

Arbitrary precision numerical methods with online computation

Alasdair McAndrew

Melbourne, Australia
amca01@gmail.com

ABSTRACT

Texts on numerical methods usually present the same sets of methods: for solving equations there are a few bracketing methods, then the secant and Newton's method; for integration we have the Newton-Cotes rules, maybe Romberg integration, possibly Gaussian integration and a couple of others. There are, however, many more methods than these, often just as simple, but with faster convergence. It is hard however, to demonstrate to students—particularly pre-service teachers—the power of these methods when using only limited precision arithmetic, which is all that is available on their standard tools. Experiments with the software Pari/GP and high precision provided powerful evidence of the speed of convergence, in a way that theoretical discussions could never manage. The students thus experienced some profound mathematics from an experimental and experiential perspective. This article explores some of those methods, as well as several others, in an invitation to experimental mathematics.

1 Introduction

The author has spent many years teaching numerical methods to pre-service teachers: students studying a teaching degree with the hope of teaching mathematics at a senior high school level. These students were, in general, poorly prepared mathematically, and the course devolved into a sort of “recipe book” of numerical techniques; short on theoretical underpinnings, long on examples and general discussion. Most of the time we used the spreadsheet MS Excel (including writing macros), and Geogebra. These are tools that the students are very likely to use in their own classrooms, and one object of the course was to encourage the students to become familiar with them both, and to reach a standard of competency in their use. However, these tools are inadequate for discussing fast converging techniques, such as Newton's method for solving equations, or even faster methods, such as Ostrowski's method. For this we needed a system which could cope with arbitrary precision arithmetic. After some experimentation we chose Pari/GP [1], which can be used as a powerful calculator, with arbitrary precision. It also has the advantage of having an online version, so students can use it without the difficulty of downloading and installing it themselves. We were this able to replace the theory of numerical analysis with experimentation, and provide the students with insight which would otherwise be unavailable to them.

We used the software mainly as a demonstration tool, although students were encouraged to experiment with it themselves. At the end of the course, the final assessment required the students to give a very short talk on some technique not covered in the course, describe it and

explain its advantages and disadvantages. A few students chose techniques which could be best explained using Pari/GP.

This article then, has several aims:

- to demonstrate convergence rates by examples using high precision arithmetic,
- to explore some techniques not normally covered in an elementary numerical methods course,
- encourage readers to explore some of these techniques themselves.

2 Root finding, and rates of convergence

2.1 A note on rates of convergence

Suppose that x_n is a sequence of values approaching a root r of an equation. We say that the sequence has a rate of convergence p if

$$\lim_{k \rightarrow \infty} \frac{|x_k - r|}{|x_{k-1} - r|^p} = L$$

where L is a non-zero positive value. This can be written as

$$|x_k - r| \sim L |x_{k-1} - r|^p.$$

for k sufficiently large. Since each value $x_k - r$ (for k large enough), is close to zero, this means that if $p > 1$, then each value is closer to zero than the previous value.

This can be described in terms of correct decimal places. Suppose that

$$|x_k - r| \sim R_k 10^{-d_k}.$$

where $1 \leq R_k < 10$ and $d_k - 1$ is the number of correct decimal places. Note that as k increases, the difference of 1 becomes negligible. Then from the previous equation,

$$R_k 10^{d_k} \sim L (R_{k-1} 10^{-d_{k-1}})^p,$$

and so, taking logarithms of each side, we end up with something like

$$d_k \sim A + p d_{k-1}$$

where A is a value based on L , R_k , p and their logarithms. We can assume that as k increases, A becomes constant. This means that the values of d_k (which measure the number of correct decimal places), will grow such that the value of A becomes irrelevant, and so

$$d_k \sim p d_{k-1}.$$

This means that the rate of convergence being p is equivalent to the number of correct decimal places increasing by a factor of p for each iteration (and of course for k sufficiently large).

2.2 Newton's method

Newton's method, also known as the Newton-Raphson Method, is a technique for solving non-linear equations, using a linear approximation to the function. Using the first two terms of Taylor's series:

$$f(x) = f(x_0) + f'(x_0)(x - x_0)$$

(where the equals sign is understood to be an approximation; the remainder being left out) and solving for x the right hand side set equal to zero:

$$f(x_0) + f'(x_0)(x - x_0) = 0 \Rightarrow x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

This leads to a sequence x_n of values approximating a root of $f(x)$ defined by

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

and x_0 chosen to be close to the root. Newton's method is known to be *quadratically convergent*; in simple terms this means that the number of decimal places roughly doubles at each iteration.

