High-Performance Numerical Integration in the Age of C++26

VINCENT REVERDY

High-Performance Numerical Integration in the Age of C++26

Vincent Reverdy
Laboratoire d'Annecy de Physique des Particules, France
September 20th, 2024

Table of contents

Introduction

Starting with an example (in good old C++20)

What is this program doing?

```cpp
// Preamble
#include <array>
#include <cmath>
#include <string>
#include <vector>
#include <iostream>
#include <limits>
// Main function
int main(int argc, char* argv[]) {
    // Constants
    using type = double;
    constexpr type au = 149597870700.;
    constexpr type year = 365.25 * 24 * 60 * 60;
    constexpr type mpc = 1.E6 * (180 * 60 * 60 * au) / std::numbers::pi_v<type>;
    // Parameters
    const type h0 = (argc > 1) ? std::stod(argv[1]) : 0.677;
    const type hubble_0 = h0 * 100 * 1000 / mpc;
    const type omega_r0 = (argc > 2) ? std::stod(argv[2]) : (2.47E-5 * h0 * h0);
    const type omega_m0 = (argc > 3) ? std::stod(argv[3]) : 0.311;
    const type omega_l0 = (argc > 4) ? std::stod(argv[4]) : 0.689 - omega_r0;
    const type omega_k0 = 1 - (omega_r0 + omega_m0 + omega_l0);
    const type h =  (argc > 5) ? std::stod(argv[5]) : (1.E3 * year);
    const type a0 =  (argc > 6) ? std::stod(argv[6]) : (1.);
    // Computation
    auto evolution = compute(hubble_0, omega_r0, omega_m0, omega_l0, omega_k0, h, a0);
    // Output
    std::cout << std::abs(evolution.back()[0] / (1.E9 * year)) << std::endl;
    return 0;
}
```

Starting with an example (in good old C++20)

What is this program doing?

```cpp
// Computation
template <class T>
constexpr auto compute(T hubble_0, T omega_r0, T omega_m0, T omega_l0, T omega_k0, T h, T a0) {
    // Variables
    T t = 0;
    T a = a0;
    // Output
    std::vector<std::array<T, 2>> evolution;
    evolution.reserve((static_cast<T>(1) / hubble_0) / h);
    // Function to integrate
    auto da_dt = [&](auto t, auto a) {
        return a * hubble_0 * std::sqrt(
              omega_r0 * std::pow(a, static_cast<T>(-4))
            + omega_m0 * std::pow(a, static_cast<T>(-3))
            + omega_k0 * std::pow(a, static_cast<T>(-2))
            + omega_l0
        );
    };
    // Computation
    while (a > std::numeric_limits<T>::epsilon()) {
        evolution.emplace_back(std::array<T, 2>{{t, a}});
        a = a - h * da_dt(t, a);
        t = t - h;
    }
    // Output
    return evolution;
}
```

The output

$$13.7839$$

What's missing?

- A quantity and units library would be very (very) nice!

$$13.7839 \quad \text{billion} \quad \text{years}$$

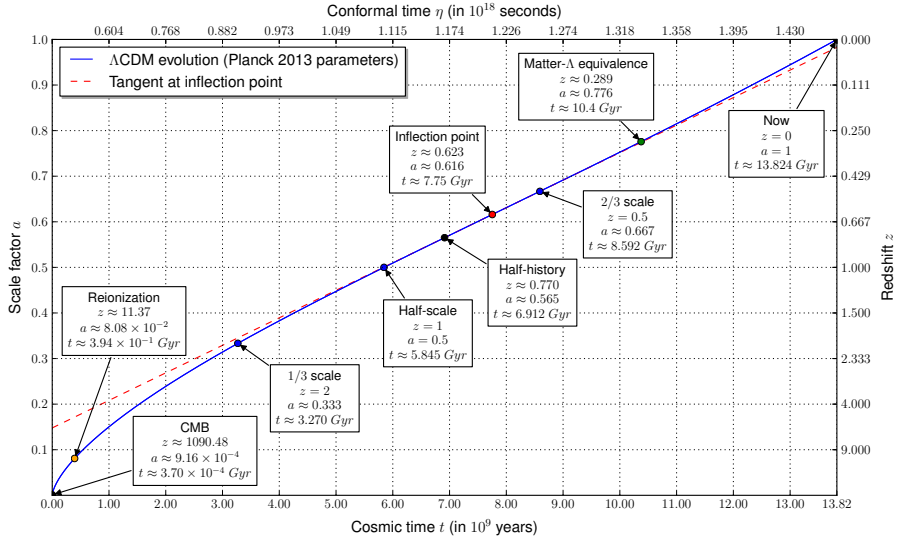
What did we just compute?

- The age of the Universe

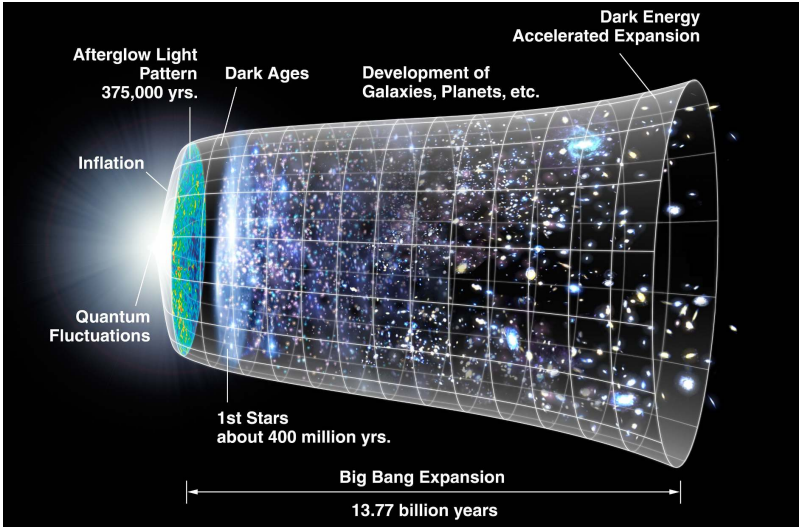
Conclusion

So we just computed the **Age of the Universe** in about **60 lines of C++20** !

