

Metaprogramming Applied to Numerical Problems

A Generic Implementation of Runge-Kutta Algorithms

Mario Mulansky
Karsten Ahnert

C++ Usergroup Berlin

June 17, 2014



Metaprogramming

Wikipedia:

Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime.

Metaprogramming

Wikipedia:

Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime.

In Short:

Metaprogramming means to write a program that writes a program that solves your problem.

Template Metaprogramming

- Here, C++ Templates will be used for Metaprogramming
- This technique is called **Template Metaprogramming**
- It basically means using the template engine to generate a program from which the compiler creates then the binary
- C++ compilers always use the template engine (no additional compile step required)

Template Metaprogramming

- Here, C++ Templates will be used for Metaprogramming
- This technique is called **Template Metaprogramming**
- It basically means using the template engine to generate a program from which the compiler creates then the binary
- C++ compilers always use the template engine (no additional compile step required)
- Template engine and templates form a Turing-complete programming language
- → You can solve any problem with the template engine

Example: Calculate factorial at compile time

Compute the factorial of n :

```
template< int n >
struct factorial {
    // recursive definition
    static const int value = factorial<n-1>::value*n;
};

template<>
struct factorial< 0 > {
    // condition to end recursion
    static const int value = 1;
};
```

`factorial<5>::value` gives $5! = 120$, but computed at **compile time!**

Why use Metaprogramming in numeric algorithms

- Increase run time performance by shifting some calculations to compile time (but no floating point arithmetics!)
- Leads to more generic algorithms with no performance loss

Why use Metaprogramming in numeric algorithms

- Increase run time performance by shifting some calculations to compile time (but no floating point arithmetics!)
- Leads to more generic algorithms with no performance loss

Downfalls:

- Longer compile time
- Complicated compiler errors

A generic pow routine

- Compute x^n where $n > 0$ is known at compile time.
- Intelligent algorithm: $x^8 = ((x^2)^2)^2$, $\log_2 n + 1$ multiplications
- Implemented in `boost/math/special_functions/pow.hpp`

A generic pow routine

```
template < int N , int M = N%2 >
struct pow_impl
{ // implementation for N even
    static double result( double base )
    {
        double power = pow<N/2>::result( base );
        return power*power;
    }
};

template <int N>
struct pow_impl< N , 1 >
{ // specialization for N odd
    double operator()( double base )
    {
        double power = pow<N/2>::result( base );
        return base*power*power;
    }
};
```

A generic pow routine

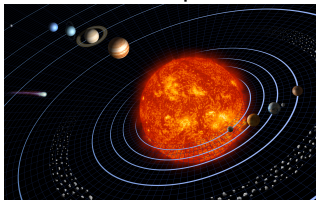
```
template <>
struct pow_impl< 1 , 1 >
{ // specialization to stop recursion
  double operator()( double base )
  { return base; }
};

template< int N >
double pow( double base ) { //convenience function
  return pow_impl<N>()::result( base );
}
```

Using `pow<8>(x)` the compiler sees `t1=x*x; t2=t1*t1; return t2*t2;`

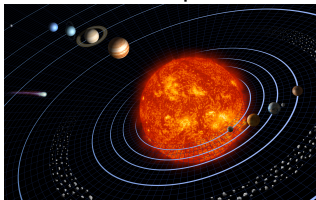
Ordinary Differential Equations

Newtons equations



Ordinary Differential Equations

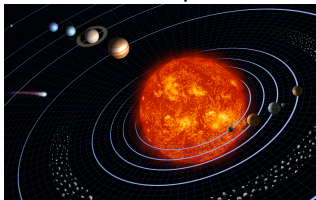
Newtons equations



Reaction and relaxation
equations (i.e. blood alcohol
content, chemical reaction
rates)

Ordinary Differential Equations

Newtons equations



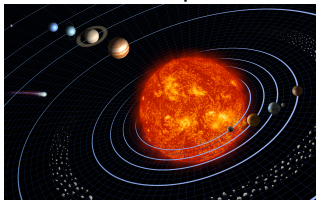
Reaction and relaxation equations (i.e. blood alcohol content, chemical reaction rates)