Suppose we wish to solve $x^5 + x^2 - 9 = 0$. By sketching the graphs $y = x^5$ and $y = 9 - x^2$ we see that there's a solution round about $x = 1,5$. Using double precision, such as might be seen on an old calculator, or in a spreadsheet, produces the first column in Table 1. Using 15 decimal places, such as can be produced using Geogebra and its `IterationList` command, is shown in the second column.

<u>Double precision</u>	<u>15 significant figures</u>
1.5	1.5
1.4701986755	1.470198675496689
1.4690476497	1.469047649682589
1.4690459950	1.469045995027990
1.4690459950	1.469045995024576
1.4690459950	1.469045995024576

Table 1: Newton's method results with two different levels of precision

Both columns in the table indicate that Newton's method (when it works, which it usually does, although that is not a guarantee), works extremely fast. But to see just *how* fast, more decimal places are needed.

Enter Pari/GP. We can write the following:

```
\p100
f = x -> x^5 + x^2 - 9
df = x -> deriv(f)(x)
nr = x -> x - f(x)/df(x)
x = 1.4
{ for(n = 1, 8,
  x = nr(x);
  print(x) ) }
```

The first line sets the displayed precision to 100 significant figures (the internal working precision is automatically set higher), and the result is:

```
1.475507088331515812431842966194111232279171210468920392584514721919302071973827699018538713195201745
1.469097660806927730303152317012975437713828752758646871516141005401506834779826503570097119531040660
1.469045998353145655570724943231892160714578403528249919752637060906909298903961425262191703652659191
1.469045995024576141982419144195810413451545904099355870091271818717365304438440317345097012648716081
1.469045995024576128166072373416532549952275577629562632854788222095049316523292628381340066586439612
1.469045995024576128166072373416532311904344983394630164831191409059432672558871047954625079014700660
```

1.469045995024576128166072373416532311904344983394630164831191409059362007165403992829078650424926775
1.469045995024576128166072373416532311904344983394630164831191409059362007165403992829078650424926775

We have shown the correct figures in red, to give an idea of the speed of convergence. However, rather than display the results, we can display instead the successive differences between the results, which also gives a good idea of the speed of convergence:

Script for Newton's method	Successive differences
<code>\p500</code>	0.02980132450
<code>f = x -> x^5 + x^2 - 9</code>	0.001151025814
<code>df = x -> deriv(f)(x)</code>	1.654654598 e-6
<code>nr = x -> x - f(x)/df(x)</code>	3.414236573 e-12
<code>x = 1.5</code>	1.453667835 e-23
<code>{ for(n = 1, 8,</code>	2.635168087 e-46
<code> y = nr(x);</code>	8.659535670 e-92
<code> printf("%.10g\n",abs(y-x));</code>	9.351196254 e-183
<code> x = y) }</code>	1.090466439 e-364

Table 2: Newton's method showing successive differences

Immediately we can see that the number of correct figures basically doubles at each step. The nice thing about this approach is that we can increase the precision, say replacing the first line above with `\p50000` and performing 16 iterations. The first nine outputs are of course the same as in Table 2 but after that we obtain:

1.482868257 e-728
2.742098794 e-1456
9.376573368 e-2912
1.096393048 e-5822
1.499030629 e-11644
2.802199000 e-23288
9.792101539 e-46576

If we divide successive numbers of correct figures (given in each place as the negative of the exponent, less one), we have:

[2.000, 2.500, 2.200, 2.000, 2.045, 2.022, 2.000, 1.995, 2.003, 2.001, 2.001, 2.000, 2.000, 2.000, 2.000]

This is a powerful demonstration of the quadratic convergence of Newton's rule.