Computing the evolution of the Universe



Our current understanding of the history of the Universe



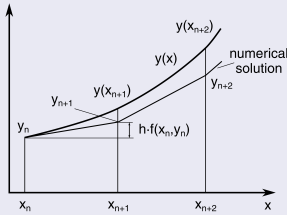
Remark

The solver

Coming back to the core of computation

```
1  // Function to integrate
2  auto da_dt = [&](auto t, auto a) {
3      return a * hubble_0 * std::sqrt(
4                 omega_r0 * std::pow(a, static_cast<T>(-4))
5             + omega_m0 * std::pow(a, static_cast<T>(-3))
6             + omega_k0 * std::pow(a, static_cast<T>(-2))
7             + omega_l0
8      );
9  };
10 // Computation
11 while (a > std::numeric_limits<T>::epsilon()) {
12     evolution.emplace_back(std::array<T, 2>{{t, a}});
13     a = a - h * da_dt(t, a);
14     t = t - h;
15 }
```

Euler's method



$$y_{n+1} = y_n + h \times f(t_n, y_n)$$

$$G_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu} \quad \Rightarrow \quad \begin{aligned} H^2 &= \left(\frac{\dot{a}}{a}\right)^2 = \frac{8\pi G}{3}\rho - \frac{kc^2}{a^2} \\ \dot{H} + H^2 &= \frac{\ddot{a}}{a} = -\frac{4\pi G}{3}\left(\rho + \frac{3P}{c^2}\right) \end{aligned} \quad \Rightarrow \quad \frac{H^2}{H_0^2} = \Omega_{R_0} a^{-4} + \Omega_{M_0} a^{-3} + \Omega_{k_0} a^{-2} + \Omega_{\Lambda_0}$$

Components

- Ω_R: radiation
- Ω_M: matter
- Ω_k: curvature
- Ω_Λ: dark-energy

Friedmann-Lemaitre equation that is being solved

$$\frac{da(t)}{dt} = a(t) H_0 \sqrt{\Omega_{R_0} a(t)^{-4} + \Omega_{M_0} a(t)^{-3} + \Omega_{k_0} a(t)^{-2} + \Omega_{\Lambda_0}}$$

Going beyond

End of story?

No!

Plenty of things can be improved

- This is the simplest possible integrator: can we do better and achieve more **accuracy** and **precision**?
- Can we **parallelize** the code?
- Can we use **more modern constructs**?
- Can we make it **more generic**?
- ...

Firsts steps

Toward an improved version

Integration function (1/4)

```cpp
// Integrate the Friedmann-Lemaitre(-Robertson-Walker) equations
template <
    typename T,
    class Container = std::vector<std::tuple<T, T>>
>
requires std::is_floating_point_v<T>
Container integrate_flrw(
    const T a_0,
    const T hubble_0,
    const T omega_r0,
    const T omega_m0,
    const T omega_l0,
    const T dt,
    const T dt_backup,
    const T t_backward,
    const T t_forward
)
{
    // Constants
    using value_type = T;
    using container_type = Container;
    constexpr value_type zero = 0;
    constexpr value_type one = 1;
    constexpr value_type two = 2;
    constexpr value_type three = 3;
    constexpr value_type six = 6;
    constexpr value_type w_r = one / three;
    constexpr value_type w_m = zero;
    constexpr value_type w_k = -one / three;
    constexpr value_type w_l = -one;
```

Toward an improved version

Integration function (2/4)

```
// Initialization
const value_type omega = omega_r0 + omega_m0 + omega_l0;
const value_type omega_k0 = one - omega;
container_type past(1, std::make_pair(zero, a_0));
container_type future(1, std::make_pair(zero, a_0));

// Expression of da/dt
auto da_dt = [&](const value_type a, const value_type sign) {
    return std::copysign(one, sign) * a * hubble_0 * std::sqrt(
        omega_r0 * std::pow(a, -three * (one + w_r))
        + omega_m0 * std::pow(a, -three * (one + w_m))
        + omega_k0 * std::pow(a, -three * (one + w_k))
        + omega_l0 * std::pow(a, -three * (one + w_l))
    );
};

// Runge-kutta of order 4
auto rk4 = [&](const value_type a, const value_type sign) {
    const value_type k1 = da_dt(a, sign);
    const value_type a1 = a + dt / two * k1;
    const value_type k2 = da_dt(a1, sign);
    const value_type a2 = a + dt / two * k2;
    const value_type k3 = da_dt(a2, sign);
    const value_type a3 = a + dt * k3;
    const value_type k4 = da_dt(a3, sign);
    return a + dt / six * (k1 + k2 + k2 + k3 + k3 + k4);
};
```

Toward an improved version

Integration function (3/4)

```
auto loop = [&](container_type& c, const value_type sign, const value_type t_lim) {
    // Initialization
    value_type t_sign = std::copysign(one, sign); value_type t_back = std::get<0>(c.back());
    value_type t_old = t_back; value_type t_new = t_back; bool t_ok = std::abs(t_new) < std::abs(t_lim);
    value_type a_sign = std::copysign(one, sign); value_type a_back = std::get<1>(c.back());
    value_type a_old = a_back; value_type a_new = a_back; bool a_ok = a_new > zero && std::isnormal(a_new);
    value_type origin = std::get<0>(c.back());
    c.reserve(std::abs((t_back - t_lim) / dt_backup));
    // Integration loop
    for (std::size_t i = 0; t_ok && a_ok; ++i) {
        t_new = origin + i * t_sign * dt;
        a_new = rk4(a_old, t_sign);
        a_ok = !(std::isnan(a_new) && a_old > a_0);
        if (!a_ok && !std::signbit(a_sign * t_sign)) {
            a_sign = -a_sign;
            a_new = rk4(a_old, a_sign);
        }
        t_ok = std::abs(t_new) < std::abs(t_lim);
        a_ok = a_new > zero && std::isnormal(a_new);
        if (t_ok && a_ok) {
            t_old = t_new;
            a_old = a_new;
            if (!(std::abs(t_back - t_new) < dt_backup)) {
                t_back = t_new;
                a_back = a_new;
                c.emplace_back(t_back, a_back);
            }
        } else if (std::abs(t_old - t_back) > zero) {c.emplace_back(t_old, a_old);}
    }
};
```

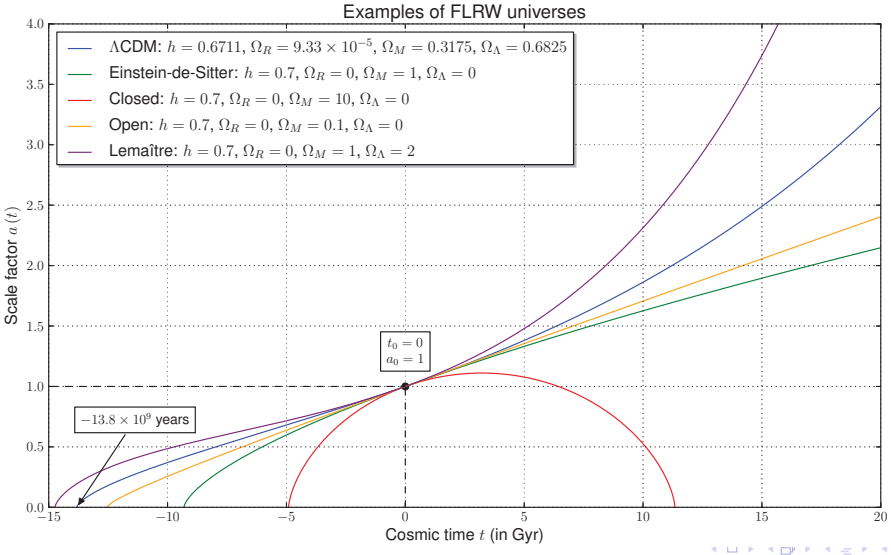
Toward an improved version

Integration function (4/4)

