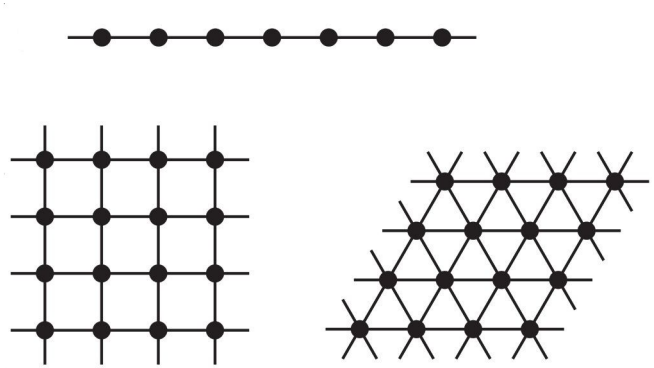
Lattice systems



Discretiztations of PDEs

Large systems

Lattice systems

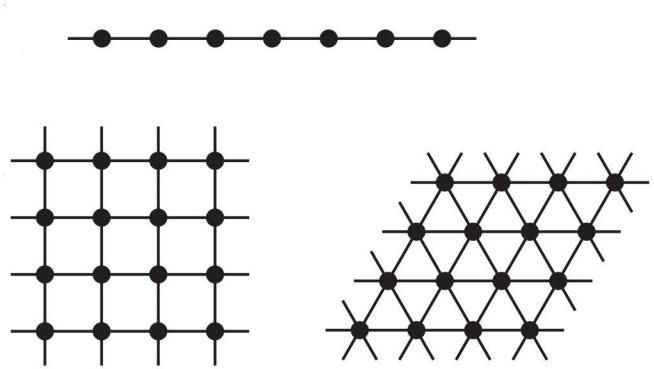


Discretiztations of PDEs

Granular systems

Large systems

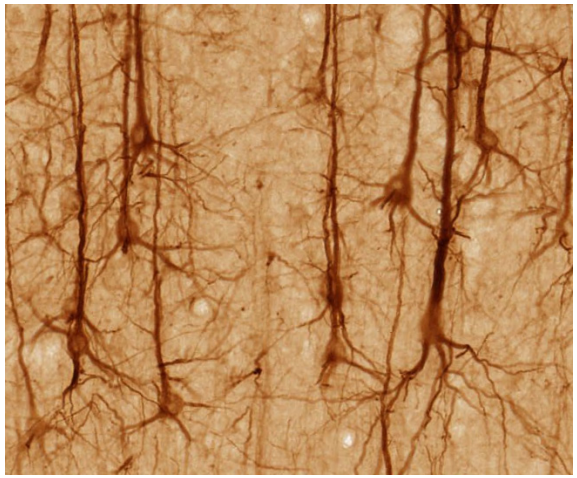
Lattice systems



Discretiztations of PDEs

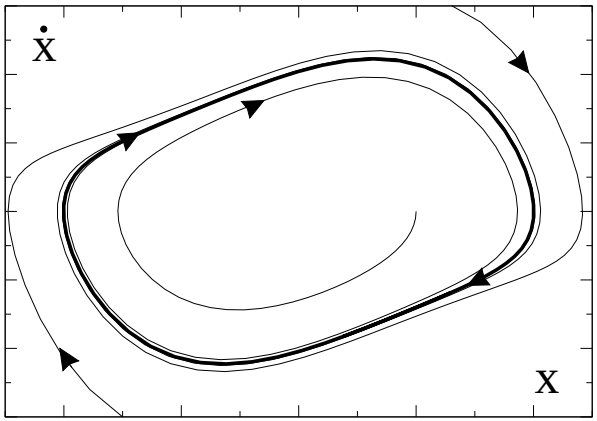
Granular systems

ODEs on graphs



High-Performance-Computing

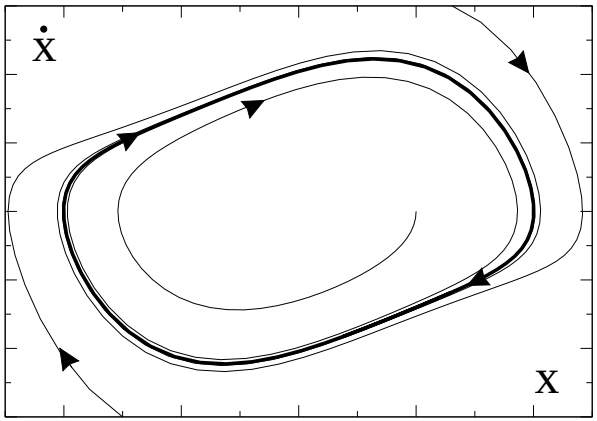
Coupled phase oscillators



Any oscillator can be described by one variable – its phase!