Granular systems



Ordinary Differential Equations

Newtons equations

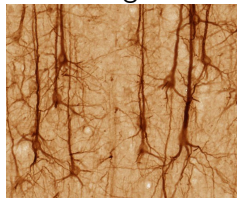


Reaction and relaxation equations (i.e. blood alcohol content, chemical reaction rates)

Granular systems

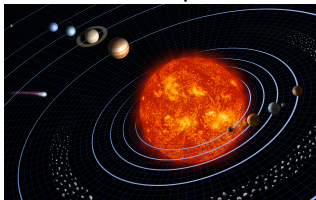


Interacting neurons



Ordinary Differential Equations

Newtons equations

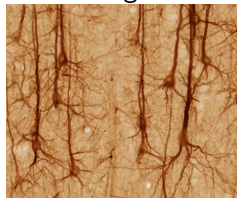


Reaction and relaxation equations (i.e. blood alcohol content, chemical reaction rates)

Granular systems



Interacting neurons



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

Ordinary Differential Equations

A first order ODE is written in its most general form as:

$$\frac{d}{dt}\vec{x}(t) = \vec{f}(\vec{x}, t) \quad (1)$$

- $\vec{x}(t)$ is the function in demand (here: trajectory)
- t is the independent variable (here: time)
- $f(x, t)$ is the rhs, governing the behavior of x

Initial Value Problem (IVP):

$$\dot{x} = f(x, t), \quad x(t = 0) = x_0 \quad (2)$$

Examples

- $\dot{x} = -\lambda x$ solution: $x(t) = x_0 e^{-\lambda t}$
- $\ddot{x} = \omega^2 x \rightarrow \begin{cases} \dot{x} = p \\ \dot{p} = -\omega^2 x \end{cases}$ solution: $x(t) = A \sin(\omega t + \varphi_0)$.

Examples

- $\dot{x} = -\lambda x$ solution: $x(t) = x_0 e^{-\lambda t}$
- $\ddot{x} = \omega^2 x \rightarrow \begin{cases} \dot{x} = p \\ \dot{p} = -\omega^2 x \end{cases}$ solution: $x(t) = A \sin(\omega t + \varphi_0)$.
 $\dot{x} = \sigma(y - x)$
- Lorenz System: $\dot{y} = x(R - z) - y$ solution: ?
 $\dot{z} = xy - \beta z$.

Chaotic system (for certain parameter values σ, R, β), hence the solution can not be written in analytic form.

Examples

- $\dot{x} = -\lambda x$ solution: $x(t) = x_0 e^{-\lambda t}$
- $\ddot{x} = \omega^2 x \rightarrow \begin{cases} \dot{x} = p \\ \dot{p} = -\omega^2 x \end{cases}$ solution: $x(t) = A \sin(\omega t + \varphi_0)$.
- $\dot{x} = \sigma(y - x)$
- Lorenz System: $\dot{y} = x(R - z) - y$ solution: ?
 $\dot{z} = xy - \beta z$.

Chaotic system (for certain parameter values σ, R, β), hence the solution can not be written in analytic form.

\Rightarrow numerical methods to solve ODEs are required for more complicated systems.

Runge-Kutta Scheme

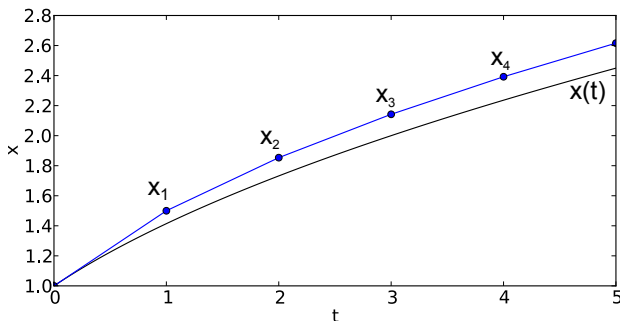
One class of algorithms to solve IVP of ODEs.

