Numerical Recipes for Multiprecision Computations

Henri Cohen

May 13, 2014

IMB, Université de Bordeaux

Multiprecision Computations: Goal

Multiprecision: from 38 to 1000 decimal digits.

Computations: Numerical differentiation, integration, summation, extrapolation, evaluation of continued fractions, Euler products and sums, complete finite sums such as Gauss and Jacobi sums.

Unsolved Problem: compute the Kloosterman sum

$$K(p) = \sum_{x \in (\mathbb{Z}/p\mathbb{Z})^*} e^{2\pi i (x+x^{-1})/p}$$

for p prime in $O(p^{1-\delta})$ for $\delta > 0$.

In this talk, only integration, summation, and extrapolation.

Multiprecision Computations: Goal

Multiprecision: from 38 to 1000 decimal digits.

Computations: Numerical differentiation, integration, summation, extrapolation, evaluation of continued fractions, Euler products and sums, complete finite sums such as Gauss and Jacobi sums.
Unsolved Problem: compute the Kloosterman sum

$$K(p) = \sum_{x \in (\mathbb{Z}/p\mathbb{Z})^*} e^{2\pi i (x+x^{-1})/p}$$

for p prime in $O(p^{1-\delta})$ for $\delta > 0$.

In this talk, only integration, summation, and extrapolation.

Multiprecision Computations: Goal

Multiprecision: from 38 to 1000 decimal digits.

Computations: Numerical differentiation, integration, summation, extrapolation, evaluation of continued fractions, Euler products and sums, complete finite sums such as Gauss and Jacobi sums.
Unsolved Problem: compute the Kloosterman sum

$$K(p) = \sum_{x \in (\mathbb{Z}/p\mathbb{Z})^*} e^{2\pi i (x+x^{-1})/p}$$

for p prime in $O(p^{1-\delta})$ for $\delta > 0$.

In this talk, only integration, summation, and extrapolation.

Multiprecision Numerical Analysis

In what follows, D is always the number of desired decimal digits.
Can use multiprecision to compensate for errors in the algorithms.

- For instance, summing millions of terms we work at accuracy $D + 9$ (same as $D + 19$ on 64 bit machines).
- If for some reason there is a loss of accuracy (example: compute the power series for e^{-x} with x large), work at accuracy $3D/2$, $2D$, or more.
- If the algorithm is fundamentally unstable, we can hope to compensate by working at accuracy $D \log(D)$ or similar.

All this is cheating, but works very well in practice.

Compendium of the Methods: Integration on [a,b]

At least nine methods studied:

1. Trapezes, Simpson, and more generally Newton–Cotes methods with equally spaced abscissas.
2. Newton–Cotes methods with abscissas roots of Chebyshev polynomials.
3. Classical Romberg method using Richardson extrapolation, the `intnumromb` function of `Pari/GP`.
4. Romberg method using Richardson 2-3 extrapolation, more efficient.
5. Extrapolation methods using Lagrange extrapolation.
6. Extrapolation methods using Zagier extrapolation (see below for these extrapolation methods).
7. Gauss–Legendre integration.
8. Doubly-exponential methods, homemade.
9. The `intnum` function of `Pari/GP` also doubly-exponential.

Compendium of the Methods: Integration on [a,b]

At least nine methods studied:

1. Trapezes, Simpson, and more generally Newton–Cotes methods with equally spaced abscissas.
2. Newton–Cotes methods with abscissas roots of Chebyshev polynomials.
3. Classical Romberg method using Richardson extrapolation, the `intnumromb` function of `Pari/GP`.
4. Romberg method using Richardson 2-3 extrapolation, more efficient.
5. Extrapolation methods using Lagrange extrapolation.
6. Extrapolation methods using Zagier extrapolation (see below for these extrapolation methods).
7. Gauss–Legendre integration.
8. Doubly-exponential methods, homemade.
9. The `intnum` function of `Pari/GP` also doubly-exponential.

Compendium of the Methods: Integration on [a,b]

At least nine methods studied:

1. Trapezes, Simpson, and more generally Newton–Cotes methods with equally spaced abscissas.
2. Newton–Cotes methods with abscissas roots of Chebyshev polynomials.
3. Classical Romberg method using Richardson extrapolation, the `intnumromb` function of `Pari/GP`.
4. Romberg method using Richardson 2-3 extrapolation, more efficient.
5. Extrapolation methods using Lagrange extrapolation.
6. Extrapolation methods using Zagier extrapolation (see below for these extrapolation methods).
7. Gauss–Legendre integration.
8. Doubly-exponential methods, homemade.
9. The `intnum` function of `Pari/GP` also doubly-exponential.

Compendium of the Methods: Integration on [a,b]

At least nine methods studied:

1. Trapezes, Simpson, and more generally Newton–Cotes methods with equally spaced abscissas.
2. Newton–Cotes methods with abscissas roots of Chebyshev polynomials.
3. Classical Romberg method using Richardson extrapolation, the `intnumromb` function of `Pari/GP`.
4. Romberg method using Richardson 2-3 extrapolation, more efficient.
5. Extrapolation methods using Lagrange extrapolation.
6. Extrapolation methods using Zagier extrapolation (see below for these extrapolation methods).
7. Gauss–Legendre integration.
8. Doubly-exponential methods, homemade.
9. The `intnum` function of `Pari/GP` also doubly-exponential.

Newton–Cotes Methods: Theory

Subdivide the interval of integration into N equal small subintervals.
In each subinterval, Lagrange-interpolate the integrand at $k+1$ points by a polynomial of degree k.
Original trapezes, Simpson, Newton–Cotes: equally spaced points with $k = 1, 2, \geq 4$ ($k = 2n+1$ is the same as $k = 2n$).
Problem: Runge's phenomenon: approximation by large degree polynomials is far from uniform. To diminish this effect: Chebyshev nodes: instead, choose scaled roots of Chebyshev polynomials in each subinterval.
Almost always better and more stable, attenuated Runge's phenomenon but still present.

Newton–Cotes Methods: Theory

Subdivide the interval of integration into N equal small subintervals.
In each subinterval, Lagrange-interpolate the integrand at $k+1$ points by a polynomial of degree k.
Original trapezes, Simpson, Newton–Cotes: equally spaced points with $k = 1, 2, \geq 4$ ($k = 2n+1$ is the same as $k = 2n$).
Problem: Runge's phenomenon: approximation by large degree polynomials is far from uniform. To diminish this effect: Chebyshev nodes: instead, choose scaled roots of Chebyshev polynomials in each subinterval.
Almost always better and more stable, attenuated Runge's phenomenon but still present.

Newton–Cotes Methods: Implementation

Must choose N, k, and working accuracy. Experimentation shows the following:

- For the initial Newton–Cotes:
  Choose $N = 64$ (note independent of D!!!), $k = D/2$, and work at accuracy $3D/2 + 9$.
- For Newton–Cotes with Chebyshev nodes:
  Choose also $N = 64$, but $k = 5D/12$ and work at accuracy $5D/4 + 9$ (thus slightly lower k and working accuracy).

Newton–Cotes Methods: Implementation

Must choose N, k, and working accuracy. Experimentation shows the following:

- For the initial Newton–Cotes:
  Choose $N = 64$ (note independent of D!!!), $k = D/2$, and work at accuracy $3D/2 + 9$.
- For Newton–Cotes with Chebyshev nodes:
  Choose also $N = 64$, but $k = 5D/12$ and work at accuracy $5D/4 + 9$ (thus slightly lower k and working accuracy).