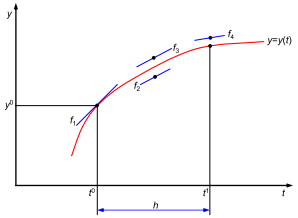
```
    // Execution of backward and forward integration
    std::thread backward(loop, std::ref(past), -one, t_backward);
    std::thread forward(loop, std::ref(future), one, t_forward);

    // Finalization
    backward.join();
    forward.join();
    future.insert(future.begin(), past.rbegin(), past.rend()-1);
    return future;
}
```

Toward an improved version



Runge-Kutta of order 4 (RK4)



Equations

$$
\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right) \\
k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right) \\
k_4 &= f(t_n + h, y_n + hk_3)
\end{aligned}
$$

$$
\begin{aligned}
y_{n+1} &= y_n + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\
t_{n+1} &= t_n + h
\end{aligned}
$$

Code

```cpp
// Runge-kutta of order 4
template <class T>
constexpr rk4(T t, T y, auto dy_dt, T dt) {
    // Step 1
    const T t1 = t;
    const T y1 = y;
    const T k1 = dy_dt(t1, y1);
    // Step 2
    const value_type t2 = t + dt / 2;
    const value_type y2 = y + dt / 2 * k1;
    const value_type k2 = dy_dt(t2, y2);
    // Step 3
    const value_type t3 = t + dt / 2;
    const value_type y3 = y + dt / 2 * k2;
    const value_type k3 = dy_dt(t3, y3);
    // Step 4
    const value_type t4 = t + dt;
    const value_type y4 = y + dt * k3;
    const value_type k4 = dy_dt(t3, y3);
    // Result
    return y + dt / 6 * (k1 + 2 * k2 + 2 * k3 + k4);
};
```

Sequentiality

- Every intermediate step k depends on the previous one!

Going multidimensionnal

Code

```
// Runge-kutta of order 4
template <class T, std::size_t N, class F>
requires std::floating_point<T> && std::invocable<F, T, std::array<T, N>>
constexpr std::array<T, N> rk4(T t, std::array<T, N> y, F dy_dt, T dt) {
    // Types and constants
    using value_type = T; using vector_type = std::array<T, N>;
    constexpr value_type two = 2; constexpr value_type six = 6;
    const value_type dt_2 = dt / two;
    // Variables
    value_type ti = t;
    vector_type yi;
    // Step 1
    ti = t;
    for (std::size_t j = 0; j < N; ++j) {yi[j] = y[j];}
    const vector_type k1 = dy_dt(ti, yi);
    // Step 2
    ti = t + dt_2;
    for (std::size_t j = 0; j < N; ++j) {yi[j] = y[j] + dt_2 * k1[j];}
    const vector_type k2 = dy_dt(ti, yi);
    // Step 3
    ti = t + dt_2;
    for (std::size_t j = 0; j < N; ++j) {yi[j] = y[j] + dt_2 * k2[j];}
    const vector_type k3 = dy_dt(ti, yi);
    // Step 4
    ti = t + dt;
    for (std::size_t j = 0; j < N; ++j) {yi[j] = y[j] + dt * k3[j];}
    const vector_type k4 = dy_dt(ti, yi);
    // Result
    for (std::size_t j = 0; j < N; ++j) {
        yi[j] = y[j] + dt / six * (k1[j] + two * k2[j] + two * k3[j] + k4[j]);
    }
    return yi;
};
```

Vector equation

$$Y(t_0) = Y_0$$
$$\frac{dY(t)}{dt} = f\left(t, Y\right)$$

Three-body problem equations

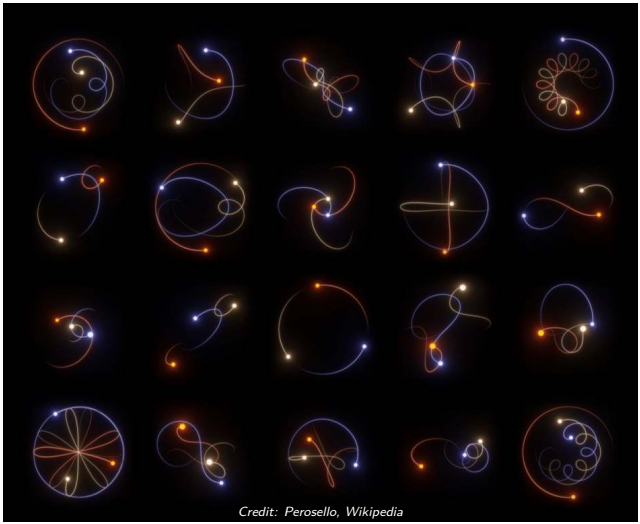
Code

```cpp
// Three body problem
template <std::floating_point T>
constexpr std::array<T, 18> dy_dt(
    T t, std::array<T, 18> y,
    T G, T m1, T m2, T m3
) {
    constexpr std::size_t ndim = 3;
    std::array<T, 18> output = {};
    std::ranges::subrange x(std::begin(y) + 0, 9);
    std::ranges::subrange x1(std::begin(y) + 0, 3);
    std::ranges::subrange x2(std::begin(y) + 3, 3);
    std::ranges::subrange x3(std::begin(y) + 6, 3);
    const T d12 = std::hypot(x1[0] - x2[0], x1[1] - x2[1], x1[2] - x2[2]);
    const T d23 = std::hypot(x2[0] - x3[0], x2[1] - x3[1], x2[2] - x3[2]);
    const T d31 = std::hypot(x3[0] - x1[0], x3[1] - x1[1], x3[2] - x1[2]);
    std::ranges::subrange v(std::begin(y) + 9, 18);
    std::array<T, 3> a1 = {};
    std::array<T, 3> a2 = {};
    std::array<T, 3> a3 = {};
    for (std::size_t i = 0; i < ndim; ++i) {
        a1[i] = -G * m2 * (x1[i] - x2[i]) / std::pow(d12, 3)
                -G * m3 * (x1[i] - x3[i]) / std::pow(d31, 3);
        a2[i] = -G * m3 * (x2[i] - x3[i]) / std::pow(d23, 3)
                -G * m1 * (x2[i] - x1[i]) / std::pow(d12, 3);
        a3[i] = -G * m1 * (x3[i] - x1[i]) / std::pow(d31, 3)
                -G * m1 * (x3[i] - x2[i]) / std::pow(d23, 3);
    }
    std::copy(v, std::begin(output) + 0);
    std::copy(a1, std::begin(output) + 9);
    std::copy(a2, std::begin(output) + 12);
    std::copy(a3, std::begin(output) + 15);
    return output;
}
```

Vector equation

$$Y = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ x_2 \\ y_2 \\ z_2 \\ x_3 \\ y_3 \\ z_3 \\ v_{x1} \\ v_{y1} \\ v_{z1} \\ v_{x2} \\ v_{y2} \\ v_{z2} \\ v_{x3} \\ v_{y3} \\ v_{z3} \end{pmatrix} \qquad \frac{dY}{dt} = \begin{pmatrix} v_{x1} \\ v_{y1} \\ v_{z1} \\ v_{x2} \\ v_{y2} \\ v_{z2} \\ v_{x3} \\ v_{y3} \\ v_{z3} \\ a_{x1} \\ a_{y1} \\ a_{z1} \\ a_{x2} \\ a_{y2} \\ a_{z2} \\ a_{x3} \\ a_{y3} \\ a_{z3} \end{pmatrix}$$

Three-body problem visualizations



Credit: Perosello, Wikipedia

Problems

To achieve long-term correct integration, far better integrators are needed.

