A brief introduction to Rochardson acceleration

Luca Formaggia
Politecnico di Milano
APSC Course

January 2022

1 Richardson acceleration

Richardson acceleration is a particular acceleration technique applicable when you have a quantity with a known convergence rate with respect to a parameter going to zero.
More precisely, is used when you are constructing an approximation $y(h)$ of a quantity y that depends on a parameter h (often a discretization step) and you may write,

$$y(h) = y + ch^k + O(h^l), \quad \text{with } l > k. \tag{1}$$

We say that $y(h)$ is an approximation for y at order k with respect to $h \to 0$. We assume that k is known. By neglecting the higher order term and performing simple algebraic manipulations we can discover that, chosen a $t > 1$ the function y^R given by

$$y^R(h) = \frac{t^k y(h/t) - y(h)}{t^k - 1} \tag{2}$$

provides an approximation of y of order l for $h \to 0$, thus improving the order of convergence (at the price of an additional evaluation).
In case we know the full expansion of the error:

$$y = y(h) + c_0 h^{k_0} + c_1 h^{k_1} + c_2 h^{k_2} + \ldots, \quad \text{with } k_j > k_{j-1}, \tag{3}$$

where the k_j are known (the c_j of course not), recursive application of Richardson formula (2) provides the following procedure:
Given a h, $m \geq 0$ and $t > 1$ (typically $t = 2$)

$$\begin{cases} y_0^R = y(h); \\ y_{i+1}^R(h) = \dfrac{t^{k_i} y_i^R(h/t) - y_i^R(h)}{t^{k_i} - 1}, \quad i = 0, \ldots m-1. \end{cases} \tag{4}$$

and we have an approximation error

$$y - y_m^R(h) = O(h^{k_m}) \tag{5}$$

Moreover, the quantity $|y_{i+1}^R(h) - y_i^R(h)|$ is a measure of the error.
An important note. If the expansion is correct, in (4) the numerator $t^{k_i} y_i^R(h/t) - y_i^R(h)$ will eventually lead to the difference of two nearby quantities: cancellation error! So, normally one does not take m too large. Another aspect is that we need to know the k_i. Of course one may assume that, for a linear converging $y(h)$ we have $k_0 = 1$ and $k_{j+1} = k_j + 1$, i.e. a Taylor like expansion. But if you know that some coefficients in the Taylor expansion are missing (for instance, only odd or even coefficients are different from zero) it is better to exploit it!

1.1 A pseudo code

Let's suppose that our approximation satifies (3), for simplicity we take here $t = 2$, which is often the case. We choose an h and a m, we want to get $y_m^R(h)$. Let's do thing step by step to realize what we need. Let $Y_{j,k} = y_k^R(h/t^{j-k}) = y_k^R(h/2^{j-k})$, thus $Y_{j,0} = y(h/2^j)$ and $Y_{k,k} = y_k^R(h)$
Let's look at the numerator of formula (4) (the denumerator are easy!)
• First step, $i = 0$, then

$$y_0^R(h) = y(h) \to Y_{0,0} = y(h)$$

That's easy. The second step ($i = 1$) is

$$y_1^R(h) = \frac{2^{k_0} y_0^R(h/2) - y_0^R(h)}{2^{k_0} - 1} \to Y_{1,1} = \frac{2^{k_0} Y_{1,0} - Y_{0,0}}{2^{k_0} - 1}$$

• So the second step is then implemented as

1. Compute $Y_{1,0} = y(h/2)$;
2. Compute $Y_{1,1} = \frac{2^{k_0} Y_{1,0} - Y_{0,0}}{2^{k_0} - 1}$

• The third step ($i = 2$) is a little harder:

$$y_2^R(h) = \frac{2^{k_1} y_1^R(h/2) - y_1^R(h)}{2^{k_1} - 1} \to Y_{2,2} = \frac{2^{k_1} Y_{2,1} - Y_{1,1}}{2^{k_1} - 1}$$

I need $Y_{2,1} = y_1^R(h/2)$:

$$y_1^R(h/2) = \frac{2^{k_0} y_0^R(h/4) - y_0^R(h/2)}{2^{k_0} - 1} \to Y_{2,1} = \frac{2^{k_0} Y_{2,0} - Y_{1,0}}{2^{k_0} - 1}$$

Therefore, step 3 reads:

1. Compute $Y_{2,0} = y(h/4)$;
2. For $j = 1, 2$ compute $Y_{2,j} = \frac{2^{k_{j-1}} Y_{2,j-1} - Y_{1,j-1}}{2^{k_{j-1}} - 1}$

In general, we need to built the $Y_{i,k}$ according to the following algorithm:

Algorithm 1

- For $i = 0, \ldots, m$
  1. $Y_{i,0} = y(h/2^i)$;
  2. For $j = 1, \ldots, i$
     (a) Compute $Y_{i,j} = \frac{2^{k_{j-1}} Y_{i,j-1} - Y_{i-1,j-1}}{2^{k_{j-1}} - 1}$
- The final value is $y_m^R(h) = Y_{m,m}$

We may note the following,

- Matrix $\mathbf{Y} \in \mathbb{R}^{m+1 \times m+1}$ is lower triangular;
- For each i the algorithm requires just two rows of $\mathbf{Y}$, namely rows i and the $i - 1$. A part the very first step. So I need to store only the two last rows, not the whole matrix.