All of the above can be done with a bit of extra programming: simply create a vector to hold the values of correct figures, and then divide successive values.

```
\p20000
N = 16
v = vector(16)
f(x) = x^5 + x - 1
df(x) = 5*x^4 + 1
nr(x) = x - f(x)/df(x)
x = 0.7
{ for(n = 1, N,
  y = nr(x);
  a = abs(y-x);
  printf("%.10g\n",a);
  if(a > 0, t = truncate(log(a)/log(0.1)));
  v[n] = t;
```

```
x = y) }
print(v)
```

The vector produced by this is

```
[1, 2, 5, 11, 22, 45, 91, 182, 363, 727, 1455, 2911, 5821, 11643, 23287, 46575]
```

and successive quotients can be produced by:

```
{ for(n = 1, length(v)-2,
  printf("%.5g, ", v[n+1]/v[n]+0.0) ) }
```

and this produces the same list of values seen earlier.

2.3 The secant method

The same method can be applied to investigate the rate of convergence of other methods. For example, the well known *secant method* takes two successive approximations x_{k-1} and x_k to the root, and finds the x -intercept of the line between the points $(x_{k-1}, f(x_{k-1}))$ and $(x_k, f(x_k))$. This intercept is the next approximation, so that

$$x_{k+1} = x_k - f(x_k) \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}.$$

The program is very similar to that for Newton's method:

```
\p20000
f(x) = x^5 + x^2 - 9
v = vector(1)
x = 1.4; y = 1.6;
{
while(abs(x - y) > 10.0^-20000,
z = y - f(y)*(y - x)/(f(y) - f(x));
a = abs(z - y);
if(a > 0, t = truncate(log(a)/log(0.1)); v = concat(v,[t]));
x = y; y = z)
print(v)

N = length(v)
v2 = vector(N-3);
{ for(i = 1, N-1, v2[i] = v[k+1]/v[k]+0.0) }
printf("%.5g",v2)
```

The final 12 values for v and v2 are

```
80, 130, 211, 342, 553, 895, 1448, 2343, 3792, 6135, 9927, 16063
```

and

```
1.6327, 1.6250, 1.6231, 1.6209, 1.6170, 1.6184, 1.6179, 1.6181, 1.6184, 1.6179, 1.6181, 1.6181
```

The final value here should be recognizable as the “golden ratio” $(\sqrt{5} + 1)/2$, which appears to be the rate of convergence. This can be shown analytically. For simplicity, write $\epsilon_k = |x_k - r|$, so from Section 2.1 the rate of convergence is p if $\epsilon_k \propto \epsilon_{k-1}^p$. For the secant method, it can be shown that $\epsilon_{k+1} \propto \epsilon_k \epsilon_{k-1}$. This can be written as $\epsilon_k^p \propto \epsilon_{k-1}^p \epsilon_{k-1} = \epsilon_{k-1}^{p+1}$. Thus $\epsilon_k \propto \epsilon_{k-1}^{(p+1)/p}$. And this can be written as $\epsilon_{k-1}^p \propto \epsilon_{k-1}^{(p+1)/p}$. Equating powers produces the equation $p^2 = p + 1$ for which the golden ratio is a root.

2.4 Ostrowski's method

This method [2] was developed by the mathematician Alexander Markovich Ostrowski, and published in a monograph on numerical solutions of equations. It is a Newton's method with a correction. It converges even faster, as demonstrated below. It works in two steps: the first computes a Newton iteration; the second applies a "correction" to it:

$$y_n = x_n - \frac{f(x_n)}{f'(x_n)}$$
$$x_{n+1} = y_n - f(y_n) \frac{x_n - y_n}{f(x_n) - 2f(y_n)}$$

The computation is very similar to before, but note the very high precision:

```
\p500000
v = vector(1)
f(x) = x^5 + x^2 - 9
df = x -> deriv(f)(x)
x = 1.5; a = 1.0
{
while(a > 10.0^-500000,
y = x - f(x)/df(x); z = y - f(y)*(x - y)/(f(x) - 2*f(y));
a = abs(z - x); /* printf("%.10g\n",a); */
if(a > 0, t = truncate(log(a)/log(0.1)); v = concat(v,[t]));
x = z)}
print(v)
```

The vector of successive quotients can be computed using exactly the same commands as previously. The two vectors (v and $v2$) are seen to be:

```
[0, 1, 6, 24, 97, 391, 1564, 6258, 25034, 100136, 400545]
[6.0000, 4.0000, 4.0417, 4.0309, 4.0000, 4.0013, 4.0003, 4.0000, 4.0000]
```

Here the number of correct figures *quadruples* at each stage; demonstrating that this method has *quartic convergence*.