Going beyond

After this gentle introduction, we will dive into the details!

Context

An ambiguity

Integrals

$$\int_a^b f\left(x\right) dx$$

- *Numerical integration* as in the computation of the numerical value of a **definite integral**
- Synonyms: **quadrature**, sometimes cubature in higher-dimensions ($\int\int$, $\int\int\int$, ...)

Diffential equations

$$\frac{dy}{dt} = f\left(t, y\left(t\right)\right) \qquad y\left(t_0\right) = y_0$$

$$f : [t_0, +\infty) \times \mathbb{R}^d \to \mathbb{R}^d \qquad y_0 \in \mathbb{R}^d$$

- *Numerical integration* as in numerical methods for **Ordinary Differential Equations** (ODEs)
- Synonym: **diffential equation solver**
- The term integration can go beyond ODEs, and include many other types of differential equations

Definite integrals and differential equations

An equivalence

$$\int f(x)\,dx \quad \Leftrightarrow \quad \frac{dy}{dx} = f(x)$$

- Finding antiderivatives $\Leftrightarrow$ solving an ODE

Algorithms for definite integrals

- Gaussian quadrature
- Clenshaw-Curtis quadrature
- Romberg's method
- ...

Algorithms for ordinary differential equations

- Euler's method
- Runge-Kutta methods (RK)
- General Linear Methods (GLM)
- ...

This talk (because we only have 1h!)

- Focus on **Ordinary Diffential Equations**
- *Integrators* mean **diffential equations solvers**

The lands of differential equations

Differential equation solvers

- In reality go way beyond this map
- Including Partial Differential Equations (PDEs)

This talk

- Focus on good old classical **Ordinary Diffential Equations**

Going beyond

- Many techniques presented here can be extended to other types of solvers

In programming languages

Python

- `scipy.integrate`

Julia

- `DifferentialEquations.jl`

C++

- `Boost.Numeric.Odeint`

Comparison with C++

- `Boost.Numeric.Odeint`: Copyright 2009-2015 Karsten Ahnert and Mario Mulansky
- C++ is stuck in the past, other languages do far better in terms of everything: functionality, ease of use, and even performance

This talk

- The goal **is NOT** to revolutionize everything or show a library that beats everything
- The goal **is** to start from the foundation and illustrate a path forward with basic examples

Theoretical foundations

Problem statement: Initial Value Problem (IVP)

Ordinary Diffential Equation (ODE)

$$\frac{dy(t)}{dt} = f(t, y(t))$$
$$f : [t_0, +\infty) \times \mathbb{R}^d \to \mathbb{R}^d$$

Initial condition

$$y(t_0) = y_0$$
$$y_0 \in \mathbb{R}^d$$

Mathematical and software architecture

Question

- How to abstract the problem?

Mathematical abstractions

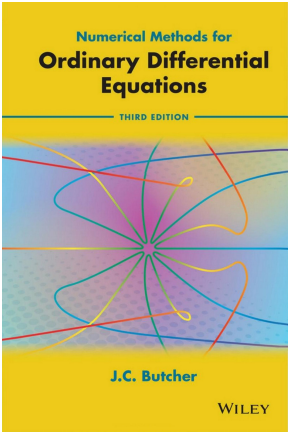
- Well-posed and abstracted in mathematics ⇒ "easy" to abstract in software
- Mathematical architecture ⇒ Software architecture

Counter-example: units

- Intuitively understood in physics
- But very fuzzy foundations in mathematics
- ⇒ Incredibly hard to abstract correctly in software

The mathematical architecture of algorithms

John C. Butcher



Numerical Methods for Ordinary Differential Equations,Third Edition
John C. Butcher
John Wiley & Sons, 2016

Main classes of algorithms

Runge-Kutta Methods (RK)

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i \qquad k_i = f(t_n + c_i h, y_n + (a_{i1} k_1 + a_{i2} k_2 + \cdots + a_{i,i-1} k_{i-1}) h)$$

Linear Multistep Methods (LLM)

$$y_{n+s} + a_{s-1} \cdot y_{n+s-1} + a_{s-2} \cdot y_{n+s-2} + \cdots + a_0 \cdot y_n = h \cdot \Big(b_s \cdot f(t_{n+s}, y_{n+s}) + \cdots + b_0 \cdot f(t_n, y_n) \Big)$$

$$\Leftrightarrow \sum_{j=0}^{s} a_j y_{n+j} = h \sum_{j=0}^{s} b_j f(t_{n+j}, y_{n+j})$$

General Linear Methods (GLM)

$$y^{[n-1]} = \begin{bmatrix} y_1^{[n-1]} \\ y_2^{[n-1]} \\ \vdots \\ y_r^{[n-1]} \end{bmatrix}, \quad y^{[n]} = \begin{bmatrix} y_1^{[n]} \\ y_2^{[n]} \\ \vdots \\ y_r^{[n]} \end{bmatrix}, \quad Y = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_s \end{bmatrix}, \quad F = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_s \end{bmatrix} = \begin{bmatrix} f(Y_1) \\ f(Y_2) \\ \vdots \\ f(Y_s) \end{bmatrix}$$

$$Y_i = \sum_{j=1}^{s} a_{ij} h F_j + \sum_{j=1}^{r} u_{ij} y_j^{[n-1]}, \quad i = 1, 2, \ldots, s \qquad y_i^{[n]} = \sum_{j=1}^{s} b_{ij} h F_j + \sum_{j=1}^{r} v_{ij} y_j^{[n-1]}, \quad i = 1, 2, \ldots, r$$

Different approaches

Runge-Kutta Methods (RK)

- May use intermediary steps
- But discard the information
- Only need

Linear Multistep Methods (LMM)

- Take advantages of previous steps to improve efficiency

General Linear Methods (GLM)

- Can combine advantages of both approaches

Explicit methods

- Solution at the next time step is computed directly using only the information from previous steps
- **Less robust for stiff problems**

Implicit methods

- Solution implicitly defined through an equation that includes the unknown future value
- A system of equations need to be solved at each time step
- **Better stability for stiff problems**

Mathematical abstraction

Butcher tableau for Runge-Kutta methods (explicit)

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array} = \begin{array}{c|cccccc} c_1 & a_{1,1} & 0 & 0 & \cdots & 0 & 0 \\ c_2 & a_{2,1} & a_{2,2} & 0 & \cdots & 0 & 0 \\ c_3 & a_{3,1} & a_{3,2} & a_{3,3} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{s-1} & a_{s-1,1} & a_{s-1,2} & a_{s-1,3} & \cdots & a_{s-1,s-1} & 0 \\ c_s & a_{s,1} & a_{s,2} & a_{s,3} & \cdots & a_{s,s-1} & a_{s,s} \\ \hline & b_1 & b_2 & b_3 & \cdots & b_{s-1} & b_s \end{array}$$