Trivial dynamic: $\dot{\varphi} = \omega$

Coupled phase oscillators



Any oscillator can be described by one variable – its phase!

Trivial dynamic: $\dot{\varphi} = \omega$

Interesting behaviour occurs if oscillators are coupled.

- Synchronization, oscillation death, phase chaos, pattern formation, ...

Applications: Neurosciences, Heart dynamics, social systems

Any weakly coupled oscillator system

$$\dot{\varphi}_k = \omega_k + q(\varphi_{k+1}, \varphi_k) + q(\varphi_k, \varphi_{k-1})$$

Phase compacton lattice

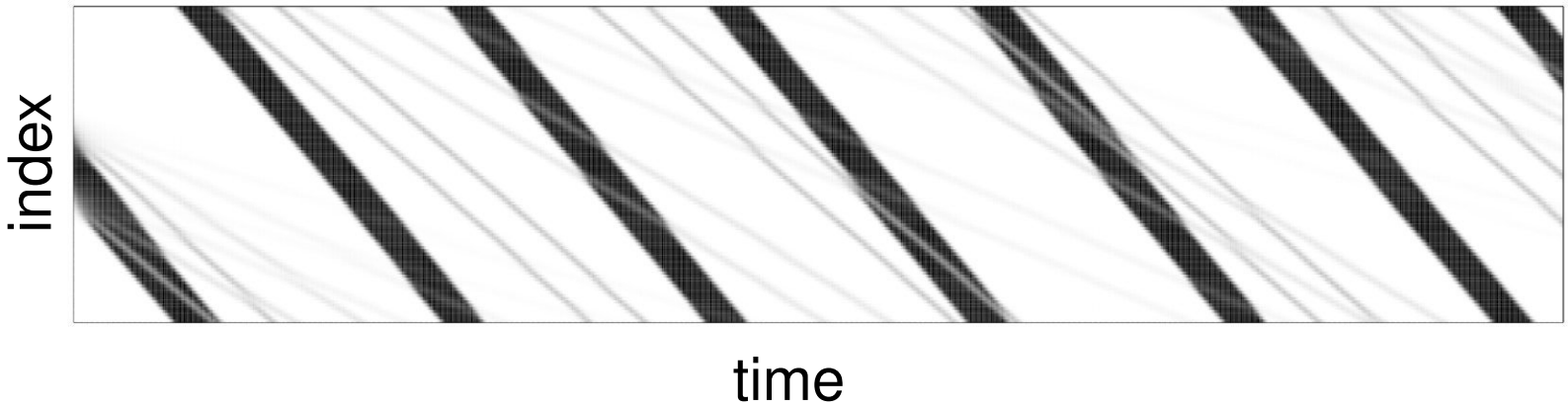
$$\dot{\varphi}_k = \cos\varphi_{k+1} - \cos\varphi_{k-1}$$