- Discretized time $t \rightarrow t_n = t_0 + n \cdot h$ with (small) time step h
- Trajectory $x(t) \rightarrow x_n \approx x(t_n)$
- Iteration along trajectory: $x_0 \longrightarrow x_1 \longrightarrow x_2 \dots$
- One-step method: $x_1 = \Phi(x_0)$, $x_2 = \Phi(x_1)$, \dots

Runge-Kutta Scheme

One class of algorithms to solve IVP of ODEs.

- Discretized time $t \rightarrow t_n = t_0 + n \cdot h$ with (small) time step h
- Trajectory $x(t) \rightarrow x_n \approx x(t_n)$
- Iteration along trajectory: $x_0 \rightarrow x_1 \rightarrow x_2 \dots$
- One-step method: $x_1 = \Phi(x_0)$, $x_2 = \Phi(x_1)$, ...



Runge-Kutta Scheme

Numerically solve the Initial Value Problem (IVP) of the ODE:

$$\dot{x}(t) = f(x, t), \quad x(t = 0) = x_0. \quad (3)$$

A Runge-Kutta scheme with s stages and parameters $c_1 \dots c_s$, $a_{21}, a_{31}, a_{32}, \dots, a_{ss-1}$ and $b_1 \dots b_s$ gives the approximate solution for $x_1 \approx x(h)$ starting at x_0 by computing:

$$x_1 = x_0 + h \sum_{i=1}^s b_i F_i \quad \text{where} \quad F_i = f\left(x_0 + h \sum_{j=1}^{i-1} a_{ij} F_j, h c_i\right). \quad (4)$$

This approximate solution x_1 is exact up to some order p .
Repeating the whole procedure brings you from x_1 to x_2 , then to x_3 and so on.

At each stage i the following calculations have to be performed ($y_1 = x_0$) :

$$F_i = f(y_i, hc_i), \quad y_{i+1} = x_0 + h \sum_{j=1}^i a_{i+1,j} F_j, \quad i = 1 \dots s-1$$
$$F_s = f(y_s, hc_s), \quad x_1 = x_0 + h \sum_{j=1}^s b_j F_j.$$

The parameters a , b and c define the so-called Butcher tableau.

Butcher Tableau

Parameters a , b , and c are typically written as Butcher tableau:

c_1					
c_2	$a_{2,1}$				
c_3	$a_{3,1}$	$a_{3,2}$			
\vdots	\vdots		\ddots		
c_s	$a_{s,1}$	$a_{s,2}$	\dots	$c_{s,s-1}$	
	b_1	b_2	\dots	b_{s-1}	b_s

The Butcher Tableau fully defines the Runge-Kutta scheme. Each line of the tableau represents one stage of the scheme.

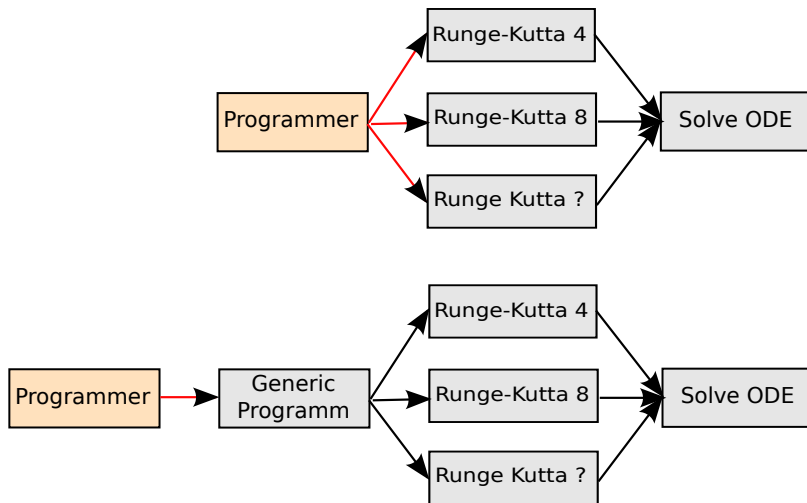
Generic Runge-Kutta Algorithm

Idea:

- Write a Metaprogram that creates Runge-Kutta algorithms
- Metaprogram input: Parameters of the RK scheme (Butcher Tableau)
- Main objective: **Resulting program should be as fast as direct implementation**

With such a Metaprogram you can implement any new Runge-Kutta scheme by just providing the Butcher tableau.