Newton–Cotes Methods: Results

Very reasonable methods, even for $D = 1000$, although slower by a factor of 5 to 50 than other methods: for $D = 1000$, with the choice of parameters above, they require around 30000 function evaluations.

For the following computation, Newton–Cotes–Chebyshev is the only method giving sensible results:

$$J(T) = \frac{1}{2\pi i} \int_{R(T)} \frac{\zeta'(z)}{\zeta(z)}\, dz \ ,$$

where $R(T)$ is the boundary of the rectangle $[0, 1] \times [1, T]$, which counts the number of zeros of $\zeta(s)$ in the critical strip up to height T (by far not the best way, but simply an example). Very close to integers up to $T = 100$, still recognizable up to $T = 300$.

No other method gives sensible results.

Newton–Cotes Methods: Results

Very reasonable methods, even for $D = 1000$, although slower by a factor of 5 to 50 than other methods: for $D = 1000$, with the choice of parameters above, they require around 30000 function evaluations.

For the following computation, Newton–Cotes–Chebyshev is the only method giving sensible results:

$$J(T) = \frac{1}{2\pi i} \int_{R(T)} \frac{\zeta'(z)}{\zeta(z)}\, dz \ ,$$

where $R(T)$ is the boundary of the rectangle $[0, 1] \times [1, T]$, which counts the number of zeros of $\zeta(s)$ in the critical strip up to height T (by far not the best way, but simply an example).

Very close to integers up to $T = 100$, still recognizable up to $T = 300$.

No other method gives sensible results.

Extrapolation Methods: Theory I

To compute $I = \int_a^b f(x)\,dx$, by trapezes we have $I = \lim_{n\to\infty} u(n)$ with

$$u(n) = \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{1\le j\le n-1} f\left(a + j\frac{b-a}{n}\right)\right),$$

and by Euler–MacLaurin we have

$$u(n) = I + \frac{c_2}{n^2} + \frac{c_4}{n^4} + \frac{c_6}{n^6} + \cdots$$

(note: only even powers of $1/n$).

Idea: extrapolate the sequence $u(n)$, knowing its asymptotic expansion.

Extrapolation Methods: Theory I

To compute $I = \int_a^b f(x)\,dx$, by trapezes we have $I = \lim_{n\to\infty} u(n)$ with

$$u(n) = \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{1\le j\le n-1} f\left(a + j\frac{b-a}{n}\right)\right),$$

and by Euler–MacLaurin we have

$$u(n) = I + \frac{c_2}{n^2} + \frac{c_4}{n^4} + \frac{c_6}{n^6} + \cdots$$

(note: only even powers of $1/n$).
Idea: extrapolate the sequence $u(n)$, knowing its asymptotic expansion.

Extrapolation Methods: Theory II

Several methods: the most classical, Romberg integration, is based on Richardson extrapolation:

$$u'(n) = (4u(2n) - u(n))/3 = I + c'_4/n^4 + c'_6/n^6 + \cdots ,$$

continue in this way:

$$u''(n) = (16u'(2n) - u'(n))/15 = I + c''_6/n^6 + \cdots .$$

Convergence faster and faster.

In double precision, very nice since need to do this only a few times. In multiprecision, inapplicable: time in $O(\exp(a \cdot D^{1/2}))$. Richardson can be improved (2-3 method), but still slow.

Extrapolation Methods: Theory II

Several methods: the most classical, Romberg integration, is based on Richardson extrapolation:

$$u'(n) = (4u(2n) - u(n))/3 = I + c'_4/n^4 + c'_6/n^6 + \cdots ,$$

continue in this way:

$$u''(n) = (16u'(2n) - u'(n))/15 = I + c''_6/n^6 + \cdots .$$

Convergence faster and faster.
In double precision, very nice since need to do this only a few times. In multiprecision, inapplicable: time in $O(\exp(a \cdot D^{1/2}))$.
Richardson can be improved (2-3 method), but still slow.

Extrapolation Methods: Theory and Implementation

Much better extrapolation methods, but less robust, based on Lagrange interpolation: look for a function f defined on the reals such that $u(n) = f(1/n^2)$, so that around $z = 0$ we have $f(z) = I + c_2 z + c_4 z^2 + \cdots$. Use Lagrange interpolation on several values of $u(n)$ to find a polynomial f, and compute

$$I = f(0) \approx \frac{2}{(2N)!} \sum_{n=1}^{N} (-1)^{N-n} \binom{2N}{N-n} n^{2N} u(n) .$$

Experiment leads to choosing $N = 0.9D$ and working at accuracy $4D/3$ leads to good results, and the times are reasonable, contrary to Romberg.
Much slower than other methods (420000 function evaluations for $D = 1000$) but still feasible at $D = 1000$, while Romberg explodes at more than 57 decimals.

Extrapolation Methods: Theory and Implementation

Much better extrapolation methods, but less robust, based on Lagrange interpolation: look for a function f defined on the reals such that $u(n) = f(1/n^2)$, so that around $z = 0$ we have $f(z) = I + c_2 z + c_4 z^2 + \cdots$. Use Lagrange interpolation on several values of $u(n)$ to find a polynomial f, and compute

$$I = f(0) \approx \frac{2}{(2N)!} \sum_{n=1}^{N} (-1)^{N-n} \binom{2N}{N-n} n^{2N} u(n) .$$

Experiment leads to choosing $N = 0.9D$ and working at accuracy $4D/3$ leads to good results, and the times are reasonable, contrary to Romberg.
Much slower than other methods (420000 function evaluations for $D = 1000$) but still feasible at $D = 1000$, while Romberg explodes at more than 57 decimals.

Gaussian Integration: Theory I

This is a very classical method: it scales extremely well in multiprecision computations, and when applicable, it is an order of magnitude faster than all other methods, not including the initialization step which is much longer but done once and for all.

To compute a family of integrals $\int_a^b f(x)w(x)\,dx$ for a given weight function $w(x)$ (or more generally for a suitable measure $w(x)dx$), using a suitable Gram–Schmidt orthogonalization scheme we compute the unique family of monic polynomials P_n of degree n such that $\int_a^b P_m(x)P_n(x)w(x) = 0$ for $m \neq n$.

Gaussian Integration: Theory I

This is a very classical method: it scales extremely well in multiprecision computations, and when applicable, it is an order of magnitude faster than all other methods, not including the initialization step which is much longer but done once and for all.

To compute a family of integrals $\int_a^b f(x)w(x)\,dx$ for a given weight function $w(x)$ (or more generally for a suitable measure $w(x)dx$), using a suitable Gram–Schmidt orthogonalization scheme we compute the unique family of monic polynomials P_n of degree n such that $\int_a^b P_m(x)P_n(x)w(x) = 0$ for $m \neq n$.