It's worthwhile here to reflect on the relative efficiency of these methods. Newton's method requires one function computation and one derivative computation at each step. Ostrowski's method requires two function computations and one derivative computation. Newton's method takes two steps to achieve what Ostrowski's method does in one step. But two steps of Newton's method would require two derivative computations. Note that in our little script above for Ostrowski's method, we have computed $fx = f(x)$ and $fy = f(y)$ before using them. This decreases the amount of computation; in a single iteration we thus compute each of $f(x)$, $f(y)$, $f'(x)$ once only.

One way of comparing two methods is by an *efficiency index*, which is computed as $n^{1/k}$, where n is the convergence rate, and k is the number of evaluations of either the function or its derivatives. Newton's method thus has an efficiency index of $2^{1/2} \approx 1.414$ and Ostrowski's method has an efficiency index of $4^{1/3} \approx 1.587$.

2.5 Illinois methods

These are so-called because they seem to have first been described in an internal memo in the Computer Science Department of the University of Illinois in the 1950s [3]. The algorithm was

“coded by” J. N. Snyder, but it’s not known whether Snyder was the creator of this method, or only the person who implemented the method in code. The code is given without any analysis or examples. An early article describing and analysing them was published in 1971[4]. Later, that work was extended further[5]. They are versions of the “method of false position”, or *regula falsi*. This method uses the same computation as the secant method, except that successive iterations are always chosen to be on either side of the root; that is $f(x_k)f(x_{k+1}) < 0$. This is an example of a “bracketing method”, and is thus guaranteed to converge. But it can converge very slowly if f is concave at the root. In this case the iterations tend to creep to the root from one side only, as shown in Figure 1.

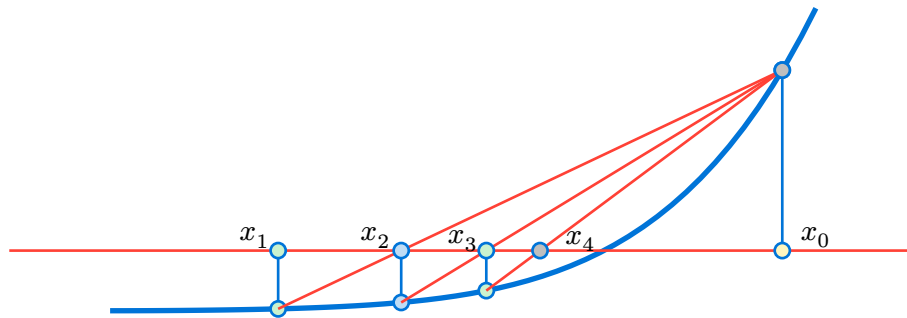


Figure 1: False position converging slowly from one side only

The Illinois method makes a very simple adjustment: if two successive iterations are on the same side of the root, the next iteration will use *half* the function value. In other words, from the values x_{k-1} and x_k on either side of the root, we compute x_{k+1} using the secant computation, Then, using the abbreviation f_k for $f(x_k)$:

- (i) If $f_k f_{k+1} < 0$, replace (x_{k-1}, f_{k-1}) with (x_k, f_k) .
- (ii) If $f_k f_{k+1} > 0$, replace (x_{k-1}, f_{k-1}) with $(x_{k-1}, f_{k-1}/2)$.

The first choice simply says that if the new value is on the other side of the root from the most recent value, we continue as usual. Otherwise, we halve the previous function value and go again.