- $a_{i,j}$: **coefficients** (A: Runge-Kutta matrix)
- b_j: **weights**
- c_i: **nodes**
- s: number of **stages**

Design of methods

- The coefficients, weights, nodes have to be carefully derived and will determine the properties of the corresponding method

Adaptive Runge-Kutta methods

Butcher tableau for explicit adaptive Runge-Kutta methods

$$
\begin{array}{c|c}
c & A \\
\hline
 & b^T \\
 & \tilde{b}^T
\end{array}
=
\begin{array}{c|cccccc}
c_1 & a_{1,1} & 0 & 0 & \cdots & 0 & 0 \\
c_2 & a_{2,1} & a_{2,2} & 0 & \cdots & 0 & 0 \\
c_3 & a_{3,1} & a_{3,2} & a_{3,3} & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
c_{s-1} & a_{s-1,1} & a_{s-1,2} & a_{s-1,3} & \cdots & a_{s-1,s-1} & 0 \\
c_s & a_{s,1} & a_{s,2} & a_{s,3} & \cdots & a_{s,s-1} & a_{s,s} \\
\hline
 & b_1 & b_2 & b_3 & \cdots & b_{s-1} & b_s \\
 & \tilde{b}_1 & \tilde{b}_2 & \tilde{b}_3 & \cdots & \tilde{b}_{s-1} & \tilde{b}_s
\end{array}
$$

$$\tilde{y}_{n+1} = y_n + h \sum_{i=1}^{s} \tilde{b}_i k_i$$

$$e_{n+1} = \tilde{y}_{n+1} - y_{n+1} = h \sum_{i=1}^{s} (\tilde{b}_i - b_i) k_i \qquad \text{(error estimate)}$$

Implicit Runge-Kutta methods

Butcher tableau for implicit adaptive Runge-Kutta methods

$$
\begin{array}{c|c}
c & A \\
\hline
 & b^T \\
 & \tilde{b}^T
\end{array}
=
\begin{array}{c|cccccc}
c_1 & a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,s-1} & a_{1,s} \\
c_2 & a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,s-1} & a_{2,s} \\
c_3 & a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,s-1} & a_{3,s} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
c_{s-1} & a_{s-1,1} & a_{s-1,2} & a_{s-1,3} & \cdots & a_{s-1,s-1} & a_{s-1,s} \\
c_s & a_{s,1} & a_{s,2} & a_{s,3} & \cdots & a_{s,s-1} & a_{s,s} \\
\hline
 & b_1 & b_2 & b_3 & \cdots & b_{s-1} & b_s \\
 & \tilde{b}_1 & \tilde{b}_2 & \tilde{b}_3 & \cdots & \tilde{b}_{s-1} & \tilde{b}_s
\end{array}
$$

$$
y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i
$$

$$
k_i = f\left(t_n + c_i h,\ y_n + h \sum_{j=1}^{s} a_{ij} k_j \right), \quad i = 1, \ldots, s
$$

Linear Multistep Methods (LMM)

Specification

$$y_n = \alpha_1 y_{n-1} + \alpha_2 y_{n-2} + \cdots + \alpha_k y_{n-k} + \beta_0 h f\left(x_n, y_n\right) + \beta_1 h f\left(x_{n-1}, y_{n-1}\right) + \cdots + \beta_k h f\left(x_{n-k}, y_{n-k}\right)$$

The method is specified by $[\alpha, \beta]$ where:

$$\alpha(z) = 1 - \alpha_1 z - \alpha_2 z^2 - \cdots - \alpha_k z^k$$
$$\beta(z) = \beta_0 + \beta_1 z + \beta_2 z^2 + \cdots + \beta_k z^k$$

General Linear Methods (GLM)

General abstraction

$$\begin{bmatrix} A & U \\ B & V \end{bmatrix} \quad \text{with} \quad \begin{array}{l} A \text{ of size } s \times s \\ B \text{ of size } r \times s \\ U \text{ of size } s \times r \\ V \text{ of size } r \times r \end{array}$$

$$Y_i = \sum_{j=1}^{s} a_{ij} h F_j + \sum_{j=1}^{r} u_{ij} y_j^{[n-1]}, \qquad i = 1, 2, \ldots, s$$

$$y_i^{[n]} = \sum_{j=1}^{s} b_{ij} h F_j + \sum_{j=1}^{r} v_{ij} y_j^{[n-1]}, \qquad i = 1, 2, \ldots, r$$

$$\Rightarrow \qquad \begin{bmatrix} Y \\ y^{[n]} \end{bmatrix} = \begin{bmatrix} A & U \\ B & V \end{bmatrix} \begin{bmatrix} hF \\ y^{[n-1]} \end{bmatrix}$$

Relationships between RKs, LMMs, and GLMs

- Runge-Kutta and Linear Multisteps methods can be represented through $\begin{bmatrix} A & U \\ B & V \end{bmatrix}$
- They can be seen as special cases of General Linear Methods

Special cases of the general framework

Runge-Kutta

- Euler
- Heun's method
- Runge-Kutta-Fehlberg method
- Cash-Karp method
- Dormand-Prince method
- Runge-Kutta-Nystrom methods
- Crank-Nicolson method
- Gauss-Legendre methods
- Lobatto methods
- Radau methods
- ...

Linear Multistep Methods

- Backward Differentiation Formulas
- Adams-Bashforth methods
- Adams-Moulton methods
- ...

General Linear Methods

- Diagonally Implicit Multistage Integration Methods
- Almost Runge-Kutta methods
- G-symplectic methods
- ...

Strategy

- Use the General Linear Methods matrix A, U, B, V as a foundation for the software architecture

Outline of an implementation

Implementation strategy

This talk

- This talk: we will focus on Runge-Kutta subcase using Butcher's tableau a_{ij}, b_j, c_i
- Generalizing to GLM would use the same approach

Reminder

$$
\begin{array}{c|c}
c & A \\
\hline
 & b^T \\
 & \tilde{b}^T
\end{array}
=
\begin{array}{c|cccccc}
c_1 & a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,s-1} & a_{1,s} \\
c_2 & a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,s-1} & a_{2,s} \\
c_3 & a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,s-1} & a_{3,s} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
c_{s-1} & a_{s-1,1} & a_{s-1,2} & a_{s-1,3} & \cdots & a_{s-1,s-1} & a_{s-1,s} \\
c_s & a_{s,1} & a_{s,2} & a_{s,3} & \cdots & a_{s,s-1} & a_{s,s} \\
\hline
 & b_1 & b_2 & b_3 & \cdots & b_{s-1} & b_s \\
 & \tilde{b}_1 & \tilde{b}_2 & \tilde{b}_3 & \cdots & \tilde{b}_{s-1} & \tilde{b}_s
\end{array}
$$

$$
y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i \qquad k_i = f\left(t_n + c_i h,\ y_n + h \sum_{j=1}^{s} a_{ij} k_j \right), \quad i = 1, \ldots, s
$$