Gaussian Integration: Theory II

Then for a suitable N we compute the roots x_i of P_N, the nodes, which will all be real, simple, and in the interval $]a, b[$, and then we compute the weights w_i by the formula

$$w_i = \frac{1}{P'_N(x_i)} \int_a^b w(x) \frac{P_N(x)}{x - x_i}\, dx \ .$$

We then have

$$I = \int_a^b f(x) w(x)\, dx \approx \sum_{1 \le i \le N} w_i f(x_i) \ .$$

Gaussian Integration: Theory III

Simplest weights $w(x)$ such as $w(x) = 1$, $w(x) = 1/\sqrt{1-x^2}$, $w(x) = \sqrt{1-x^2}$, etc... give rise to well-known families of orthogonal polynomials such as Legendre polynomials or Chebyshev polynomials of the first and second kind, for which everything is known explicitly (the coefficients of the polynomials, no need to compute an integral for the weights, etc...).

For more complicated weights, there are explicit algorithms which are rather unstable, so need to work at accuracy of the type $D\log(D)$, but worth it since it is only the initialization step: after, very fast.

Gaussian Integration: Theory III

Simplest weights $w(x)$ such as $w(x) = 1$, $w(x) = 1/\sqrt{1-x^2}$, $w(x) = \sqrt{1-x^2}$, etc... give rise to well-known families of orthogonal polynomials such as Legendre polynomials or Chebyshev polynomials of the first and second kind, for which everything is known explicitly (the coefficients of the polynomials, no need to compute an integral for the weights, etc...).

For more complicated weights, there are explicit algorithms which are rather unstable, so need to work at accuracy of the type $D\log(D)$, but worth it since it is only the initialization step: after, very fast.

Gaussian Integration: Implementation I

Experiment shows that a good choice is $N = 3D/4$, and working at accuracy $3D/2 + 9$ for classical polynomials, but at the much higher accuracy $D\log(D)/1.6$ if one needs to compute explicitly the orthogonal polynomials from scratch (essentially from the moments of $w(x)dx$). For $D = 1000$, requires only 750 function evaluations, compared to 30000 for Newton–Cotes and 5000 to 15000 for doubly-exponential methods, but 15 seconds initialization for classical polynomials.

Example: $\int_0^1 \log(\Gamma(1+x))\,dx$ for $D = 1000$ is computed in 8.33 seconds, compared to 2 minutes for doubly-exponential methods, and 4 minutes for Newton–Cotes.

Gaussian Integration: Implementation I

Experiment shows that a good choice is $N = 3D/4$, and working at accuracy $3D/2 + 9$ for classical polynomials, but at the much higher accuracy $D\log(D)/1.6$ if one needs to compute explicitly the orthogonal polynomials from scratch (essentially from the moments of $w(x)dx$). For $D = 1000$, requires only 750 function evaluations, compared to 30000 for Newton–Cotes and 5000 to 15000 for doubly-exponential methods, but 15 seconds initialization for classical polynomials. Example: $\int_0^1 \log(\Gamma(1+x))\,dx$ for $D = 1000$ is computed in 8.33 seconds, compared to 2 minutes for doubly-exponential methods, and 4 minutes for Newton–Cotes.

Gaussian Integration: Implementation II

Method of choice, but less robust than doubly-exponential methods: integrand needs to be "close" to a polynomial times the weight measure $w(x)$. Contrived example: $\int_0^1 dx/(x^\pi + x^{1.4} + 1)$ only 10% correct decimals. But $\int_0^{3/2} \tan(x)\,dx$ has only 30% correct decimals because of the proximity of the pole at $x = \pi/2$. Even $\int_0^1 \tan(x)\,dx$ has only 90% correct decimals.

Doubly-Exponential Integration: Theory I

Method invented by Mori–Takahashi in 1968. Extremely robust, although much slower than Gaussian integration when both apply. Two ideas:

• Theorem: if $F(t)$ tends to 0 doubly-exponentially as $t \to \pm\infty$ (i.e., like $\exp(-a\exp(b|t|))$), the optimal way to compute $I = \int_{-\infty}^{\infty} F(t)\,dt$ is to use Riemann sums (rectangles or trapezes, same), i.e., to choose N and h and write

$$I \approx h \sum_{-N \le m \le N} F(mh)\ .$$

Doubly-Exponential Integration: Theory II

- Reduce any other integral to this case by suitable changes of variable: in the compact case $[a, b]$, transform $I = \int_a^b f(x)\,dx$ into the above by setting

$$x = \phi(t) := \frac{a+b}{2} + \frac{b-a}{2}\tanh((\pi/2)\sinh(t))\ .$$

$dx/dt = \phi'(t)$ tends to 0 doubly-exponentially, and one can show that $\pi/2$ is optimal for a wide class of functions.

Strong restriction, but in general satisfied in mathematical practice: $f(x)$ must be holomorphic (or at least meromorphic with known polar parts) in a domain containing $[a, b]$.

Doubly-Exponential Integration: Theory II

- Reduce any other integral to this case by suitable changes of variable: in the compact case $[a, b]$, transform $I = \int_a^b f(x)\,dx$ into the above by setting

$$x = \phi(t) := \frac{a+b}{2} + \frac{b-a}{2}\tanh((\pi/2)\sinh(t))\ .$$

$dx/dt = \phi'(t)$ tends to 0 doubly-exponentially, and one can show that $\pi/2$ is optimal for a wide class of functions. Strong restriction, but in general satisfied in mathematical practice: $f(x)$ must be holomorphic (or at least meromorphic with known polar parts) in a domain containing $[a, b]$.

Doubly-Exponential Integration: Implementation

Experimentation shows that reasonable choices for N and h are

$$N = \frac{D\log(D)}{1.86} \quad \text{and} \quad h = \frac{\log(2\pi N/\log(N))}{N},$$

and since the method is very stable, one can simply work at accuracy $D+9$ to compensate for rounding errors in the sums. Of course, all the hyperbolic functions are computed during initialization.

Sample timings: for $D = 1000$, initialization takes 1.03 seconds, the method requires 7430 function evaluations, and the times vary between 0.05 and 5 seconds depending on the time taken to compute the function, except for higher transcendental functions such as $\Gamma(x)$ or $\zeta(x)$ for which the time is of course much larger.

Doubly-Exponential Integration: Implementation

Experimentation shows that reasonable choices for N and h are

$$N = \frac{D\log(D)}{1.86} \quad \text{and} \quad h = \frac{\log(2\pi N/\log(N))}{N},$$

and since the method is very stable, one can simply work at accuracy $D+9$ to compensate for rounding errors in the sums. Of course, all the hyperbolic functions are computed during initialization.

Sample timings: for $D = 1000$, initialization takes 1.03 seconds, the method requires 7430 function evaluations, and the times vary between 0.05 and 5 seconds depending on the time taken to compute the function, except for higher transcendental functions such as $\Gamma(x)$ or $\zeta(x)$ for which the time is of course much larger.

Doubly-Exponential Integration: Comment

The current `intnum` function of `Pari/GP` is about three times slower due to a less efficient implementation. This should be changed hopefully soon.

Integration on Infinite Intervals: Methods

Can always reduce to $[a, \infty[$. Need to distinguish functions tending to 0 slowly or fast (can also treat oscillating functions, but not in this talk). Evidently no need to tend to 0, and can go from slow to fast by changes of variables. Still, good guiding principle.

Not many methods available:

1. For slow functions $f(x)$, Gauss–Legendre on $f(1/x)$, or a suitable change of variable for doubly-exponential integration.
2. For functions $f(x) = e^{-x}g(x)$, use Gauss–Laguerre, Gauss–Legendre on $g(1/x)$, or a suitable change of variable for doubly-exponential integration.