The multiplier 1/2 has been analysed, and variants proposed: the “Pegasus method” involves multiplying by

$$\frac{f_k}{f_k + f_{k+1}}.$$

Ford[5] describes a number of different methods, the most promising of which is due to Anderson and Björck[6], with a multiplier defined by::

$$\text{if } f_k > f_{k+1} \text{ then use } 1 - \frac{f_{k+1}}{f_k} \text{ else use } 0.5.$$

The GP code looks like this:

```
\p10000
N = 25
v = vector(N);
f = x -> x^5 + x^2 - 9
a = 1.4; b = 1.6;
fa = f(a); fb = f(b);
{for(n = 1, N,
```

```

c = b - fb*(b - a)/(fb - fa);
fc = f(c);
if(fb*fc < 0, a = b; fa = fb);
if(fb*fc > 0, fa = fa/2);
b = c; fb = fc;
d = abs(b - a); printf("%0.10g\n", d);
if(d > 0, t = truncate(log(d)/log(0.1)); v[n] = t)
)
}

```

and the successive differences have the following numbers of unchanged digits:

[0, 0, 2, 2, 2, 8, 8, 8, 24, 25, 25, 75, 75, 75, 226, 226, 226, 680, 680, 680, 2041, 2041, 2041, 6124, 6124]

Because sometimes a value is unchanged at an iteration—this is where we halve a previous function value by 2—the sequence is not strictly increasing. As can be seen, the values seem to come in groups of three. If we take the first value in each (non-zero) group:

[2, 8, 24, 75, 226, 680, 2041, 6124]

then the successive quotients can be found to be

[4.000, 3.000, 3.125, 3.013, 3.009, 3.001, 3.000]

we see that in each subsequence, successive values roughly triple. We can infer from this that in the Illinois method, three iterations triple the number of correct digits, giving a convergence rate of $3^{\frac{1}{3}} \approx 1.44225$. At any rate, the Illinois method has *superlinear* convergence, which gives it a huge advantage over false position. Note that applying the method of false position to this problem produces only 9 correct digits after 24 iterations.

3 Numerical Integration (“Quadrature”)

Up until recently, there was little need for arbitrary precision quadrature, but this has changed. One important application is to determine if an unknown integral can be expressed analytically in terms of known constants and elementary functions; for this a very high precision may be needed. Then the integral can be determined analytically. For example, Bailey et al[7] showed that

$$\int_0^1 \frac{t^2 \log(t)}{(t^2 - 1)(t^4 + 1)} dt = \pi^2(2 - \sqrt{2})/32$$

when modern commercial CASs (at that time) could solve the integral, but with long and cumbersome expressions.

3.1 Gaussian quadrature

This is a method defined on the interval $[-1, 1]$, as

$$\int_{-1}^1 f(x) dx \approx \sum_{k=1}^n w_k f(x_k)$$

where x_k are values chosen from within the interval (known as *nodes* or *abscissae*), and w_k are the *weights*. The value n is the *order* of the rule. An integral over an arbitrary finite interval can be transformed to be over $[-1, 1]$ by a simple linear transformation:

$$\int_a^b f(x) dx = \frac{b+a}{2} \int_{-1}^1 f\left(\frac{b-a}{2}x + \frac{b+a}{2}\right) dx.$$

The nodes and weights are determined by a set of polynomials called the *Legendre polynomials* $P_n(x)$. They can be defined in a number of ways; first by their orthogonality:

$$\int_{-1}^1 P_n(x)P_m(x) dx = 0 \text{ if } m \neq n, \text{ and } P_k(1) = 1 \text{ for all } k.$$

Or by a *Rodrigues formula* (named for the French mathematician Olinde Rodrigues):

$$P_k(x) = \frac{1}{2^k k!} \frac{d^k}{dx^k} (x^2 - 1)^k.$$

Or by a recursive formula:

$$P_0(x) = 1, P_1(x) = x, (k+1)P_{k+1}(x) = (2k+1)xP_k(x) - kP_{k-1}(x) \text{ for } k \geq 1.$$

The first few polynomials are

$$P_0(x) = 1$$

$$P_1(x) = x$$

$$P_2(x) = (3x^2 - 1)/2$$

$$P_3(x) = (5x^3 - 3x)/2$$

$$P_4(x) = (35x^4 - 30x^2 + 3)/8$$

The Legendre polynomials have a huge number of powerful properties, but for the purposes of quadrature one of them is that the k -th order polynomial $P_k(x)$ has k distinct real roots in the open interval $(-1, 1)$, and these roots are symmetric about 0. (This last can be easily inferred from the nature of the polynomials: those of even order are even functions, and those of odd order are odd functions.) Gaussian quadrature, or more properly *Gauss-Legendre* quadrature, has nodes x_i which are the zeroes of the Legendre polynomial, and weights defined by

$$w_i = \frac{2}{(x_i - 1)^2 P_k'(x_i)^2} = \frac{-2}{(n+1)P_k'(x_j)P_{n+1}(x_j)}, \text{ with } P_n'(x) = \frac{n(xP_n(x) - P_{n-1}(x))}{x^2 - 1}.$$

The expression for the derivative is based on the generating function for Legendre polynomials, and can be used to obtain the second equality. This is easier to compute than the first expression for the weights.

