High Performance ODE solver

Peter Christofides Paton

1 Introduction

Computing numerical solutions to dynamical systems is a well studied field in mathematics. However, as the time we measure the behaviour of the system, the required accuracy or the number of dimensions of the system[1] increases, this process can become very expensive computationally. This increased computational cost raises the need of high performance code and an understanding of the low level computer architecture which accompanies it.

As a proof of concept for my master's thesis, I have created a high performance C++ library for numerically solving ODEs, which is easily interfaced in Python for plotting.

2 Picard iterates

The library offers numerical solutions to ordinary differential equations of the form:

$$\frac{dx}{dt} = f(x,t), x(t_0) = x_0$$

It can be seen quite easily that the above problem is equivalent to finding a solution $x = \lambda(t)$ such that:

$$\lambda(t) = x_0 + \int_{t_0}^{t} f(x,t)dt$$

For which with $x = \lambda(t)$ we have,

$$\lambda(t) = x_0 + \int_{t_0}^{t} f(\lambda(t),t)dt$$

This gives rise to an iterative process

$$\lambda_{n+1}(t) = x_0 + \int_{t_0}^{t} f(\lambda_n(t),t)dt,$$

$$\lambda_0(t) = x_0.$$

which, by the Picard-Lindelof theorem, is known to converge locally when $f(x,t)$ is continuously differentiable.

[1] https://en.wikipedia.org/wiki/Curse_of_dimensionality

3 Implementation

The derivative, $\frac{dx}{dt} = f(x,t)$, along with the initial value pair (t_0, x_0) and local boundary of the function (t_{min}, t_{max}) are all passed into the constructor of `DifferentialEquationSolver.hpp`. The number of samples passed in defines how many points, t_i, are taken to calculate the numerical solution, with a greater sample count increasing the accuracy but also the time it takes to calculate the solution.

The differential equations solver maintains an internal list of the sampling points, t_i, and the current picard iteration values $\lambda_n(t_i)$. By calling the `picard_iteration()` function in `DifferentialEquationSolver.hpp` this internal list is recalculated to determine the value of each $\lambda_{n+1}(t_i)$. The integral between each successive sample points is calculated by the trapezium rule. So:

$$\lambda_{n+1}(t_0) = x_0,$$

$$\lambda_{n+1}(t_1) = x_0 + \int_{t_0}^{t_1} f(\lambda_n(t), t)dt = x_0 + \frac{1}{2}(\lambda_n(t_0) + \lambda_n(t_1)) * dt$$

$$\lambda_{n+1}(t_2) = x_0 + \int_{t_0}^{t_2} f(\lambda_n(t), t)dt = x_0 + \int_{t_0}^{t_1} f(\lambda_n(t), t)dt + \int_{t_1}^{t_2} f(\lambda_n(t), t)dt$$

$$= x_0 + \frac{1}{2}(\lambda_n(t_0) + \lambda_n(t_1)) * dt + \frac{1}{2}(\lambda_n(t_1) + \lambda_n(t_2)) * dt$$

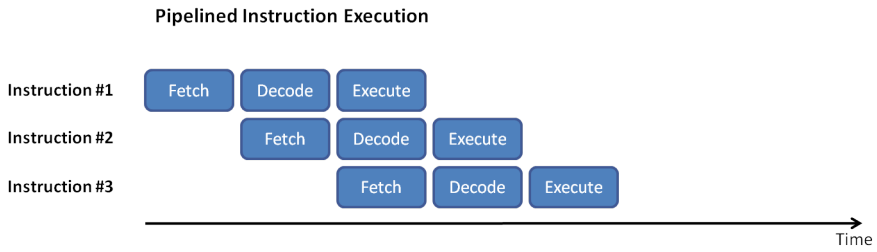
And so on.

This trapezium rule method of integration will remain accurate as long as the number of sample points provided is sufficiently high. We can afford to assume this given the high performance of this library.

The library also provides methods to access the sampling points, t_i, and the current picard iteration values $\lambda_n(t_i)$, which ensures all expensive computation is efficiently performed within the C++ library.

4 Optimisations

A CPU instruction pipeline can be simply abstracted as follows.



Initially the CPU will fetch the first instruction. In the next cycle it will decode the first instruction while simultaneously fetching the next instruction,

ideally again in the next cycle it will fetch the next instruction while the previous instructions move along to the execute and decode stages respectively, and so on.

Sometimes this process does not work so smoothly and some instructions will take longer than one cycle to fulfil - preventing movement in all the stages before it and stalling the CPU. For example, if the data is not cache resident in the memory stage then it can take much longer than one cycle to perform this memory access, this is an example of a data hazard.

When writing high performance code these various hazards must be mitigated to prevent CPU stalls and other inefficiencies occurring.

4.1 Language choice

C++ is the industry standard for high performance programming. I have outlined some of the relevant language features below.

Dynamic memory management: In C++ memory allocation is determined by the programmer. Therefore it does not suffer from performance overheads incurred by garbage collectors of languages with automatic memory management, such as Python. Additionally, the programmer has stronger guarantees of related data being stored contiguously in memory, making costly cache-misses more infrequent.