Integration on Infinite Intervals: Methods

Can always reduce to $[a, \infty[$. Need to distinguish functions tending to 0 slowly or fast (can also treat oscillating functions, but not in this talk). Evidently no need to tend to 0, and can go from slow to fast by changes of variables. Still, good guiding principle.

Not many methods available:

1. For slow functions $f(x)$, Gauss–Legendre on $f(1/x)$, or a suitable change of variable for doubly-exponential integration.
2. For functions $f(x) = e^{-x}g(x)$, use Gauss–Laguerre, Gauss–Legendre on $g(1/x)$, or a suitable change of variable for doubly-exponential integration.

Integration on Infinite Intervals: Gaussian Methods

For slow functions: change x into $1/x$. For instance $\int_1^\infty f(x)\,dx = \int_0^1 (f(1/x)/x^2)\,dx$, and compute the latter integral by Gauss–Legendre. Inapplicable for fast since for instance $\int_1^\infty e^{-x}g(x) = \int_0^1 e^{-1/x}g(x)/x^2$, and $e^{-1/x}$ has a bad singularity at $x = 0$.

For fast functions: typical examples $f(x) = e^{-x}g(x)$. The textbook method is Gauss–Laguerre: unfortunately rarely applicable because $g(x)$ rarely well approximated by polynomials.

Gauss–Laguerre inverse (new?): approximate instead $g(1/x)$ by polynomials. This now works in many other (still few) cases, such as $\int_0^\infty e^{-x}/(1+x)\,dx$. Need to compute specific orthogonal polynomials, slow initialization.

Integration on Infinite Intervals: Gaussian Methods

For slow functions: change x into $1/x$. For instance $\int_1^\infty f(x)\,dx = \int_0^1 (f(1/x)/x^2)\,dx$, and compute the latter integral by Gauss–Legendre. Inapplicable for fast since for instance $\int_1^\infty e^{-x}g(x) = \int_0^1 e^{-1/x}g(x)/x^2$, and $e^{-1/x}$ has a bad singularity at $x = 0$.

For fast functions: typical examples $f(x) = e^{-x}g(x)$. The textbook method is Gauss–Laguerre: unfortunately rarely applicable because $g(x)$ rarely well approximated by polynomials.

Gauss–Laguerre inverse (new?): approximate instead $g(1/x)$ by polynomials. This now works in many other (still few) cases, such as $\int_0^\infty e^{-x}/(1+x)\,dx$. Need to compute specific orthogonal polynomials, slow initialization.

Integration on Infinite Intervals: Gaussian Methods

For slow functions: change x into $1/x$. For instance $\int_1^\infty f(x)\,dx = \int_0^1 (f(1/x)/x^2)\,dx$, and compute the latter integral by Gauss–Legendre. Inapplicable for fast since for instance $\int_1^\infty e^{-x}g(x) = \int_0^1 e^{-1/x}g(x)/x^2$, and $e^{-1/x}$ has a bad singularity at $x = 0$.

For fast functions: typical examples $f(x) = e^{-x}g(x)$. The textbook method is Gauss–Laguerre: unfortunately rarely applicable because $g(x)$ rarely well approximated by polynomials.

Gauss–Laguerre inverse (new?): approximate instead $g(1/x)$ by polynomials. This now works in many other (still few) cases, such as $\int_0^\infty e^{-x}/(1+x)\,dx$. Need to compute specific orthogonal polynomials, slow initialization.

Integration on Infinite Intervals: Doubly-Exponential Methods

For computing $\int_0^\infty f(x)\,dx$:
If f tends to 0 slowly, use the change of variable

$$x = \phi(t) := \exp(\sinh(t)) .$$

If f tends to 0 simply exponentially, say as e^{-x} (easy to modify), use the change of variable

$$x = \phi(t) := \exp(t - \exp(-t)) .$$

Implementation: choose the larger value $N = D\log(D)/1.05$ if slow and $N = D\log(D)/1.76$ if fast, and the same h as in the compact case. Same comments: use Gaussian methods if possible (10 times faster at $D = 1000$), otherwise doubly-exponential methods, quite robust.

Integration on Infinite Intervals: Doubly-Exponential Methods

For computing $\int_0^\infty f(x)\,dx$:
If f tends to 0 slowly, use the change of variable

$$x = \phi(t) := \exp(\sinh(t)) .$$

If f tends to 0 simply exponentially, say as e^{-x} (easy to modify), use the change of variable

$$x = \phi(t) := \exp(t - \exp(-t)) .$$

Implementation: choose the larger value $N = D\log(D)/1.05$ if slow and $N = D\log(D)/1.76$ if fast, and the same h as in the compact case. Same comments: use Gaussian methods if possible (10 times faster at $D = 1000$), otherwise doubly-exponential methods, quite robust.

Integration on Infinite Intervals: Doubly-Exponential Methods

For computing $\int_0^\infty f(x)\,dx$:
If f tends to 0 slowly, use the change of variable

$$x = \phi(t) := \exp(\sinh(t)) .$$

If f tends to 0 simply exponentially, say as e^{-x} (easy to modify), use the change of variable

$$x = \phi(t) := \exp(t - \exp(-t)) .$$

Implementation: choose the larger value $N = D\log(D)/1.05$ if slow and $N = D\log(D)/1.76$ if fast, and the same h as in the compact case. Same comments: use Gaussian methods if possible (10 times faster at $D = 1000$), otherwise doubly-exponential methods, quite robust.

Summation Methods: Alternating Series I

We come to numerical summation methods. We consider only two types of sums: $\sum_{n\geq a} f(n)$ (series with positive terms) and alternating series $\sum_{n\geq a}(-1)^n f(n)$, in both cases with $f(n) > 0$ and tending regularly to 0 at infinity.

For alternating series, I have implemented five different methods, all having an analogue for series with positive terms. No need to go into details: the `sumalt` function (implemented in `Pari/GP`), explained in detail in a paper of Rodriguez-Villegas, Zagier, and the author, but known before, is at the same time the fastest, most robust, and simplest method, and in addition is the only method which uses only the values of $f(n)$ for n integral, and not the extension of f to the real line. In some cases (when the summand takes a long time to compute), a variant of this method can be faster.

Summation Methods: Alternating Series I

We come to numerical summation methods. We consider only two types of sums: $\sum_{n\geq a} f(n)$ (series with positive terms) and alternating series $\sum_{n\geq a}(-1)^n f(n)$, in both cases with $f(n) > 0$ and tending regularly to 0 at infinity.

For alternating series, I have implemented five different methods, all having an analogue for series with positive terms. No need to go into details: the `sumalt` function (implemented in `Pari/GP`), explained in detail in a paper of Rodriguez-Villegas, Zagier, and the author, but known before, is at the same time the fastest, most robust, and simplest method, and in addition is the only method which uses only the values of $f(n)$ for n integral, and not the extension of f to the real line. In some cases (when the summand takes a long time to compute), a variant of this method can be faster.