Target

This talk

- We'll have Verner's most efficient Runge-Kutta pairs in mind: https://www.sfu.ca/ jverner/

Adaptive Runge-Kutta pairs with interpolants

- Runge-Kutta (7)6 Pair with Interpolants (July 2024)
- Runge-Kutta (8)7 Pair with Interpolants (May 2024)
- Runge-Kutta (9)8 Pair with Interpolants (April 2024)

Verner's methods

- Explicit adaptive Runge-Kutta schemes
- Slightly enhanced methods with interpolants
- High-accurary interpolation inside integration steps

Interpolation with extra coefficients β_i

$$y_{n+u} = y_n + h \sum_{i=1}^{s^*} \beta_i\,(u)\,F_i \quad \text{at} \quad x_n + uh,\ u \in [0,1]$$

See *Numerically optimal Runge-Kutta pairs with interpolants, J.H. Verner, Springer Science, 2009*

Verner's coefficients examples

```
##############################################################################
#
#     A better `efficient' Runge--Kutta (16:8,9) pair
#
#     These are approximate REAL coefficients computed
#        using MAPLE with 40 digits for
#
#     a slightly more efficient SIXTEEN-stage conventional
#        pair of methods of orders  p=8  and  p=9, with
#             dominant stage order = 5,
#
#     together with approximate 40-digit coefficients
#        for two interpolants of orders 8 and 9 that
#        require 5 and 5 extra stages respectively.
#
#     Exact coefficients of the method are possible using
#  surds in terms of sqrt(6). Exact coefficients for the
#  interpolants require that c[18] must be a zero of a
#  cubic polynomial with coefficients that are surds in
#  terms of sqrt(6). Hence, exact coefficients for the
#  interpolants cannot be conveniently represented.
#
#     This procedure is a better efficient pair in the
#       sense that for an unrestricted maximum coefficient
#       from b  and  A, it has a propagating formula with
#       a minimum 2-norm of the local truncation error as

#           T_10,2 ~ .00000034399

#       This 2-norm is slightly smaller than T10,2 for the
#       (16:8,9) pair tabulated above. This 2-norm has two
#  local maximum values on  [0,1].
#
#     Additional stages and interpolating weights allow
#       for the computation of an approximation of order up
#  to order  p, at any point of the domain of solution.
#  These interpolants have continuous derivatives.
#
#     Nodes  c[17]=1, c[18] - c[21] were selected to
#       minimize the maximum of the 2-norm of the local
#       truncation error over the interval  [0,1] for the
#       interpolant of order 8, and this value is
#
#            Ti_9,2 ~ .000003721
#
#     This 2-norm has three local maximum values on  [0,1].
#
#     The remaining five nodes were selected in an attempt
#       to minimize the maximum of the 2-norm of the local
#       truncation error on the interval  [0,1]  for the
#       interpolant of order 9, and this value is
#
#             Ti_10,2 ~ .0000007830
#
#     This 2-norm has two local maximum values on  [0,1].
#
#     The formulas scanned for this optimal formula are
#       those developed in J.H. Verner, SIAM J NA 1978, 772-790,
#       "Explicit Runge--Kutta methods with estimates of the
#       Local Truncation Error". It is conceivable that the
#       pairs in J.H. Verner, Applied Numerical Mathematics 22
#       1996, 345--357, "High-order explicit Runge--Kutta pairs
#       with low stage order", which also require 16 stages,
#       but solve the order conditions in a different way
#       may yield particular pairs of equivalant or more
#       efficiency.  Coefficients for the interpolants are
#       those developed in J.H. Verner, SIAM J NA 1993,
#       1446-1466, "Differentiable Interpolants for high order
#       Runge--Kutta methods".
#
#     Instructions for using the interpolants are contained
#  in J.H. Verner, SIAM NA 30, 1993, 1446-1466, "Differentiable
#  Interpolants for high-order Runge--Kutta methods".
#
##############################################################################
```

```
#  NODES
#  -----
c[1] = 0
c[2] = .3571e-1
c[3] = .9906028091267414072062290548990445721420e-1
c[4] = .1485904213690112110809343582348566858213
c[5] = .6134
c[6] = .2327359473605626756631680289030288192566
c[7] = .5538640526394373243368319710969711807434
c[8] = .6555
c[9] = .491625
c[10] = .6858e-1
c[11] = .253
c[12] = .6620641795412045944786226689660879617864
c[13] = .8309
c[14] = .8998
c[15] = 1
c[16] = 1
#
# ************************************************************
#  COUPLING COEFFICIENTS
#  ---------------------
# for c[1] = 0
#
# for c[2] = .3571e-1
a[2,1] = .3571e-1
#
# for c[3] = .9906028091267414072062290548990445721420e-1
a[3,1] = -.3833735636677017025757228807792426014327e-1
a[3,2] = .1373976372794443109781951935678287173575
#
# for c[4] = .1485904213690112110809343582348566858213
a[4,1] = .3714760534225280277023358955871417145532e-1
a[4,2] = 0
a[4,3] = .1114428160267584083107007686761425143660
#
# for c[5] = .6134
a[5,1] = 2.674764429871505119194043347886848033362
a[5,2] = 0
a[5,3] = -9.982382134885293836441318015951354323506
a[5,4] = 7.921017705013788717247274668064506290145
#
# for c[6] = .2327359473605626756631680289030288192566
a[6,1] = .5242104050577351069841401193318449756747e-1
a[6,2] = 0
a[6,3] = 0
a[6,4] = .1796911189175953081279466649977364298948
a[6,5] = .6237879371938568368073519721078917943307e-3
#
# for c[7] = .5538640526394373243368319710969711807434
a[7,1] = .1592492223647632083060991150129732726703
a[7,2] = 0
a[7,3] = 0
a[7,4] = -.4298429877241087508189185066909803751595
a[7,5] = .6665266542726088051243445942394404211672e-1
a[7,6] = .7578051525715219863372169033510342411158
#
# for c[8] = .6555
a[8,1] = .7283333333333333333333333333333333333333e-1
a[8,2] = 0
a[8,3] = 0
a[8,4] = 0
a[8,5] = 0
a[8,6] = .3359344590665103678713422141936031057620
a[8,7] = .2467322076001562987953244524730635609046
#
# for c[9] = .491625
a[9,1] = .729755859375e-1
a[9,2] = 0
a[9,3] = 0
a[9,4] = 0
a[9,5] = 0
a[9,6] = .3348009729699333533129043555243262838993
a[9,7] = .1184158239050666466870956444756737161007
a[9,8] = -.345673828125e-1
#
```

```
# for c[10] = .6858e-1
a[10,1] = .4911213663452096382929799914888692571185e-1
a[10,2] = 0
a[10,3] = 0
a[10,4] = 0
a[10,5] = 0
a[10,6] = .3983857361308652347066808423646954244674e-1
a[10,7] = .1069675288939354812077139476183615284408
a[10,8] = -.2174259165458647598455650055976622663086e-1
a[10,9] = -.1055956474869564925231235304439517699685
#
# for c[11] = .253
a[11,1] = -.2707988818641280489175862864774930636734e-1
a[11,2] = 0
a[11,3] = 0
a[11,4] = 0
a[11,5] = 0
a[11,6] = .333e-1
a[11,7] = -.1645526070036057075627040519129748090051
a[11,8] = .3428266306497389946775103799535438271025e-1
a[11,9] = .1585264064439221046855989221358760453749
a[11,10] = .2185234256811225083011127204294936872873
#
# for c[12] = .6620641795412045944786226689660879617864
a[12,1] = .5584657769108862727526281006220448941548e-1
a[12,2] = 0
a[12,3] = 0
a[12,4] = 0
a[12,5] = 0
a[12,6] = .9166533166672539087479545790553151183505e-1
a[12,7] = .2392399655523627049569179599100121305027
a[12,8] = .1023834712248414879740316672186561113729e-1
a[12,9] = -.2679331322859542442991625976085644567159e-2
a[12,10] = .4235624181474284646680744174197106205439e-1
a[12,11] = .2253970470166604185504274586005888014087
#
# for c[13] = .8309
a[13,1] = -.4802510512725195893103945790000307739284
a[13,2] = 0
a[13,3] = 0
a[13,4] = 0
a[13,5] = 0
a[13,6] = -6.359610162555930097843606206238737368598
a[13,7] = -.2762313898040841385163666588363199062208
a[13,8] = -6.500796633979846747011407331804389650124
a[13,9] = .5734765877040956875980719881692630407295
a[13,10] = 1.347125994868138849716719646646049357523
a[13,11] = 5.936840409706221306933249144503479736029
a[13,12] = 6.590346245333924728433733996560685564589
#
# for c[14] = .8998
a[14,1] = .3307533067671401079975082476263855100019
a[14,2] = 0
a[14,3] = 0
a[14,4] = 0
a[14,5] = 0
a[14,6] = 5.956207776829962111958631943398154563177
a[14,7] = -.4868316400481527952968983900652017328476
a[14,8] = 4.462055288206771191616083778225319083232
a[14,9] = .7410258231442071778525406966672954745201
a[14,10] = -.7118192034575913119318346652024763899638
a[14,11] = -5.454619594516665440649148499095762263275
a[14,12] = -4.140803729244709693305868672684077643030
a[14,13] = .2038319723190386517589855611303633981843
#
# for c[15] = 1
a[15,1] = -.5847111122998944389651719860007729221315
a[15,2] = 0
a[15,3] = 0
a[15,4] = 0
a[15,5] = 0
a[15,6] = -12.41268417116267068926060969930886386670
a[15,7] = 1.360245445660928146977003413153919520189
a[15,8] = -22.42610531111868229005965344123311160860
a[15,9] = -.8828857055865458252669099251378431470909
a[15,10] = 1.770155128538230466813900572142546329490
```