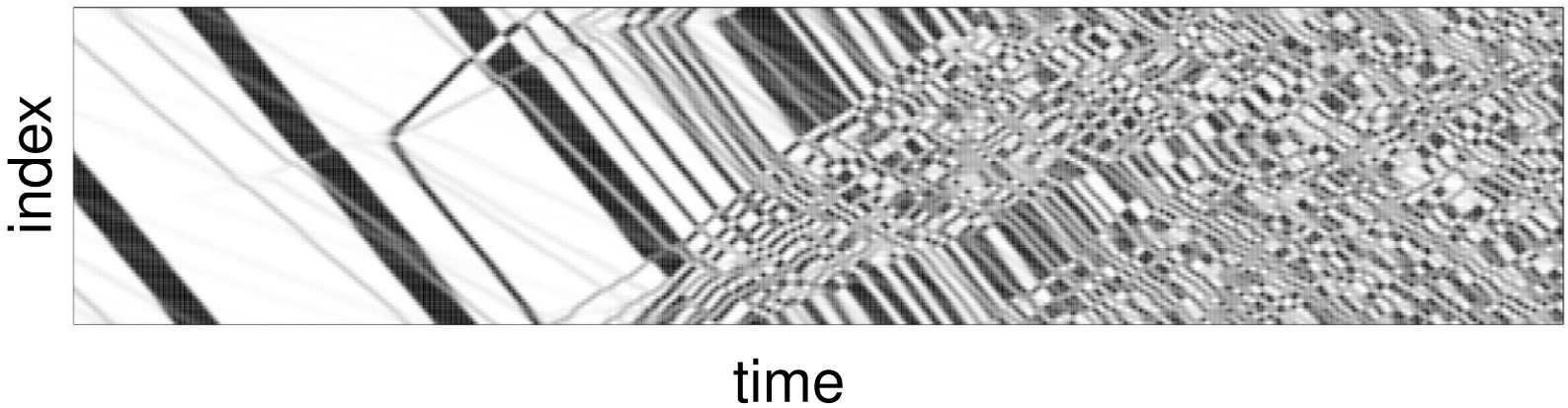
State space contains N variables

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

Animation

Phase compacton lattice – Transition to chaos





Ensemble of phase oscillators – SKIP

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

Synchronization – all oscillator oscillates with the same frequency

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

Ensemble of phase oscillators – SKIP

```
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:

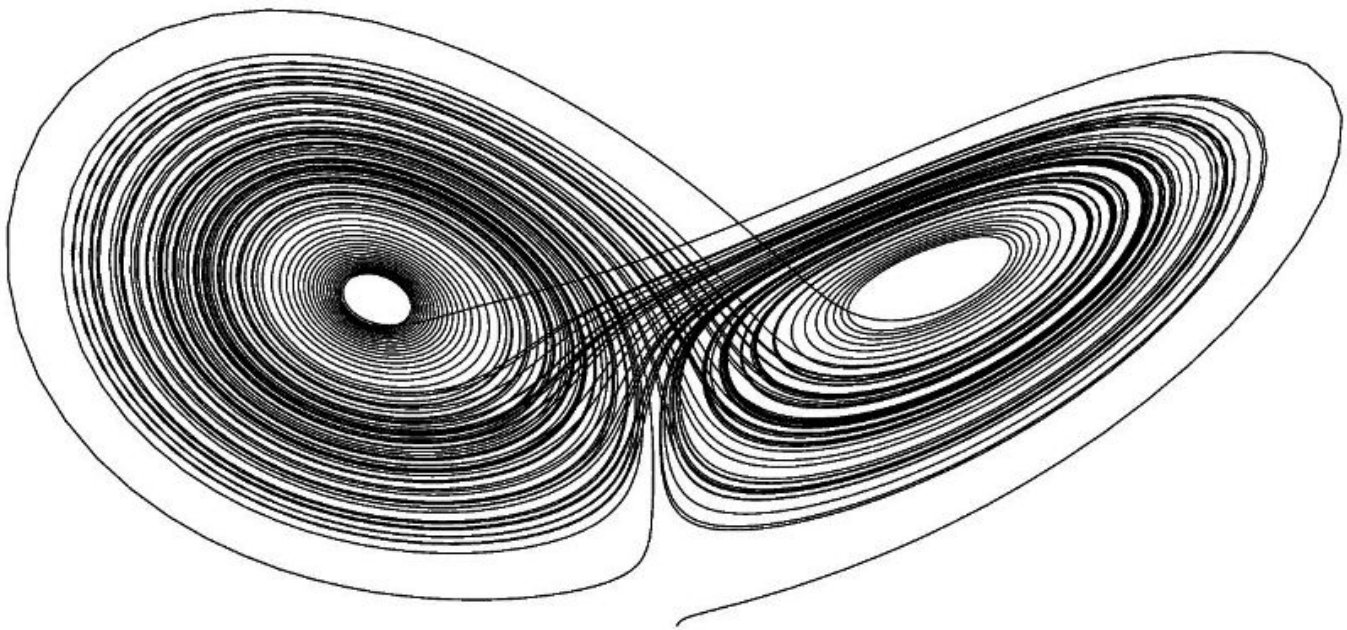
- Large systems, discretizations of PDEs, lattice systems, granular systems, etc.
- Parameter studies, solve many ODEs in parallel with different parameters
- Initial value studies, solve the same ODE with many different initial conditions in parallel

Lorenz system – Deterministic chaos

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

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

Perturbations grow exponentially fast – Butterfly effect



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!

Intermezzo: Algebras and operations

Euler method

$$x_i(t + \Delta t) = x_i(t) + \Delta t * f_i(x)$$

- Algebras perform the iteration over i
- Operations perform the elementary addition.

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

Intermezzo: Algebras and operations

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

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

Calculate an ensemble of Lorenz systems