Summation of Alternating Series II

Idea: write $f(n) = \int_0^1 x^n w(x)\,dx$ for some measure $w(x)dx$, assumed positive. Then $S := \sum_{n\geq 0}(-1)^n f(n) = \int_0^1 (w(x)/(1+x))\,dx$, so for any polynomial P_N such that $P_N(-1) \neq 0$ we have

$$S = \frac{1}{P_N(-1)} \sum_{0\leq j\leq N-1} c_{N,j} u(j) + R_N \ , \text{ with } |R_N| \leq \frac{\sup_{x\in[0,1]} |P_N(x)|}{|P_N(-1)|} S \ ,$$

where

$$\frac{P_N(-1) - P_N(X)}{X+1} = \sum_{0\leq j\leq N-1} c_{N,j} X^j \ .$$

Summation of Alternating Series III

An almost optimal choice is the shifted Chebyshev polynomial $P_N(X) = T_N(1-2X)$ for which the relative error $|R_N/S|$ satisfies $|R_N/S| \leq (3+2\sqrt{2})^{-N}$, so it is immediate to determine that $N = 1.31D$. An additional advantage of these polynomials is that the coefficients $c_{N,j}$ can be computed "on the fly".

For $D = 1000$, the implementation requires between 10 milliseconds and 1 second depending on the complexity of computing the summand.

Note that all alternating summation methods can correctly compute the "sum" of nonconvergent series such as $\sum_{n\geq 2}(-1)^n \log(n)$ or $\sum_{n\geq 2}(-1)^n n$.

Summation of Alternating Series III

An almost optimal choice is the shifted Chebyshev polynomial $P_N(X) = T_N(1-2X)$ for which the relative error $|R_N/S|$ satisfies $|R_N/S| \leq (3+2\sqrt{2})^{-N}$, so it is immediate to determine that $N = 1.31D$. An additional advantage of these polynomials is that the coefficients $c_{N,j}$ can be computed "on the fly".
For $D = 1000$, the implementation requires between 10 milliseconds and 1 second depending on the complexity of computing the summand.
Note that all alternating summation methods can correctly compute the "sum" of nonconvergent series such as $\sum_{n\geq 2}(-1)^n \log(n)$ or $\sum_{n\geq 2}(-1)^n n$.

Summation of Alternating Series III

An almost optimal choice is the shifted Chebyshev polynomial $P_N(X) = T_N(1-2X)$ for which the relative error $|R_N/S|$ satisfies $|R_N/S| \leq (3+2\sqrt{2})^{-N}$, so it is immediate to determine that $N = 1.31D$. An additional advantage of these polynomials is that the coefficients $c_{N,j}$ can be computed "on the fly".
For $D = 1000$, the implementation requires between 10 milliseconds and 1 second depending on the complexity of computing the summand.
Note that all alternating summation methods can correctly compute the "sum" of nonconvergent series such as $\sum_{n\geq 2}(-1)^n \log(n)$ or $\sum_{n\geq 2}(-1)^n n$.

Summation of Positive Series: Compendium

Five methods studied:

1. van Wijngaarten's method.
2. Use of the Euler–MacLaurin formula.
3. Use of the Abel–Plana formula.
4. Discrete Euler–MacLaurin formula.
5. Gaussian summation, method due to H. Monien.

van Wijngaarten's Method

Corresponds to the `sumpos` function of `Pari/GP`. Uses the formula

$$S := \sum_{n \geq 1} f(n) = \sum_{a \geq 1} (-1)^{a-1} F(a) \text{ with } F(a) = \sum_{j \geq 0} 2^j f(2^j a) \ ,$$

so transforms a series with positive terms into an alternating series.

Uses $f(n)$ for very large values of n: thus useless if $f(n)$ takes long to compute when n is large, but useful if $f(n)$ is "explicit". In addition, as for `sumalt`, only uses values of f at integral points. Apart from that, orders of magnitude slower than other methods when $f(n)$ tends to 0 slowly (otherwise in general sum easy to compute).

van Wijngaarten's Method

Corresponds to the `sumpos` function of `Pari/GP`. Uses the formula

$$S := \sum_{n \geq 1} f(n) = \sum_{a \geq 1} (-1)^{a-1} F(a) \text{ with } F(a) = \sum_{j \geq 0} 2^j f(2^j a) \ ,$$

so transforms a series with positive terms into an alternating series.
Uses $f(n)$ for very large values of n: thus useless if $f(n)$ takes long to compute when n is large, but useful if $f(n)$ is "explicit".
In addition, as for `sumalt`, only uses values of f at integral points. Apart from that, orders of magnitude slower than other methods when $f(n)$ tends to 0 slowly (otherwise in general sum easy to compute).

Euler–MacLaurin: Theory

Very classical method: see Bourbaki. Basic formula (many variants), assuming $\sum_{m\geq a} f(m)$ converges:

$$\sum_{m\geq a} f(m) = \sum_{a\leq m\leq N-1} f(m) + \frac{f(N)}{2} + \int_N^\infty f(t)\,dt$$
$$- \sum_{1\leq j\leq \lfloor k/2\rfloor} \frac{B_{2j}}{(2j)!} f^{(2j-1)}(N) + \frac{(-1)^k}{k!} \int_N^\infty f^{(k)}(t) B_k(\{t\})\,dt\ ,$$

with B_{2j} the Bernoulli numbers and $B_k(t)$ the Bernoulli polynomials.

Euler–MacLaurin: Implementation

Need to compute $\int_N^\infty f(t)\,dt$, done using one of the methods seen above (usually doubly-exponential integration since infinite interval). However, need also to compute the derivatives $f^{(2j-1)}(N)$, not always easy. In some cases, explicit. In many other cases, under `Pari/GP` simply write $f(N+x)$ computed as a power series in x, and obtain the derivatives from its coefficients. In other cases, impossible to do this.

Analysis and experimentation shows that a good choice is k close to $0.41D$ and even, and $N = 6.6D$. This is quite heuristic but works well. Very reasonable method, but only if one can compute the derivatives.

Euler–MacLaurin: Implementation

Need to compute $\int_N^\infty f(t)\,dt$, done using one of the methods seen above (usually doubly-exponential integration since infinite interval). However, need also to compute the derivatives $f^{(2j-1)}(N)$, not always easy. In some cases, explicit. In many other cases, under `Pari/GP` simply write $f(N+x)$ computed as a power series in x, and obtain the derivatives from its coefficients. In other cases, impossible to do this.
Analysis and experimentation shows that a good choice is k close to $0.41D$ and even, and $N = 6.6D$. This is quite heuristic but works well. Very reasonable method, but only if one can compute the derivatives.

Abel–Plana

This is currently the `sumnum` function of `Pari/GP` (should be changed soon). Based on the following result:
Assume that $a \in \mathbb{Z}$, that f is holomorphic on $\Re(z) \geq a$, $f(z) = o(\exp(2\pi|\Im(z)|))$ as $|\Im(z)| \to \infty$ in vertical strips of bounded width, and $f(x)$ and its derivatives have constant sign and tend to 0 when $x \to \infty$. Then:

$$\sum_{n \geq a} f(n) = \int_{a-1/2}^{\infty} -i \int_0^{\infty} \frac{f(a-1/2+it) - f(a-1/2-it)}{e^{2\pi t}+1}\, dt\ .$$

Can compute the last integral using the doubly-exponential methods. Interesting in principle, but between three and ten times slower than other methods.