(There is another method of computing the weights, which involves computing the eigenvalues of a tridiagonal matrix, but it is not very much more efficient.) There are a number of computational approaches to calculating the nodes; one way is to first approximate them by a simple formula, and then use Newton's rule (for example) to find each root at an appropriate precision. To find the k -th root of the n -th order polynomial $x_{n,k}$, one such method[8] is first to define

$$\theta_{n,k} = \frac{n - k + 3/4}{n + 1/2} \pi$$

and then

$$x_{n,k} = \cos\left(\theta_{n,k} + \frac{1}{16(n+1/2)^2} \cot\left(\frac{1}{2}\theta_{n,k}\right) - \tan\left(\frac{1}{2}\theta_{n,k}\right)\right) + O(n^{-4})$$

(possibly this formula is not “simple”). The efficient computation to arbitrary precision of Legendre roots, and weights, is an active area of research.

For example, let’s compute *Catalan’s constant*, defined as

$$G = \frac{1}{1^2} - \frac{1}{3^2} + \frac{1}{5^2} - \frac{1}{7^2} + \frac{1}{9^2} - \frac{1}{11^2} + \dots$$

It is not known whether G is rational or irrational, or if it is irrational, whether it is transcendental. It is considered to be the most basic constant for which the rationality is unknown.

There are a vast number of integral representations of G , for example:

$$G = \frac{1}{2} \int_0^{\frac{\pi}{2}} \frac{t}{\sin t} dt = \frac{\pi^2}{32} \int_{-1}^1 \frac{t+1}{\sin(\frac{\pi}{4}(t+1))} dt$$

PARI/GP includes the constant `Catalan` which can be used to check the quadrature computations. There is also the command `intnumgaussinit` which takes an integer parameter N , and returns the positive nodes and associated weights for Gauss-Legendre quadrature of order N .

We use the original definition for the nodes and weights, and double the number of nodes each time, and take successive differences. Since the nodes are symmetric, as are the weights, we only have to consider the positive nodes, which reduces the amount of work, and the amount of storage:

```
\p1500
f = x -> Pi^2/32*(x+1)/sin(Pi/4*(x+1))
N = 2
pN = pollegendre(N);
dN = deriv(pN);
inds = polrootsreal(pN,[0,+oo]);
wts = apply(z -> 2/((1 - z^2)*subst(dN,'x,z)^2),inds);
s = sum(i=1,N/2, (f(inds[i]) + f(inds[i]))*wts[i]);
{
for(k=2,11,
N = N*2;
pN = pollegendre(N);
dN = deriv(pN);
inds = polrootsreal(pN);
wts = apply(z -> 2/((1 - z^2)*subst(dN,'x,z)^2),inds);
t = sum(i=1,N, f(inds[i])*wts[i]);
printf("%4d, %.10g\n",N,abs(s-t));
s = t)
}
```

which (slowly) produces the output

```
4, 0.001373466013
8, 1.207898661 e-6
16, 9.256070662 e-13
32, 5.298153506 e-25
```

```

64, 1.697919893 e-49
128, 1.723405586 e-98
256, 1.764823572 e-196
512, 1.845005320 e-392
1024, 2.013353810 e-784

```

At this stage we might run into a memory problem. But we are using a highly inefficient method. If we replace the four lines

```

pN = pollegendre(N);
dN = deriv(pN);
inds = polrootsreal(pN,[0,+oo]);
wts = apply(z -> 2/((1 - z^2)*subst(dN,'x,z)^2),inds);

```

with

```
[inds,wts] = intnumgaussinit(N)
```

not only do we get much faster computation, but we can push the computation further, to obtain the next few differences:

```

1024, 2.013353810 e-784
2048, 3.697360633 e-1501

```

One problem is that the nodes have to be computed separately for each new computation. Unlike other quadrature methods, the nodes of one order do not form a subset of nodes of a higher order.