Example: A better efficient Runge–Kutta (16:8,9) pair on Verner's website

Architecture

A Runge-Kutta method can be abstracted by:

- a_{ij}: Runge-Kutta coefficients $\Rightarrow$ 2D array
- b_j^n: weights $\Rightarrow$ a sequence of 1D arrays (1 for non-adaptive, generally 2 for adaptive)
- c_i: nodes $\Rightarrow$ a sequence of 1D array
- β_j^n: interpolants: $\Rightarrow$ a sequence of 1D arrays (one for each b^n)

General idea

```
using rk = make_runge_kutta<
    runge_kutta_coefficients,
    runge_kutta_weights,
    runge_kutta_nodes,
    runge_kutta_interpolants,
    ...
>;
```

Approach

Recursive generation

We want to generate the following automatically from the coefficients:

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i$$

$$k_i = f\left(t_n + c_i h,\ y_n + h \sum_{j=1}^{s} a_{ij} k_j\right), \quad i = 1, \ldots, s$$

Performance concerns

- The Butcher Tableau can be very sparse
- Null coefficients should be optimized away
- Compilers may not be able to do it because of floating-point special values

Meta-programming helpers

What we want to be able to type

```
using runge_kutta_coefficients = make_coefficient_array<
    indexer[0, 0] = coefficient<double>(/*value*/),
    indexer[1, 0] = coefficient<double>(/*value*/),
    indexer[1, 1] = coefficient<double>(/*value*/),
    indexer[2, 0] = coefficient<double>(/*value*/),
    //...
>;
```

Indexing strategy

```
// Generic case to build an AST
template <class... Ts>
struct indexer_type {
    // Constructor
    constexpr indexer_type(Ts... i): indices(i...) {}
    // Build an AST
    template <class T>
    constexpr indexer_expression<index_type, coefficient<double>> operator=(coefficient<double> c) {
        return indexer_expression(*this, c);
    }
    // Indices
    const std::tuple<Ts...> indices;
};
// Specialization
template <>
struct indexer_type<void> {
    template <class... Args>
    constexpr indexer_type<Args...> operator[](Args... args) const noexcept {return indexer_type(args...);};
};
// Builder to be able to type indexer[i, j]
inline constexpr indexer_type<void> indexer;
```

Meta-programming helpers

Transformation

```
using runge_kutta_coefficients = make_coefficient_array<
    indexer[0, 0] = coefficient<double>(/*value*/),
    indexer[1, 0] = coefficient<double>(/*value*/),
    indexer[1, 1] = coefficient<double>(/*value*/),
    indexer[2, 0] = coefficient<double>(/*value*/),
    //...
>;
```

will compile-into:

```
using runge_kutta_coefficients = make_coefficient_array<
    indexer_expression<indexer<std::size_t, std::size_t>, coefficient<double>>,
    indexer_expression<indexer<std::size_t, std::size_t>, coefficient<double>>,
    indexer_expression<indexer<std::size_t, std::size_t>, coefficient<double>>,
    indexer_expression<indexer<std::size_t, std::size_t>, coefficient<double>>,
    //...
>;
```

Details

See the talk of last year for the implementation of this kind of expression templates:
Symbolic Calculus for High-Performance Computing from Scratch using C++23, V. Reverdy, CppCon2023

Sorting

Array of expressions

```
using runge_kutta_coefficients = make_coefficient_array<
    indexer_expression<indexer<std::size_t, std::size_t>, coefficient<double>>,
    indexer_expression<indexer<std::size_t, std::size_t>, coefficient<double>>,
    indexer_expression<indexer<std::size_t, std::size_t>, coefficient<double>>,
    //...
>;
```

Unique order

```
using runge_kutta_coefficients = make_coefficient_array<
    indexer[0, 0] = coefficient<double>(/*value*/),
    indexer[1, 0] = coefficient<double>(/*value*/),
    //...
>;
```

and

```
using runge_kutta_coefficients = make_coefficient_array<
    indexer[1, 0] = coefficient<double>(/*value*/),
    indexer[0, 0] = coefficient<double>(/*value*/),
    //...
>;
```

need to collapse to the same type!