2 The example

The example is inspired by the Wikipedia page on Richardson. It is a bit convoluted, but certainly more iteresting that accelerating the series for computing π. Suppose we are interested to the solution at the final time $y(T)$ of the differential equation

$$\begin{cases} y'(t) = -y^2(t) \quad t \in (0, T], \\ y(0) = 1. \end{cases} \tag{6}$$

Let's take $T = 5$. We can solve numerically that equation with a numerical method, for instance Crank-Nicolson, on a sufficiently fine grid, and then get $y(T)$ as the final computed numerical solution. But, since we are looking only for the final value (by the way it is equal to 1.66666.., so we can assess the error), it may be a waste of time having to compute the solution at all time steps. And we need a lot of time steps with Crank-Nicolson if we want a solution with 6 significant digits.
An alternative is to use Richardson. I have written a rather generic Richardon class able to compute the Richardson extrapolation, giving the coefficient k_j (by default they are $k_j = j + 1$) and a function **double(double)** that represents $y(h)$. I have also written the code for Crank-Nicolson (not so complicated).
Crank-Nicolson is a second order scheme (provided the forcing term is regular, as in our case), so I expect the error to behave as

$$y(T) - y_{CN}(t) = c_0 h^2 + c_1 h^3 + \dots$$

Thus, I give to the Richarson procedure $k_j = j + 2$ for $j = 0, \dots, m$, where m is the number of stages f Richardson I want to make. Here I take $t = 2$, thich means that at each stage I have to reduce h by one half.
The main points taken from the main:

- CrankNicolson is a function with signature

  ```
  template <class Function>
  std::tuple<std::vector<double>, std::vector<double>>
  CrankNicolson(Function const &f, double y0, double t0, double T, unsigned int n)
  ```

  It takes as input the forcing term, the initial value, initial and end time, and not h, but the number of intervals to be used, So, I need to define the forcing term

  ```
  auto f=[](double y,double t){return -y*y;};
  ```

  and the function $y(h)$ to pass to the Richardson object. I use a lambda also here (I have called the function r since y was already used):

  ```
  auto r = [&t0,&T,&f,&y0](double const & h)
           {
    unsigned n =(T-t0)/h;
    std::vector<double> sol;
    std::tie(std::ignore,sol) = apsc::CrankNicolson(f, y0, t0, T, n);
    return sol.back();
           };
  ```

  It takes h, compute the corresponding number of intervals, and returns the last value of the solution computed by CrankNicolson. I think that this example show the power of lambda expressions!

- Richardson is a class template.

  ```
  template<class Function=std::function<double(double const &)>>
  class Richardson;
  ```

which takes as template parameter the type of the function, defaulted to a function wrapper. Since I can initialize a function wrapper with a lambda expression, I give the constructor of the Richardson object the lambda, without stating the template parameter (I use the default)

```
apsc::LinearAlgebra::Richardson richardson{r,k};
```

Note that I could have stored the lambda expression directly! In this case the object is constructed as follows

```
apsc::LinearAlgebra::Richardson<decltype(r)> richardson{r,k};
```

and the template argument is now the type of the lambda expression (remember that a lambda expression is in fact a function object). Indeed, this solution produces a slightly more efficient code since it avoids the additional indirection due to the call of a std::function.

- I have used my Chrono utility to compute the time, comparing it with that necessary for the solution with Crank-Nicolson and 1000 intervals (needed to get the desired accuracy!).
- For the Richardson extrapolation, I start with just 4 intervals for Crank-Nicolson and I use M extrapolation levels (stages).
- Now the rest is rather standard and you can look and understand the code by yourself and note that the error at each stage of Richardson decreases very rapidly. We get an error of order 10^{-7} with a time roughly half that used by Crank-Nicolson with 1000 intervals. Not a great deal, but still a gain. Note: compile with `make DEBUG=no`.

In conclusion, in this example you have in fact two codes, a solver for scalar ODE that implements the Crank-Nicolson scheme, and the that for Richardson extrapolation.

2.1 A remark

For Richardon to be effective the initial h should not be too big. But it can be still rather large. Here the initial h is $5/4 = 1.25$. If you use a grater value, i.e. $h = 5/2 = 2.5$ the Crank-Nicolson code fails because the code that solves the non-linear equation arising from the Crank-Nicolson step fails. I have used the secant method in `basicZerofun.hpp` in the folder `LinearAlgebra/Utilities`. Remember to go in that folder and do `make install` otherwise you do not find the code for thesecant method.