- Decrease in programming time
- Less bugs
- Better maintainability



Explicit Non-Generic Implementation

Given parameters c_i , a_{ij} , b_i

```
F_1 = f( x , t + c_1*dt );  
x_tmp = x + dt*a_21 * F_1;  
  
F_2 = f( x_tmp , t + c_2*dt );  
x_tmp = x + dt*a_31 * F_1 + dt*a_32 * F_2;  
  
// ...  
  
F_s = f( x_tmp , t + c_s*dt );  
x_end = x + dt*b_1 * F_1 + dt*b_2 * F_2 + ...  
          + dt*b_s * F_s;
```

Not generic: Each stage written hard coded – you have to adjust the algorithm when implementing a new scheme.

Run Time Implementation

Given parameters $a[] []$, $b[]$, $c[]$.

```
F[0] = f( x , t + c[0]*dt );
x_tmp = x + dt*a[0][0] * F[0];

for( int i=1 ; i<s-1 ; ++i )
{
    F[i] = f( x_tmp , t + c[i]*dt );
    x_tmp = x;
    for( int j=0 ; j<i+1 : ++j )
        x_tmp += dt*a[i][j] * F[j];
}

F[s-1] = f( x_tmp , t + c[s-1]*dt );
x_end = x;
for( int j=0 ; j<s : ++j )
    x_end += dt*b[j] * F[j];
```

Run Time Implementation

Given parameters $a[] []$, $b[]$, $c[]$.

```
F[0] = f( x , t + c[0]*dt );
x_tmp = x + dt*a[0][0] * F[0];

for( int i=1 ; i<s-1 ; ++i )
{
    F[i] = f( x_tmp , t + c[i]*dt );
    x_tmp = x;
    for( int j=0 ; j<i+1 : ++j )
        x_tmp += dt*a[i][j] * F[j];
}

F[s-1] = f( x_tmp , t + c[s-1]*dt );
x_end = x;
for( int j=0 ; j<s : ++j )
    x_end += dt*b[j] * F[j];
```

Generic, but factor 2 slower than explicit implementation!

Why Bad Performance

The run time generic code is hard to optimize for the compiler, because:

- Double `for` loop with inner bound depending on outer loop variable.
- 2D array `double** a` must be dynamically allocated:

```
a = new double*[s];  
for( int i=0 ; i<s ; ++i )  
    a[i] = new double[i+1];  
a[0][0] = ...;  
a[1][0] = ...; a[1][1] = ...;  
...
```

→ lives on heap, harder to be optimized compared to stack.

- Many more issues possible (optimizers are rather complex).

What to do?

Idea:

Use template engine to generate code that can be efficiently optimized by the Compiler.

What to do?

Idea:

Use template engine to generate code that can be efficiently optimized by the Compiler.

More specifically, we will use Template Metaprogramming to:

- Generate fixed size arrays: `a_1[1]` , `a_2[2]` , ... , `a_s[s]`
- Unroll the outer `for`-loop (over stages `s`) so the compiler sees sequential code.

As result, the code seen by the compiler/optimizer (after resolving templates) is very close to the non-generic version and thus as fast, hopefully.

The Generic Implementation

Define a structure representing one stage of the Runge-Kutta scheme:

```
template< int i >
struct stage // general (intermediate) stage,  $i > 0$ 
{
    double c; // parameter  $c_i$ 
    array<double,i> a; // parameters  $a_{i+1,i} \dots a_{i,i}$ 
                        //  $b_1 \dots b_j$  for the last stage
};
```

Given an instance of this stage with c and a set appropriately the corresponding Runge-Kutta stage can be calculated.