Abel–Plana

This is currently the `sumnum` function of `Pari/GP` (should be changed soon). Based on the following result:
Assume that $a \in \mathbb{Z}$, that f is holomorphic on $\Re(z) \geq a$, $f(z) = o(\exp(2\pi|\Im(z)|))$ as $|\Im(z)| \to \infty$ in vertical strips of bounded width, and $f(x)$ and its derivatives have constant sign and tend to 0 when $x \to \infty$. Then:

$$\sum_{n \geq a} f(n) = \int_{a-1/2}^{\infty} -i \int_{0}^{\infty} \frac{f(a-1/2+it) - f(a-1/2-it)}{e^{2\pi t}+1}\, dt\ .$$

Can compute the last integral using the doubly-exponential methods. Interesting in principle, but between three and ten times slower than other methods.

Discrete Euler–MacLaurin: Theory I

As we have seen, the main problem with Euler–MacLaurin is the difficulty of computing the derivatives $f^{(2j-1)}(N)$. The discrete forms avoids this problem. Let d be a small positive real number, and set $\Delta_d(f)(x) = (f(x+d) - f(x-d))/(2d)$, which is an approximation to the derivative. First note that it is trivial to iterate:

$$\Delta_d^k(f)(x) = \frac{1}{(2d)^k} \sum_{0 \le j \le k} (-1)^{k-j} \binom{k}{j} f(x - (k-2j)d) \; .$$

Second, note that the asymptotic series $T = \sum_{j \ge 1} (-1)^j (B_j/j!) f^{(j-1)}(N)$ which occurs in Euler–MacLaurin can be written $T = W(f)(N)$, where W is the differential operator

Discrete Euler–MacLaurin: Theory I

As we have seen, the main problem with Euler–MacLaurin is the difficulty of computing the derivatives $f^{(2j-1)}(N)$. The discrete forms avoids this problem. Let d be a small positive real number, and set $\Delta_d(f)(x) = (f(x+d) - f(x-d))/(2d)$, which is an approximation to the derivative. First note that it is trivial to iterate:

$$\Delta_d^k(f)(x) = \frac{1}{(2d)^k} \sum_{0 \le j \le k} (-1)^{k-j} \binom{k}{j} f(x - (k-2j)d) \ .$$

Second, note that the asymptotic series $T = \sum_{j \ge 1} (-1)^j (B_j/j!) f^{(j-1)}(N)$ which occurs in Euler–MacLaurin can be written $T = W(f)(N)$, where W is the differential operator

Discrete Euler–MacLaurin: Theory II

$$W = \sum_{j\geq 1}(-1)^j \frac{B_j}{j!} D^{j-1} = \frac{1}{e^D - 1} - \frac{1}{D} + 1 \ ,$$

where $D = d/dx$.
Thus, if $\delta = D + \sum_{i\geq 2} a_i D^i$ is some formal differential operator, we can compute the reverse power series $D = \phi(\delta) := \delta + \sum_{i\geq 2} a'_i \delta^i$, and replace this in W so as to obtain

$$W = \frac{1}{e^{\phi(\delta)} - 1} - \frac{1}{\phi(\delta)} + 1 = \sum_{j\geq 1}(-1)^j \frac{b_j(\delta)}{j!} \delta^{j-1}$$

for some new coefficients b_j attached to the operator δ.

Discrete Euler–MacLaurin: Implementation

In the case $\delta = \Delta_d$, to avoid catastrophic cancellation we write for k even:

$$\sum_{j=2}^{k}(-1)^j \frac{b_j(\Delta_d)}{j!}\Delta_d^{j-1}(f)(N) = \sum_{m=-k/2+1}^{k/2}(-1)^{m+1}c_{j,m}f(N-(2m-1)d)$$

$$c_{j,m} = \sum_{\max(m,1-m)\le j\le k/2}(-1)^j\binom{2j-1}{j-m}\frac{b_{2j}(\Delta_d)}{(2d)^{2j-1}(2j)!}\,.$$

Analysis and experiment suggest choosing $d = 1/4$, $N = 0.873D$, and $k = 1.049D$, even. The results are spectacular: it is the most robust of all methods, and apart from Gauss–Monien summation when applicable, it is also the fastest: between 0.3 and 30 seconds for $D = 1000$ depending on the difficulty of computing the summand. Thus, method of choice.

Discrete Euler–MacLaurin: Implementation

In the case $\delta = \Delta_d$, to avoid catastrophic cancellation we write for k even:

$$\sum_{j=2}^{k}(-1)^j \frac{b_j(\Delta_d)}{j!}\Delta_d^{j-1}(f)(N) = \sum_{m=-k/2+1}^{k/2}(-1)^{m+1}c_{j,m}f(N-(2m-1)d)$$

$$c_{j,m} = \sum_{\max(m,1-m)\le j\le k/2}(-1)^j\binom{2j-1}{j-m}\frac{b_{2j}(\Delta_d)}{(2d)^{2j-1}(2j)!}\,.$$

Analysis and experiment suggest choosing $d = 1/4$, $N = 0.873D$, and $k = 1.049D$, even. The results are spectacular: it is the most robust of all methods, and apart from Gauss–Monien summation when applicable, it is also the fastest: between 0.3 and 30 seconds for $D = 1000$ depending on the difficulty of computing the summand. Thus, method of choice.

Gauss–Monien Summation: Theory I

Basic ideas of this method:
First idea: use Gaussian integration with respect to the scalar product

$$\langle f, g \rangle = \sum_{n \geq 1} \frac{f(1/n)g(1/n)}{n^2} ,$$

equivalent to giving a suitable measure: using the general Gaussian procedures, compute the moments of the measure (here $\zeta(k+2)$), then the coefficients of the three-term recursion by a Cholesky decomposition, then the orthogonal polynomials P_n, the nodes as the roots of P_N, and finally the weights by the formula given above. Quite slow initialization, although as usual afterwards the computation is very fast.

Gauss–Monien Summation: Theory II

Second idea: to find the orthogonal polynomials P_n, instead of doing the above, use continued fraction approximations to the kernel function, here $\Phi(z) = \sum_{n\geq 1}(1/(n-z) - 1/n)$ (equal to $\psi(1-z)+\gamma$ with ψ the logarithmic derivative of the gamma function, but we do not need this). Write

$$\Phi(z) = \sum_{n\geq 1}(1/(n-z)-1/n) = c(0)z/(1+c(1)z/(1+c(2)z/(1+\cdots)))\ ,$$

and let as usual $p_n(z)/q_n(z)$ be the nth convergent of the continued fraction. We let $P_n(z) = p_{2n+1}(z)$ and $Q_n(z) = q_{2n+1}(z)$, polynomials of degree n. Then the nodes x_i of Gaussian summation of order N are the roots x_i of Q_N (real, simple, greater or equal to 1, and in fact the first $N/2$ are very close to 1, 2, 3,...), and the weights w_i are simply given by $w_i = P_n(x_i)/Q_n'(x_i)$.

Gauss–Monien Summation: Theory II

Second idea: to find the orthogonal polynomials P_n, instead of doing the above, use continued fraction approximations to the kernel function, here $\Phi(z) = \sum_{n\geq 1}(1/(n-z) - 1/n)$ (equal to $\psi(1-z)+\gamma$ with ψ the logarithmic derivative of the gamma function, but we do not need this). Write

$$\Phi(z) = \sum_{n\geq 1}(1/(n-z)-1/n) = c(0)z/(1+c(1)z/(1+c(2)z/(1+\cdots)))\ ,$$

and let as usual $p_n(z)/q_n(z)$ be the nth convergent of the continued fraction. We let $P_n(z) = p_{2n+1}(z)$ and $Q_n(z) = q_{2n+1}(z)$, polynomials of degree n. Then the nodes x_i of Gaussian summation of order N are the roots x_i of Q_N (real, simple, greater or equal to 1, and in fact the first $N/2$ are very close to 1, 2, 3,...), and the weights w_i are simply given by $w_i = P_n(x_i)/Q_n'(x_i)$.