Compiled: C++ is a compiled language. In the context of this library the library needs to only be compiled once. In contrast to an interpreted language such as Python, the conversion to machine code must happen line by line every time the code is executed.

Statically typed: C++ is a statically typed language. This means the type of a variable must be determined at compile time. As opposed to dynamically typed languages where the validity of an operation on a variable must be determined when executing, incurring additional overhead at runtime.

4.2 Dynamic programming

Dynamic programming is an optimisation method in which previous calculations are stored and used to partially solve future calculations. For each sample point $t_i > t_0$ we have:

$$\lambda_{n+1}(t_i) = x_0 + \int_{t_0}^{t_i} f(\lambda_n(t), t)dt = x_0 + \int_{t_0}^{t_1} f(\lambda_n(t), t)dt + ... + \int_{t_{i-1}}^{t_i} f(\lambda_n(t), t)dt$$

$$= \lambda_{n+1}(t_{i-1}) + \int_{t_{i-1}}^{t_i} f(\lambda_n(t), t)dt$$

Therefore, if the function points are calculated outward from t_0 then we can use the value of λ_{n+1} at the previous point, t_{i-1}, in the calculation for the value of λ_{n+1} at t_i. (Note that $\forall n, \lambda_n(t_0) = x_0$). The implementation for this can be found in `DifferentialEquationSolver.cpp, picard_iteration()`.

4.3 Minimising library call overhead

Making a library call introduces some computational overhead, which we would like to minimise. If a user wants to perform many Picard iterations this would require a for loop and library and function call overheads for each iteration.

Therefore I have written a library method `perform_picard_iterations(int n)` which performs `n` Picard iterations, in one library call, eliminating the library call overhead for each iteration.

4.4 Minimising function call overhead

Function calls introduce another kind of control hazard in the instruction pipeline. Function calls require the CPU to perform a 'jump' operation, making the flow of computation jump to the the address of the called function in memory. However, the address in memory where the computation will continue cannot be determined until the a later stage of the instruction pipeline. This is an inefficiency as until the jump instruction reaches this later stage it cannot fetch new instructions.

This can be circumvented with function inlining. Inlining a function instructs the compiler to insert the body of the function at the point where the function is called. This prevents the control hazard outlined above and other function call overheads, while still maintaining modular code.

When minimising the library call overhead to perform `n` Picard iterations I inlined the function call for a single Picard iteration to prevent this function call overhead for each of the `n` iterations.

5 Example using the library

By means of example I have used my library to solve the diffential equation,

$$\frac{dx}{dt} = x, x(0) = 1.$$

And calculated the RMSE between the function points calculated and the known solution,

$$x(t) = e^t.$$

This can be found in `Example.py`.

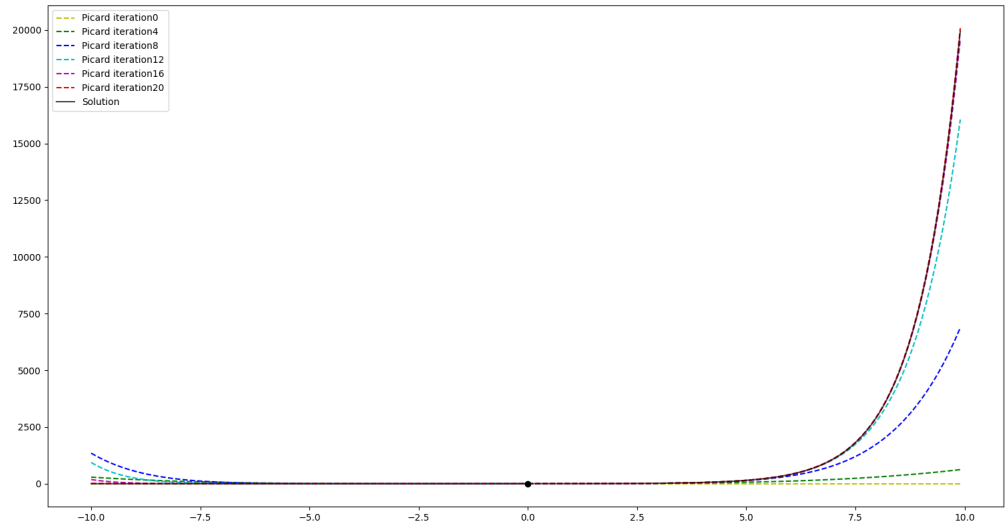


Figure 1: The Picard iterations plotted against the known solution.

```
Picard iteration 0: RMSE=3319.683251
Picard iteration 4: RMSE=3175.849330
Picard iteration 8: RMSE=2014.914147
Picard iteration 12: RMSE=556.109410
Picard iteration 16: RMSE=48.280580
Picard iteration 20: RMSE=22.538149
```

Figure 2: The RMSE of each 4 picard iterations plotted above against the known solution.