```
typedef thrust::device_vector<double> state_type;
typedef runge_kutta4<state_type,double,state_type,double,
   thrust_algebra,thrust_operations,resizer> stepper;

state_type x( N );
// initialize x
integrate_const( stepper() , lorenz_ensemble() ,
   x , 0.0 , 1000.0 , dt );
```

Everything seems easy!

But how does `lorenz_ensemble` look like?

Ensemble of Lorenz systems

```
struct lorenz_ensemble {
  size_t N;
  state_type 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(
        x.begin() , x.begin()+N , x.begin()+2*N ,
        beta.begin() ,
        dxdt.begin(), dxdt.begin()+N, dxdt.begin()+2*N
      ) ) ,
      thrust::make_zip_iterator( thrust::make_tuple(
        x.begin()+N , x.begin()+2*N , x.begin()+3*N ,
        beta.end() ,
        dxdt.begin()+N,dxdt.begin()+2*N,dxdt.begin()+3*N
      ) ) ,
      lorenz_functor() );
  }

  // ...
};
```

Ensemble of Lorenz systems

```
struct lorenz_ensemble
{
  // ...

  struct lorenz_functor
  {
    template< class T > __host__ __device__
    void operator()( T t ) const
    {
      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 ;
    }
  };
};
```

Advanced features - continued

Reference wrapper `std::ref`, `boost::ref`

The ODE and the observers are always passed by value

```
integrate_const{s,ode,x,0.0,1.0,dt,obs);
s.do_step(ode,x,t,dt);
```

Reference wrapper `std::ref`, `boost::ref`

The ODE and the observers are always passed by value

```
integrate_const{s,ode,x,0.0,1.0,dt,obs);
s.do_step(ode,x,t,dt);
```

Use `std::ref` or `boost::ref` to pass by reference

```
integrate_const{s,std::ref(ode),x,0.0,1.0,dt,
  std::ref(obs));
```

Using Boost.Range

Use Boost.Range to integrate separate parts of the overall state

Example: Lyapunov exponents for the Lorenz system

Complete ODE = Lorenz system + Perturbation

- Calculate transients by solving only the Lorenz system (initialize x, y, z)
- Solve whole system (state + perturbations)

```
std::vector<double> x(6,0.0);
integrate(s,lorenz,make_pair(x.begin(),x.begin
    ()+3),0.0,10.0,dt);
integrate(s,lorenz_pert,x,10.0,1000.0,dt);
```

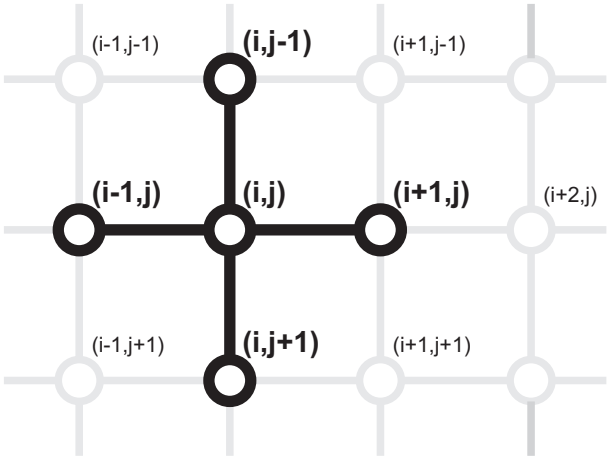
ODEs with complex numbers

Discrete Nonlinear Schrödinger equation

$$i\dot{\psi}_k = \varepsilon_k \psi_k + V(\psi_{k+1} + \psi_{k-1}) - \gamma|\psi_k|^2\psi_k \quad , \quad \psi_k \in \mathbb{C}$$

```
typedef std::vector<std::complex<double> > state_type;
struct dnls
{
  std::vector<double> eps;
  void operator()(const state_type &x, state_type &dxdt,
    double t) const
  {
    const double V=0.5 , gamma = 2.0;
    const complex<double> I(0.0,1.0);