Gauss–Monien Summation: Implementation

Three aspects:

1. To convert a power series into a continued fraction, we use the quotient-difference algorithm, interesting in itself: construct recursively a triangular array from the coefficients of the series, and read off the coefficients of the continued fraction from its diagonal. However, quite unstable, so need to double or triple the working accuracy.
2. To compute the roots of Q_N, we use fundamentally the fact that the first half are very close to 1, 2, 3,...: direct Newton iteration on those, and then compute the last half with polroots.
3. Analysis shows that a good choice of N is $N = D\log(10)/(\log(D\log(10)) - 1)$, and to work at accuracy $2D$ to partly compensate for the instability of the quotient-difference algorithm.

Gauss–Monien Summation: Implementation

Three aspects:

1. To convert a power series into a continued fraction, we use the quotient-difference algorithm, interesting in itself: construct recursively a triangular array from the coefficients of the series, and read off the coefficients of the continued fraction from its diagonal. However, quite unstable, so need to double or triple the working accuracy.
2. To compute the roots of Q_N, we use fundamentally the fact that the first half are very close to 1, 2, 3,...: direct Newton iteration on those, and then compute the last half with polroots.
3. Analysis shows that a good choice of N is $N = D\log(10)/(\log(D\log(10)) - 1)$, and to work at accuracy $2D$ to partly compensate for the instability of the quotient-difference algorithm.

Gauss–Monien Summation: Implementation

Three aspects:

1. To convert a power series into a continued fraction, we use the quotient-difference algorithm, interesting in itself: construct recursively a triangular array from the coefficients of the series, and read off the coefficients of the continued fraction from its diagonal. However, quite unstable, so need to double or triple the working accuracy.
2. To compute the roots of Q_N, we use fundamentally the fact that the first half are very close to 1, 2, 3,...: direct Newton iteration on those, and then compute the last half with polroots.
3. Analysis shows that a good choice of N is $N = D\log(10)/(\log(D\log(10)) - 1)$, and to work at accuracy $2D$ to partly compensate for the instability of the quotient-difference algorithm.

Gauss–Monien Summation: Results

As usual with Gaussian methods, orders of magnitude faster than anything else after initialization, when applicable. For a spectacular example: $S = \sum_{n \geq 1} \log(\Gamma(1 + 1/n))/n$ for $D = 1000$ is computed in 4.8 seconds by this method (initialization requires 18 seconds), while Euler–MacLaurin needs 8 minutes, and discrete Euler–MacLaurin needs 2 minutes.

Generalized Gauss–Monien Summation

This method is made for sums $\sum_{n\geq a} f(n)$ where f has the asymptotic expansion $f(n) = a_2/n^2 + a_3/n^3 + \cdots$. Easy to generalize to the case $f(n) = \sum_{m\geq 2} a_m/n^{\alpha(m-1)+\beta}$, and also to sums of the form $\sum_{n\geq a} w(n)f(n)$ for some regular weight function $w(n)$. Only the initialization step becomes longer, as in all Gaussian methods, the summation itself is very fast.

Summation with Positive Terms: Conclusion

The conclusion is clear: whenever possible, use Gauss–Monien summation. Otherwise, use discrete Euler–MacLaurin.

Extrapolation Methods: Compendium

Here, we let $u(n)$ be a sequence tending to a limit u as $n \to \infty$ in a regular manner, and we want to compute u. The main point is that $u(n)$ is a priori only defined for $n \in \mathbb{Z}_{\geq 0}$. Seven methods studied:

1. Use of the `sumpos` summation program.
2. Use of an integer version of Gauss–Monien summation.
3. Standard Richardson extrapolation.
4. Richardson $3/2$-2 extrapolation.
5. Richardson 2-3 extrapolation.
6. Lagrange interpolation.
7. Extrapolation using Zagier's method.

Extrapolation Methods: Compendium

Here, we let $u(n)$ be a sequence tending to a limit u as $n \to \infty$ in a regular manner, and we want to compute u. The main point is that $u(n)$ is a priori only defined for $n \in \mathbb{Z}_{\geq 0}$. Seven methods studied:

1. Use of the `sumpos` summation program.
2. Use of an integer version of Gauss–Monien summation.
3. Standard Richardson extrapolation.
4. Richardson $3/2$-2 extrapolation.
5. Richardson 2-3 extrapolation.
6. Lagrange interpolation.
7. Extrapolation using Zagier's method.

Use of Summation Programs

We can of course write $u = u(0) + \sum_{n \geq 0}(u(n+1) - u(n))$ and use a summation program: however, only the built-in sumpos program uses only values of $u(n)$ for integers n. In addition, sumpos is applicable only to very special sequences.

One can modify Gauss–Monien summation to also have this restriction: recall that one-half of the nodes x_i are very close to integers 1, 2, 3, etc... Idea: take as new nodes the closest integer to all the x_i, and compute the corresponding weights using a linear system.

This works, and gives a reasonable method which is orders of magnitude faster than the above, and applicable to most sequences. It is usually a little slower than Lagrange-based methods, but it is the fastest when higher transcendental functions need to be computed, such as $u(n) = \log(\Gamma(n)) - ((n-1/2)\log(n) - n)$.

Use of Summation Programs

We can of course write $u = u(0) + \sum_{n \geq 0}(u(n+1) - u(n))$ and use a summation program: however, only the built-in sumpos program uses only values of $u(n)$ for integers n. In addition, sumpos is applicable only to very special sequences.

One can modify Gauss–Monien summation to also have this restriction: recall that one-half of the nodes x_i are very close to integers 1, 2, 3, etc... Idea: take as new nodes the closest integer to all the x_i, and compute the corresponding weights using a linear system.

This works, and gives a reasonable method which is orders of magnitude faster than the above, and applicable to most sequences. It is usually a little slower than Lagrange-based methods, but it is the fastest when higher transcendental functions need to be computed, such as $u(n) = \log(\Gamma(n)) - ((n - 1/2)\log(n) - n)$.

Richardson Methods: the Basic Method

We have explained in the numerical integration section how Richardson extrapolation works: if for instance $u(n) = u + a/n^k + O(1/n^{k+1})$ converges to u as $1/n^k$, then $(2^k u(2n) - u(n))/(2^k - 1) = u + O(1/n^{k+1})$ will converge to u faster, as $1/n^{k+1}$, and we apply this idea recursively.

The main problem with this method is that it requires the computation of $u(2^m n)$ for large m. This is not a difficulty in double-precision computations, but is usually prohibitive in multiprecision. It can only be used if there is a "formula" for $u(n)$, say if $u(n)$ can be computed in time polynomial in $\log(n)$. In those cases, this basic Richardson method is almost always the best available.