C++26 proposal that will help

- Language constructs to uniquely order types
- See P2830R4: Standardized Constexpr Type Ordering, G. Azman & N. Nichols

Indexing strategy

Strategy outline

- `indexer_expression` will generate an `indexed_coefficient`
- `indexed_coefficient` will have an access operator that will return a `numeric_constant`
- `numeric_constant` will hijack the compiler's arithmetic

Indexed coefficient outline

```
template <class Index, class Coefficient>
struct indexed_coefficient {
    using index_type = Index;
    using coefficient_type = Coefficient;
    using value_type = coefficient_type::value_type;
    static constexpr index_type index = {};
    static constexpr coefficient_type coefficient = {};
    constexpr value_type operator()(index_type) const noexcept {
        return coefficient_type::value;
    }
    constexpr coefficient_type operator[](index_type) const noexcept {
        return {};
    }
};
```

Accessing via overload resolution

`indexed_coefficient_sequence[index<1, 2>]` will redirect with the correct `indexed_coefficient`

Highjacking arithmetic (1/3)

Numeric constant

```
template <class T, T v>
requires std::is_arithmetic_v<T>
struct numeric_constant {
    using value_type = T;
    using type = numeric_constant<T, v>;
    static constexpr T value = v;
    constexpr operator value_type() const noexcept {
        return v;
    }
    template <class U>
    requires std::is_arithmetic_v<U>
    explicit constexpr operator U() const noexcept {
        return v;
    }
    constexpr value_type operator()() const noexcept {
        return v;
    }
};

template <class T, auto v>
requires std::is_arithmetic_v<T> && std::is_arithmetic_v<decltype(v)>
constexpr numeric_constant<T, static_cast<T>(v)> make_numeric_constant(
) noexcept {
    return {};
}
```

Highjacking arithmetic (2/3)

Null constant

```
template <class T>
using null_constant = numeric_constant<T, static_cast<T>(0)>;

template <class T>
struct is_null_constant: std::false_type {};

template <class T, T v>
struct is_null_constant<numeric_constant<T, v>>:
std::bool_constant<v == static_cast<T>(0)> {};

template <std::floating_point T, T v>
struct is_null_constant<numeric_constant<T, v>>:
std::bool_constant<
    std::isfinite(v) &&
    !(std::isless(v, static_cast<T>(0)) || std::isgreater(v, static_cast<T>(0)))
> {};

template <class T>
constexpr bool is_null_constant_v = is_null_constant<T>::value;

template <class T>
requires std::is_arithmetic_v<T>
constexpr null_constant<T> make_null_constant() noexcept {
    return {};
}
```

Highjacking arithmetic (3/3)

Operators

```
template <class T1, T1 v1, class T2, T2 v2>
requires (!is_null_constant_v<numeric_constant<T1, v1>>) && (!is_null_constant_v<numeric_constant<T2, v2>>)
constexpr numeric_constant<std::common_type_t<T1, T2>, v1 + v2>
operator+(numeric_constant<T1, v1>, numeric_constant<T2, v2>) noexcept {return {};}

template <class T1, T1 v1, class T2>
requires (!is_null_constant_v<numeric_constant<T1, v1>>) && is_null_constant_v<null_constant<T2>>
constexpr numeric_constant<std::common_type_t<T1, T2>, v1>
operator+(numeric_constant<T1, v1>, null_constant<T2>) noexcept {return {};}

template <class T1, class T2, T2 v2>
requires is_null_constant_v<null_constant<T1>> && (!is_null_constant_v<numeric_constant<T2, v2>>)
constexpr numeric_constant<std::common_type_t<T1, T2>, v2>
operator+(null_constant<T1>, numeric_constant<T2, v2>) noexcept {return {};}

template <class T1, class T2>
requires is_null_constant_v<null_constant<T1>> && is_null_constant_v<null_constant<T2>>
constexpr null_constant<std::common_type_t<T1, T2>>
operator+(null_constant<T1>, null_constant<T2>) noexcept {return {};}

template <class T1, T1 v1, class T2>
requires (!is_null_constant_v<numeric_constant<T1, v1>>) && std::is_arithmetic_v<T2>
constexpr std::common_type_t<T1, T2>
operator+(numeric_constant<T1, v1>, T2 v2) noexcept {return v1 + v2;}

// etc...
```

New rules

Rewriting arithmetic rules

- `numeric_constant<T, v>{} + numeric_constant<T, 0>{}` collapses to `numeric_constant<T, v>`
- `numeric_constant<T, v>{} * numeric_constant<T, 0>{}` collapses to `null_constant<T>`

Going beyond

- `numeric_constant` can have a `value_category`: `finite`, `infinite`, `nan`, `integer` ...
- We can rewrite all arithmetic rules to optimize during the compilation process: doable in C++23, but easier with reflection and `meta_info` in C++26

Conclusion

- We can have a nice unified interface corresponding to Butcher's Tableau: $a_{ij}, b_j^n, c_i, \beta_j^n$
- Coefficients are easy to specify
- Yet, the formulas can be optimized so that special numeric values do not cause overhead
- Multidimensional access through `std::mdspan` can then be generated for the method's coefficients

Outlooks

Toward a computer algebra system for C++

By combining last year strategies for symbolic computing and ode solvers so that we can have a nice syntax:

```
solve[{gpu}](
    derivative(y, t) = a * H0 * sqrt(
          Omega_r0 * pow(a, -4)
        + Omega_m0 * pow(a, -3)
        + Omega_k0 * pow(a, -2)
        + Omega_lambda0
    ),
    my_runge_kutta_solver
)
```

$\Rightarrow$ Goal for next year!

Science outlooks

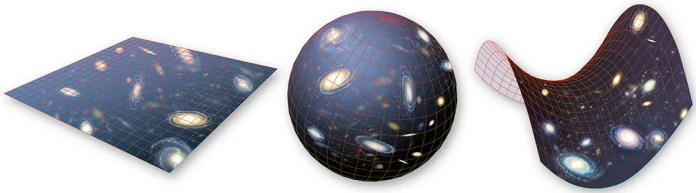
Basic yet interesting cosmological test

Have the same interface to integrate:

- space-time geometry (FLRW equations)
- direct N-body dynamics

for example to check:

- what happens to a N-body system with a global constant curvature?



Conclusion

Summary

Most important lesson in terms of software architecture

- The best software abstractions are derived from well-posed mathematical abstractions
- Conversely, if your problem is not well abstracted it is because solid mathematical foundations are missing

Ordinary Differential Equation solvers

- Good excercise to implement your own: it is not that difficult
- ODE methods have been well abstracted in mathematics: Runge-Kutta (RK), Linear Multistep Methods (LMM), General Linear Methods (GLM)
- Runge-Kutta methods can be represented as a Butcher Tableau
- General Linear Methods matrices A, B, U, V can abstract most common methods

Implementation outline

- Rely on coefficient multiarrays
- Intuitive syntax through expression templates
- Hyjacking arithmetic to remove operations will null coefficients
- Generate integration steps from coefficients

Outlooks

- Continue implementation and combine it with symbolic computing methods
- Interesting scientific use case even with simple integrators

Thank you for your attention

Any question?