    size_t N = x.size();
    dxdt[0] = dxdt[N-1] = 0.0;
    for(size_t i=1; i<N-1; ++i)
    {
      dxdt[i] = -I * ( eps[i]*x[i] + V*([x+1]+x[i-1])
        - gamma*norm(x[i])*x[i] );
    }
  }
};
```

Matrices as state types



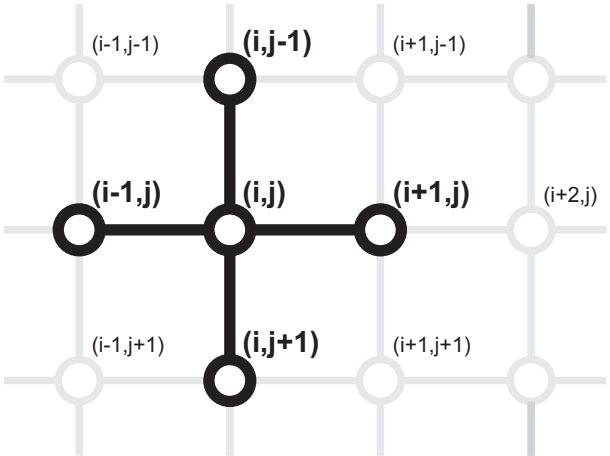
Example:

Two-dimensional phase lattice

$$\dot{\varphi}_{i,j} = q(\varphi_{i+1,j}, \varphi_{i,j}) + q(\varphi_{i-1,j}, \varphi_{i,j}) + q(\varphi_{i,j+1}, \varphi_{i,j}) + q(\varphi_{i,j-1}, \varphi_{i,j})$$

```
typedef ublas::matrix<double> state_type1;
typedef mtl::dense2D<double> state_type2;
```

Matrices as state types

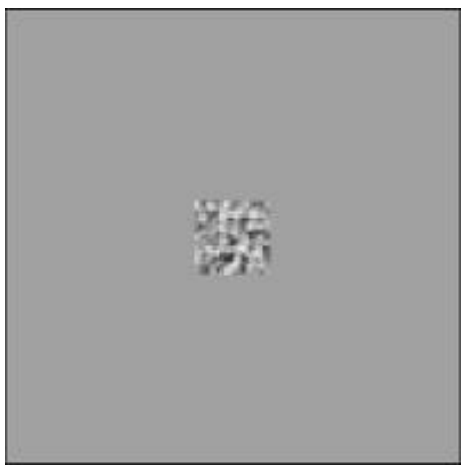


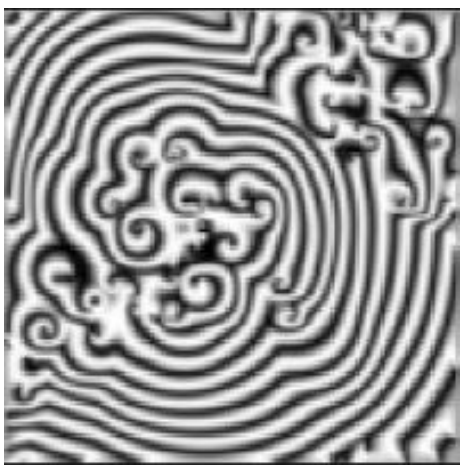
Example:

Two-dimensional phase lattice

$$\dot{\varphi}_{i,j} = q(\varphi_{i+1,j}, \varphi_{i,j}) + q(\varphi_{i-1,j}, \varphi_{i,j}) + q(\varphi_{i,j+1}, \varphi_{i,j}) + q(\varphi_{i,j-1}, \varphi_{i,j})$$

```
typedef ublas::matrix<double> state_type1;
typedef mtl::dense2D<double> state_type2;
```





Compile-time sequences and Boost.Units

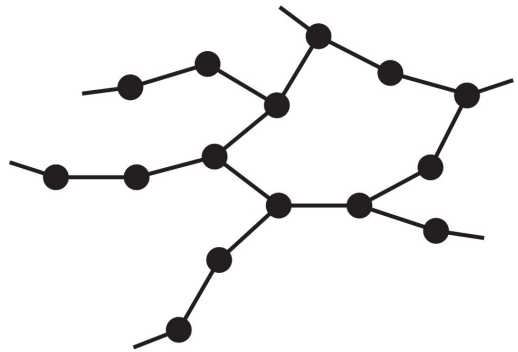
$$\begin{pmatrix} \dot{x} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v \\ f(x, v) \end{pmatrix}$$

- x – length, dimension m
- v – velocity, dimension ms^{-1}
- a – acceleration, dimension ms^{-2}

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

typedef runge_kutta_dopri5< state_type , double , deriv_type , time_type ,
     fusion_algebra > stepper_type;
```