The Generic Implementation

```
// x , x_tmp , t , dt and F defined outside
template< int i >
void calc_stage( const stage< i > &stage )
{ // performs the calculation of the i-th stage
  if( i == 1 ) // first stage?
    F[i-1] = f( x , t + stage.c * dt );
  else
    F[i-1] = f( x_tmp , t + stage.c * dt );

  if( i < s ) { // intermediate stage?
    x_tmp = x;
    for( int j=0 ; j<i : ++j )
      x_tmp += dt*stage.a[j] * F[j];
  } else { // last stage
    x_end = x;
    for( int j=0 ; j<i : ++j )
      x_end += dt*stage.a[j] * F[j];
  }
}
```

The Generic Implementation

Generate list of stage types: stage<1> , stage<2>, ... , stage<s>
using Boost.MPL (MetaProgramming Library) and Boost.Fusion.

```
typedef mpl::range_c< int , 1 , s > stage_indices;

typedef typename fusion::result_of::as_vector
< typename mpl::push_back
  < typename mpl::copy
    < stage_indices,
      mpl::inserter
    <
      mpl::vector0<> ,
      mpl::push_back< mpl::_1 , stage_wrapper< mpl::_2 , stage > >
    >
  >::type , stage< double , stage_count , last_stage >
>::type
>::type stage_vector_base; //fusion::vector< stage<1> , stage<2> , ... , stage<s>

struct stage_vector : stage_vector_base
{
  // initializer methods
  stage_vector( const a_type &a , const b_type &b , const c_type &c )
  {
    // ...
  }
}
```

The Generic Implementation

Parameter types for a, b and c:

```
typedef typename fusion::result_of::as_vector
< typename mpl::copy
  < stage_indices ,
    mpl::inserter
      < mpl::vector0< > ,
        mpl::push_back< mpl::_1 ,
          array_wrapper< double , mpl::_2 > >
        >
      >::type
>::type a_type; //fusion::vector< array<double,1> , array<double,2> , ... >

typedef array< double , s > b_type;
typedef array< double , s > c_type;
```

The Generic Implementation

Parameter types for a, b and c:

```
typedef typename fusion::result_of::as_vector
< typename mpl::copy
  < stage_indices ,
    mpl::inserter
      < mpl::vector0< > ,
        mpl::push_back< mpl::_1 ,
          array_wrapper< double , mpl::_2 > >
      >
    >::type
>::type a_type; //fusion::vector< array<double,1> , array<double,2> , ... >

typedef array< double , s > b_type;
typedef array< double , s > c_type;
```

Instead of a dynamically allocated `double**` the compiler/optimizer sees fixed size arrays: `array<double,1>` , `array<double,2>`, ...

→ **better optimization possibilities**

The Generic Implementation

The actual Runge-Kutta step (details omitted):

```
fusion::for_each( stages ,  
                  calc_stage_caller( f , x , x_tmp , x_end , F , t , dt ) );
```

Remember: `stages` is `fusion::vector< stage<1> , stage<2> , ... >`
For each of the stages, `calc_stage` gets called, but the
`for_each`-loop is **executed by the compiler!**

The Generic Implementation

The actual Runge-Kutta step (details omitted):

```
fusion::for_each( stages ,  
                  calc_stage_caller( f , x , x_tmp , x_end , F , t , dt ) );
```

Remember: `stages` is `fusion::vector< stage<1> , stage<2> , ... >`

For each of the stages, `calc_stage` gets called, but the `for_each`-loop is **executed by the compiler!**

The compiler/optimizer sees sequential code:

```
calc_stage( stage_1 ); // stage_1 is an  
calc_stage( stage_2 ); // instance of stage<1>  
... // similar for stage_2 ...  
calc_stage( stage_s );
```

→ **better optimization possibilities**

The Generic Stepper

Provide some handy interface to the generic algorithm:

```
template< int s >
class generic_runge_kutta
{
public:
    generic_runge_kutta( const coef_a_type &a ,
                        const coef_b_type &b ,
                        const coef_c_type &c )
        : m_stages( a , b , c )
    { }

    void do_step( System f , const state_type &x , const double t ,
                  state_type &x_out , const double dt )
    {
        fusion::for_each( m_stages , calc_stage_caller( f , x , m_x_tmp , x_out ,
                                                         m_F , t , dt ) );
    }

private:
    stage_vector m_stages;
    state_type m_x_tmp;

protected:
    state_type m_F[s];
};
```

Example: Runge-Kutta 4

Butcher Tableau:

0				
0.5	0.5			
0.5	0	0.5		
1.0	0	0	1.0	
	1/6	1/3	1/3	1/6

```
// define the butcher array
const array< double , 1 > a1 = {{ 0.5 }};
const array< double , 2 > a2 = {{ 0.0 , 0.5 }};
const array< double , 3 > a3 = {{ 0.0 , 0.0 , 1.0 }};

const a_type a = fusion::make_vector( a1 , a2 , a3 );
const b_type b = {{ 1.0/6.0 , 1.0/3.0 , 1.0/3.0 , 1.0/6.0 }};
const c_type c = {{ 0.0 , 0.5 , 0.5 , 1.0 }};

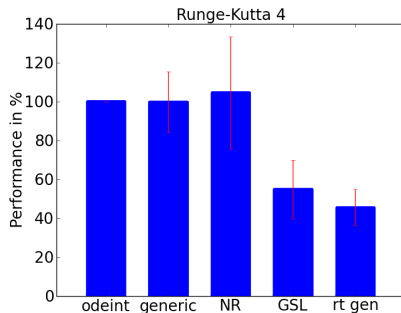
// create the stages with the rk4 parameters a,b,c
generic_runge_kutta< 4 > rk4( a , b , c );
// do one rk4 step
rk4.do_step( lorenz , x , 0.0 , x , 0.1 );
```

Performance

Did we achieve our aim? Test RK4 on Lorenz System!

Performance

Did we achieve our aim? Test RK4 on Lorenz System!

**Processors:**

Intel Core i7 830

Intel Core i7 930

Intel Xeon X5650

Intel Core2Quad Q9550

AMD Opteron 2224

AMD PhenomII X4 945

Compilers:

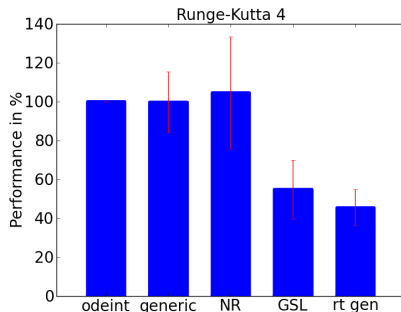
gcc 4.3 , 4.4 , 4.5 , 4.6

intel icc 11.1 , 12.0

msvc 9.0

Performance

Did we achieve our aim? Test RK4 on Lorenz System!



Yes!

Processors:

Intel Core i7 830

Intel Core i7 930

Intel Xeon X5650

Intel Core2Quad Q9550

AMD Opteron 2224

AMD PhenomII X4 945

Compilers:

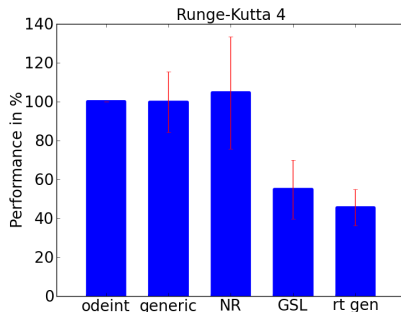
gcc 4.3 , 4.4 , 4.5 , 4.6

intel icc 11.1 , 12.0

msvc 9.0

Performance

Did we achieve our aim? Test RK4 on Lorenz System!



Processors:

Intel Core i7 830

Intel Core i7 930

Intel Xeon X5650

Intel Core2Quad Q9550

AMD Opteron 2224

AMD PhenomII X4 945

Compilers:

gcc 4.3 , 4.4 , 4.5 , 4.6

intel icc 11.1 , 12.0

msvc 9.0

Yes!

- On modern compilers (Intel 12, gcc 4.5/4.6) as fast as explicit code.
- Older compilers might produce slightly worse performant code.
- Always factor 2 better than run time generic implementation.

Conclusions

We implemented a generic Runge-Kutta algorithm that executes **any** RK scheme and has the following properties:

- Parameters (Butcher Tableau) can be defined in a natural way as C++ Arrays
- By virtue of Template Metaprogramming our code is as fast as direct implementation of the specific scheme
- **Major improvement (factor 2) compared to generic run time implementation** (but some increase in compile time)
- Embedded methods with error estimate can also be easily covered in a generic way
- This technique can be applied to other numerical problems, e.g. spline fitting, ...