3.2 Tanh-Sinh quadrature

This is a method dating to 1974 [9], and is one of a family called “double exponential methods”. It takes a function $f(x)$ defined over the interval $[-1, 1]$, and applies to it the transformation (which can be shown to be optimal):

$$x = g(t) = \tanh\left(\frac{\pi}{2} \sinh(t)\right)$$

and which transforms the integral into one over $(-\infty, \infty)$:

$$\int_{-1}^1 f(x) dx = \int_{-\infty}^{\infty} f(g(t))g'(t) dt.$$

We note that as $x \rightarrow \infty$, $g(x) \rightarrow 1$. The function

$$g'(t) = \frac{\frac{\pi}{2} \cosh t}{\cosh^2\left(\frac{\pi}{2} \sinh t\right)}$$

decays away very quickly—for example, $g'(10) \approx 4.42162 \times 10^{-15022}$. It is this fast decay, of the order of $\exp(-\exp(x))$, which gives to these methods the “double exponential” description.

Rather than go through the theory, we can look at an integral to compute the *Euler-Mascheroni constant* γ , defined as

$$\gamma = \lim_{n \rightarrow \infty} \sum_{k=1}^n \frac{1}{k} - \log(n).$$

Its value is roughly 0.5772156649, and like Catalan’s constant, it is not known whether it is irrational (though all the evidence points to it); or, if it is irrational, whether it is

transcendental. It is another basic constant about which very little is known. One of many integral representations is

$$-\int_0^1 \log\left(\log\left(\frac{1}{x}\right)\right) dx = -\frac{1}{2} \int_{-1}^1 \log\left(\log\left(\frac{2}{x+1}\right)\right) dx.$$

This is an integral on which Gauss-Legendre quadrature performs very badly, mainly because of the asymptote at the right hand end. Even using 1000 points, the result is only accurate to 6 decimal places.

Here's how we might obtain an initial approximation:

```
\p4000
N = 16; h = 0.5;
inds = vector(N,k,k*h); // The indices are evenly spaced
xs = apply(z -> g(z), inds); // Apply g to the indices for the nodes
ws = apply(z -> gd(z), inds); // Apply g' to the indices for the weight
ts = h*dg(0)*f(g(0));
{ for(i = 1,N,
  ts = ts + h*ws[i]*(f(xs[i])+f(-xs[i])) // Nodes and weights are symmetric
)}
```

Note that $Nh = 8$, and $(g'(Nh) \approx 2.497 \times 10^{-2030})$, which is the limiting accuracy we can get with these values. If we want greater accuracy, we can increase N or the initial h ; for example using $N = 16$ and an initial $h = 0.6$ gives $g'(Nh) \approx 2.0277 \times 10^{-10068}$

This produces (at 10 decimal places) 0.5772109267, which is correct to 5 decimal places. And note that we only used 16 points. Note also the very high precision. This is to ensure that the function doesn't return an error trying to compute $\log(0)$.

We can now at each stage double the number of nodes N and halve the distance h . Since the values kh are evenly spaced, we only have to compute the new "in between" values at each stage. This greatly reduces computation.

```
{ for(j = 2,10,
  h = h/2;
  inds = vector(N,k,h+2*(k-1)*h);
  xs = apply(z -> g(z), inds); ws = apply(z -> gd(z), inds);
  tsn = ts/2 + h*sum(i = 1,N,ws[i]*(f(xs[i]) + f(-xs[i])));
  printf("%2d: %14.8e %14.8e\n",j,abs(ts - tsn),abs(tsn + psi(1)));
  ts = tsn;
  N = N*2) }
```

PARI/GP doesn't contain γ as a constant, but it does contain the Ψ function, defined as the logarithmic derivative of the gamma function:

$$\Psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

for which $\Psi(1) = -\gamma$. The output from the above script contains the absolute difference between successive computations, and also the difference between a computation and $-\Psi(1)$:

```
2: 4.73820925 e-6      5.68938150 e-14
3: 5.68938150 e-14    1.71898280 e-30
4: 1.71898280 e-30    4.11107494 e-64
5: 4.11107494 e-64    4.85331231 e-132
```

6:	4.85331231 e-132	1.40776633 e-268
7:	1.40776633 e-268	2.79875331 e-542
8:	2.79875331 e-542	2.64627995 e-1090
9:	2.64627995 e-1090	1.21022945 e-2033
10:	5.82539155 e-2032	5.94641450 e-2032

At this stage 1024 points have been used, to obtain over 2000 place accuracy. Note that the accuracy roughly doubles for each iteration, when we double the number of points. This demonstrates the power of the tanh-sinh method. See [7] for a deeper analysis.