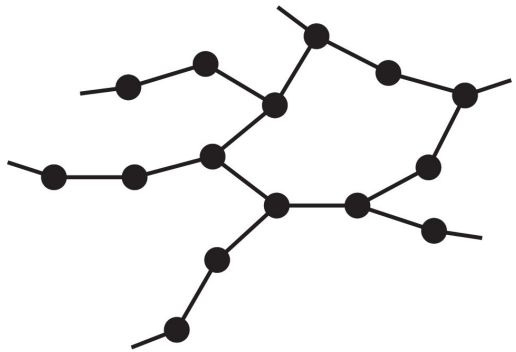
What else

- ODEs on graphs

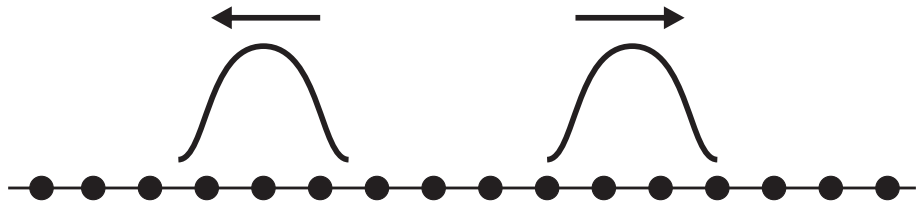


What else

- ODEs on graphs



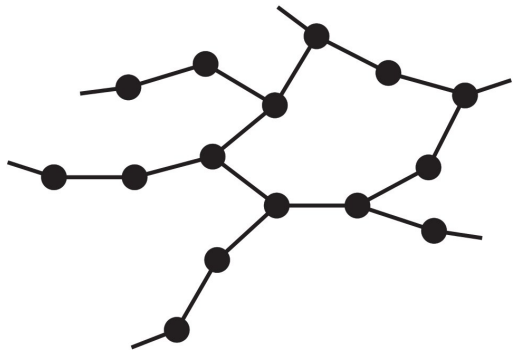
- Automatic memory management



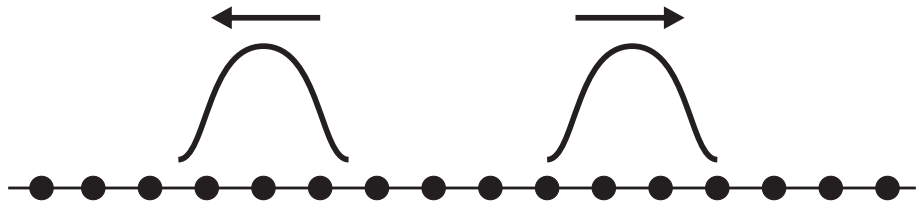
Enlarge the state when waves hit the boundary

What else

- ODEs on graphs



- Automatic memory management



Enlarge the state when waves hit the boundary

- Arbitrary precision types, GMPXX

Independent Algorithms

Goal

Container- and computation-independent implementation of the numerical algorithms.

Benefit

High flexibility and applicability, ODEINT can be used for virtually any formulation of an ODE.

Approach

Detatch the algorithm from memory management and computation detail and make each part interchangeable.

Mathematical Algorithm

Typical mathematical computation to calculate the solution of an ODE ($\dot{\vec{x}} = \vec{f}(\vec{x}, t)$):

$$
\begin{aligned}
\vec{F}_1 &= \vec{f}(\vec{x}_0, t_0) \\
\vec{x}' &= \vec{x}_0 + a_{21} \cdot \Delta t \cdot \vec{F}_1 \\
\vec{F}_2 &= \vec{f}(\vec{x}', t_0 + c_1 \cdot \Delta t) \\
\vec{x}' &= \vec{x}_0 + a_{31} \cdot \Delta t \cdot \vec{F}_1 + a_{32} \cdot \Delta t \cdot \vec{F}_2 \\
&\vdots \\
\vec{x}_1 &= \vec{x}_0 + b_1 \cdot \Delta t \cdot \vec{F}_1 + \cdots + b_s \cdot \Delta t \cdot \vec{F}_s
\end{aligned}
$$

Strucutural Requirements

$$\vec{F}_1 = \vec{f}(\vec{x}_0, t_0) \qquad \vec{x}' = \vec{x}_0 + a_{21} \cdot \Delta t \cdot \vec{F}_1$$

Types:

- vector type, mostly, but not neccessarily, some container like `vector<double>` (actually we have `state_type` and `deriv_type`)
- time type, usually `double`, but might be a multi-precision type
- value type, most likely the same as time type

Strucutural Requirements

$$\vec{F}_1 = \vec{f}(\vec{x}_0, t_0) \qquad \vec{x}' = \vec{x}_0 + a_{21} \cdot \Delta t \cdot \vec{F}_1$$