Richardson Methods: the Basic Method

We have explained in the numerical integration section how Richardson extrapolation works: if for instance $u(n) = u + a/n^k + O(1/n^{k+1})$ converges to u as $1/n^k$, then $(2^k u(2n) - u(n))/(2^k - 1) = u + O(1/n^{k+1})$ will converge to u faster, as $1/n^{k+1}$, and we apply this idea recursively.

The main problem with this method is that it requires the computation of $u(2^m n)$ for large m. This is not a difficulty in double-precision computations, but is usually prohibitive in multiprecision. It can only be used if there is a "formula" for $u(n)$, say if $u(n)$ can be computed in time polynomial in $\log(n)$. In those cases, this basic Richardson method is almost always the best available.

Richardson Methods: Modified Methods

To attenuate the 2^m explosion, we can modify the basic method in several ways. First, by choosing n divisible by some large power of 2, say 2^a, so that we can first perform a Richardson-type acceleration with factor $3/2$ instead of 2 a times, and the rest with the usual factor 2.

A second method which is always much better is to use instead of the sequence $M(m) = 2^m$ the sequence $M(2m-1) = 2^m$ and $M(2m) = 3 \cdot 2^m$, by using after the first doubling the single auxilliary sequence $(3^k u(3n) - u(n))/(3^k - 1)$ for a suitable k. The conclusion is surprising: when $u(n)$ can be computed in time polynomial in $\log(n)$ (a reasonable polynomial of course), the best extrapolation method is basic Richardson. Otherwise, non-Richardson methods are always much faster, so the modified Richardson methods, although theoretically better, are completely useless!

Richardson Methods: Modified Methods

To attenuate the 2^m explosion, we can modify the basic method in several ways. First, by choosing n divisible by some large power of 2, say 2^a, so that we can first perform a Richardson-type acceleration with factor $3/2$ instead of 2 a times, and the rest with the usual factor 2.

A second method which is always much better is to use instead of the sequence $M(m) = 2^m$ the sequence $M(2m-1) = 2^m$ and $M(2m) = 3 \cdot 2^m$, by using after the first doubling the single auxilliary sequence $(3^k u(3n) - u(n))/(3^k - 1)$ for a suitable k.

The conclusion is surprising: when $u(n)$ can be computed in time polynomial in $\log(n)$ (a reasonable polynomial of course), the best extrapolation method is basic Richardson. Otherwise, non-Richardson methods are always much faster, so the modified Richardson methods, although theoretically better, are completely useless!

Richardson Methods: Modified Methods

To attenuate the 2^m explosion, we can modify the basic method in several ways. First, by choosing n divisible by some large power of 2, say 2^a, so that we can first perform a Richardson-type acceleration with factor $3/2$ instead of 2 a times, and the rest with the usual factor 2.

A second method which is always much better is to use instead of the sequence $M(m) = 2^m$ the sequence $M(2m-1) = 2^m$ and $M(2m) = 3 \cdot 2^m$, by using after the first doubling the single auxilliary sequence $(3^k u(3n) - u(n))/(3^k - 1)$ for a suitable k.

The conclusion is surprising: when $u(n)$ can be computed in time polynomial in $\log(n)$ (a reasonable polynomial of course), the best extrapolation method is basic Richardson. Otherwise, non-Richardson methods are always much faster, so the modified Richardson methods, although theoretically better, are completely useless!

Lagrange Interpolation

We have also already seen this in the context of numerical integration: If for instance $u(n) = u + \sum_{m \geq 1} a_m / n^{\alpha m}$, we look for a function $f(x)$ such that $u(n) = f(1/n^{\alpha})$, so that $u = f(0)$. The function f can be obtained by simple Lagrange interpolation, and the formula for $f(0)$ is particularly simple when $\alpha = 1$ or 2, the most frequent cases.

Analysis and experiments show that for $\alpha = 1$ one should choose a polynomial of degree $N = 1.1D$, and work at accuracy $5D/3$, while for $\alpha = 2$ we choose $N = 0.9D$ and accuracy $4D/3$. Similar constants can be computed for general α.

This is the fastest method for extrapolation, but is not always robust, i.e., it may fail in a number of cases.

Lagrange Interpolation

We have also already seen this in the context of numerical integration: If for instance $u(n) = u + \sum_{m \geq 1} a_m / n^{\alpha m}$, we look for a function $f(x)$ such that $u(n) = f(1/n^{\alpha})$, so that $u = f(0)$. The function f can be obtained by simple Lagrange interpolation, and the formula for $f(0)$ is particularly simple when $\alpha = 1$ or 2, the most frequent cases.
Analysis and experiments show that for $\alpha = 1$ one should choose a polynomial of degree $N = 1.1D$, and work at accuracy $5D/3$, while for $\alpha = 2$ we choose $N = 0.9D$ and accuracy $4D/3$. Similar constants can be computed for general α.
This is the fastest method for extrapolation, but is not always robust, i.e., it may fail in a number of cases.

Zagier Extrapolation: Theory

Essentially the same as Lagrange, but first presented differently, and second applied to the sequence $u(rn)$ for some r with $20 \le r \le 100$, say. Not faster, but much more robust. Idea: assume for simplicity $u(n) = u + \sum_{m \ge 1} a_m/n^m$. Choose suitable k and set $v(n) = n^k u(rn)$. If Δ is the forward difference operator $\Delta(w)(n) = w(n+1) - w(n)$, we have

$$\Delta^k(v)(n)/k! = u + \sum_{m \ge k+1} b_m/n^m$$

for other coefficients b_m, so much faster convergence to u. Of course we use

$$\Delta^k(v)(n) = \sum_{0 \le j \le k} (-1)^{k-j} \binom{k}{j} v(n+j) \; .$$

Zagier Extrapolation: Implementation

Need to choose r, k, and the working accuracy. Good choices for r are $r = 20$, 30,...,100, and the corresponding values of k are $k = z(r)D$, with $z(20) = 0.48$, $z(30) = 0.43$,... $z(100) = 0.35$. Using a working accuracy of $5D/4 + 9$ is sufficient.

This method can be slower or faster than pure Lagrange interpolation (which essentially corresponds to $r = 1$), but is much more robust, so should be used instead if time is not essential. Roughly: for sequences which take a polynomial time in $\log(n)$ to compute, it is usually faster (but when applicable, basic Richardson is even faster in that case), and for other sequences Lagrange interpolation is much faster.

Zagier Extrapolation: Implementation

Need to choose r, k, and the working accuracy. Good choices for r are $r = 20$, 30,...,100, and the corresponding values of k are $k = z(r)D$, with $z(20) = 0.48$, $z(30) = 0.43$,... $z(100) = 0.35$. Using a working accuracy of $5D/4 + 9$ is sufficient.

This method can be slower or faster than pure Lagrange interpolation (which essentially corresponds to $r = 1$), but is much more robust, so should be used instead if time is not essential. Roughly: for sequences which take a polynomial time in $\log(n)$ to compute, it is usually faster (but when applicable, basic Richardson is even faster in that case), and for other sequences Lagrange interpolation is much faster.

Extrapolation: Conclusion

Except in very special cases, use Lagrange or Zagier extrapolation, both being good methods. Zagier extrapolation being much more robust, if one wants to compute asymptotic expansions, one uses Zagier extrapolation recursively, and in this way it is easy to compute 20 or so coefficients of usual expansions, such as Stirling or similar.

Thank you for your attention !