Types:

- vector type, mostly, but not neccessarily, some container like `vector<double>` (actually we have `state_type` and `deriv_type`)
- time type, usually `double`, but might be a multi-precision type
- value type, most likely the same as time type

Function Call:

```
void rhs( const vector_type &x , vector_type &
    dxdt , const time_type t )
{ /* user defined */ }

rhs( x0 , F1 , t ); //memory allocation for F1?
```

- Memory allocation for temporary results (`F`, `x'`)

Computational Requirements

$$\vec{x}_1 = \vec{x}_0 + b_1 \cdot \Delta t \cdot \vec{F}_1 + \cdots + b_s \cdot \Delta t \cdot \vec{F}_s$$

- vector-vector addition
- scalar-scalar multiplication
- scalar-vector multiplication

($\longrightarrow$ vector space)

Type Declarations

Tell ODEINT which types your are working with:

```
/* define your types */
typedef vector<double> state_type;
typedef vector<double> deriv_type;
typedef double value_type;
typedef double time_type;

/* define your stepper algorithm */
typedef runge_kutta4< state_type , value_type ,
    deriv_type , time_type > stepper_type;
```

Reasonable standard values for the template parameters allows for:

```
typedef runge_kutta4<state_type> stepper_type;
```

Memory Allocation / Resizing

Two possible situations: dynamic size / fixed size `vector_type`

dynamic size - memory allocation required

- e.g. `vector<double>`
- declare type as resizeable
- specialize resize template
- use `initially_resizer` or `always_resizer` in stepper algorithm

fixed size - memory allocation not required

- e.g. `array<double,N>`
- declare type as not resizeable
- that's it

Declare Resizeability

```
/* by default any type is not resizable */
template< class Container >
struct is_resizeable
{
    typedef boost::false_type type;
    const static bool value = type::value;
};

/* specialization for std::vector */
template< class T, class A >
struct is_resizeable< std::vector< T , A  > >
{
    typedef boost::true_type type;
    const static bool value = type::value;
};
```

To use a new dynamic sized type, this has to be specialized by the user.

Tell ODEINT how to resize

Again: only required if

`is_resizeable<state_type>::type == boost::true_type`.

Class Template responsible for resizing:

```
template< class StateOut , class StateIn >
struct resize_impl
{
    /* standard implementation */
    static void resize( StateOut &x1 , const
        StateIn &x2 )
    {
        x1.resize( boost::size( x2 ) );
    }
};
```

For anything that does not support `boost::size` or `resize` the user must provide a specialization.

Scalar Computations

For the scalar types we require the following:

Assume:

```
time_type t , dt;
value_type a1 , a2 , c;
```

Valid Expressions:

- `a1 = static_cast< value_type >(1)`
- `a1*a2`
- `a1/a2`
- `t + c*dt`
- `t + dt/c`
- `t += dt`

Vector Computations

Remember: $\vec{x}_1 = \vec{x}_0 + b_1 \cdot \Delta t \cdot \vec{F}_1 + \cdots + b_s \cdot \Delta t \cdot \vec{F}_s$

Split into two parts:

1. Algebra: responsible for iteration over vector elements
2. Operations: does the mathematical computation on the elements

Very similar to `std::for_each`, ODEINT uses something like:

```
Algebra algebra;

algebra.for_each3( x1 , x0 , F1 ,
    Operations::scale_sum3( 1.0, b1*dt );
```

Vector Computations

Remember: $\vec{x}_1 = \vec{x}_0 + b_1 \cdot \Delta t \cdot \vec{F}_1 + \cdots + b_s \cdot \Delta t \cdot \vec{F}_s$

Split into two parts:

1. Algebra: responsible for iteration over vector elements
2. Operations: does the mathematical computation on the elements

Very similar to `std::for_each`, ODEINT uses something like:

```
Algebra algebra;

algebra.for_each3( x1 , x0 , F1 ,
    Operations::scale_sum3( 1.0, b1*dt );
```

The types `Algebra` and `Operations` are template parameters of the steppers, hence exchangeable.

Vector Computations

```
time_type dt;
state_type x1, x2, ...
algebra_type algebra;
```

Algebra has to have defined the following member functions:

- `algebra.for_each1( x1 , unary_operation );`
- `algebra.for_each2( x1, x2, binary_operation );`
- `algebra.for_each3( ... );`

⋮

- `algebra.for_each15( .. , fifteen_ary_op );`

Vector Computations

```
time_type dt;
state_type x1, x2, ...
algebra_type algebra;
```

Algebra has to have defined the following member functions:

- `algebra.for_each1( x1 , unary_operation );`
- `algebra.for_each2( x1, x2, binary_operation );`
- `algebra.for_each3( ... );`

⋮

- `algebra.for_each15( .. , fifteen_ary_op );`

ODEINT takes the operations from the class `Operations`.

Operations

`Operations` is a class with the following member classes:

- `scale`
- `scale_sum1`
- `scale_sum2`

⋮

- `scale_sum14`

These classes need a constructor and `()`-operator that works together with the algebra:

```
value_type b1, b2;
time_type dt;
algebra.for_each4( x1 , x0 , F1 ,
   Operations::scale_sum3( 1.0, b1*dt );
```

This computes: $\vec{x}_1 = 1.0 \cdot \vec{x}_0 + b_1 \Delta t \cdot \vec{F}_1$.

Example Implementation: `range_algebra`

```
struct range_algebra {
...
 template< class S1 , class S2 , class S3 , class Op >
 static void for_each2( S1 &s1, S2 &s2, S3 &s3, Op op )
 {
   detail::for_each2( boost::begin(s1), boost::end(s1),
                      boost::begin(s2), boost::begin(s3),
                      op );
 }
...
};

namespace detail {
...
 template< class Iter1 , class Iter2 , Iter3 , class Op >
 void for_each2( Iter1 first1 , Iter1 last1 ,
                 Iter2 first2 , Iter3 first3 , Op op )
 {
     for( ; first1 != last1 ; )
         op( *first1++ , *first2++ , *first3++ );
 }
...
};
```

Example Implementation: `default_operations`

```
template< class Fac1 , class Fac2 >
struct scale_sum2
{
  const Fac1 m_alpha1;
  const Fac2 m_alpha2;

  scale_sum2( Fac1 &alpha1 , Fac2 &alpha2 )
    : m_alpha1( alpha1 ) , m_alpha2( alpha2 )
  { }

  template< class T1 , class T2 , class T3 >
  void operator()( T1 &t1 , const T2 &t2 ,
     const T3 &t3 )
  { t1 = m_alpha1 * t2 + m_alpha2 * t3; }

  typedef void result_type;
};
```

`range_algebra & default_operations` can be used with any Container that supports Boost.Range and whose `container::value_type` supports operators `+, *`.
For example `vector< double >`:

```
typedef vector< double > state_type;
typedef double value_type;
typedef double time_type;

typedef runge_kutta4< state_type , value_type ,
                      state_type , time_type ,
                      range_algebra ,
                      default_operations
                    > stepper_type
```

As these are also the default values, this can be shortened:

```
typedef runge_kutta4<state_type> stepper_type;
```

`range_algebra & default_operations` work also with

- `vector< complex<double> >`
- `list< double >`
- `array< double , N >`

`range_algebra & default_operations` work also with

- `vector< complex<double> >`
- `list< double >`
- `array< double , N >`

What about

- Ublas `vector`
- trivial `state_type`s like `double`
- generally: `state_type` that support operators `+, *`

$\longrightarrow$ `vector_space_algebra`!

vector_space_algebra

```
struct vector_space_algebra {
...
 template< class S1 , class S2 , class S3 ,
     class Op >
 static void for_each3( S1 &s1 , S2 &s2 ,
                        S3 &s3 , Op op )
 {
   op( s1 , s2 , s3 );
 }
...
};
```

- delegates operations directly to the `state_type`
- no iteration
- works together with `default_operations` with any `state_type` that supports operators `+,*`

Other Examples

fusion_algebra: works with `fusion::vector` as `state_type`, (e.g. Boost.Units)

thrust_algebra & thrust_operations: Use thrust library to perform computation on CUDA graphic cards

nested_algebra: can handle nested containers that support Boost.Range, e.g. `vector< vector<double> >`

See tutorial and documentation on `www.odeint.com` for more.

Other Examples

fusion_algebra: works with `fusion::vector` as `state_type`, (e.g. Boost.Units)

thrust_algebra & thrust_operations: Use thrust library to perform computation on CUDA graphic cards

nested_algebra: can handle nested containers that support Boost.Range, e.g. `vector< vector<double> >`

See tutorial and documentation on `www.odeint.com` for more.

Important

Division into Algebra and Operations gives us great flexibility. However, State_Type, Algebra and Operations must coorporate to make ODEINT work!