(Using starting values of $N = 16, h = 0.6$, a precision of 11000 decimal places, and 12 iterations gets an accuracy of about $7.71569836 \times 10^{-10071}$ - that is, about 10070 decimal places¹.)

3.3 Clenshaw-Curtis quadrature

A standard quadrature method is *Newton-Cotes integration*, in which n equally spaced points are chosen over the integration interval $[a, b]$, and the interpolating polynomial is used to approximate the integral. Thus

$$\int_a^b f(x) dx \approx \int_a^b p(x) dx$$

where $p(x)$ is the interpolating polynomial through the points $(x_i, f(x_i))$, with $x_i = a + i(b - a)/n$, for i ranging from 0 to n . The trouble is that high order polynomials through equally spaced points are known to have extremely poor fits especially towards the endpoints. For this reason high order Newton-Cotes rules are never used; instead the interval is subdivided into smaller intervals and a lower order rule is used on each subinterval instead. The poor fit with high order polynomials is known as *Runge's phenomenon* and is illustrated on the left in Figure 2 with 12 equally spaced points and the function (known as "Runge's function") $y = 1/(1 + 25x^2)$.

However, if we choose instead of equally spaced points, the points $x_i = \cos((i + 1/2)\pi/n)$ and interpolate through those, we get a much closer fit to the function, as shown on the right in Figure 2.

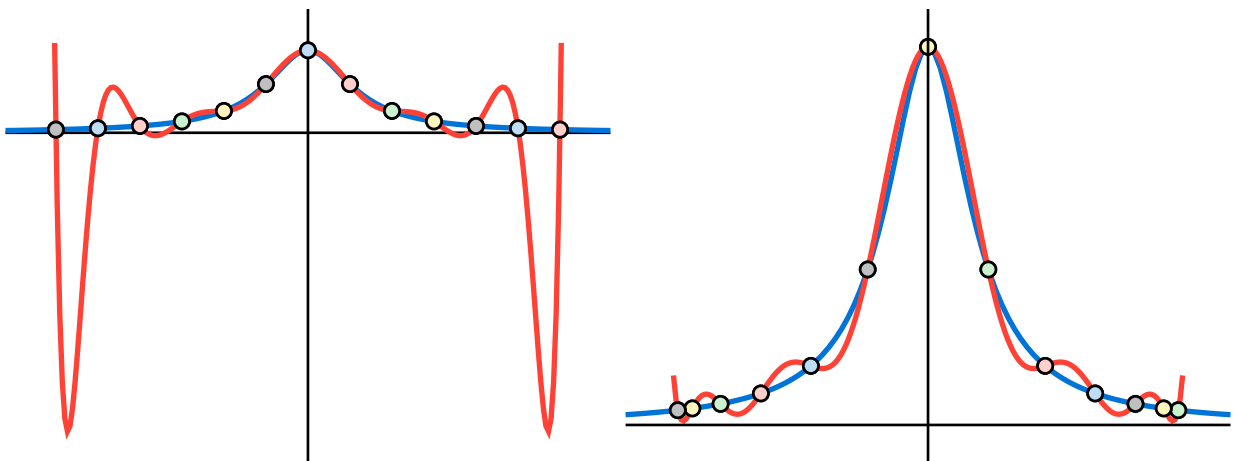


Figure 2: Interpolation at differently spaced points

The idea of integrating through the polynomial interpolating at cosine values is the basis of *Clenshaw-Curtis quadrature*, which is considered to have equivalent accuracy to Gauss-Legendre

¹And takes over an hour and 20 minutes on the author's laptop.

quadrature[10], but with much simpler weights. There are in fact three slightly different rules here, all of which work similarly. Two predate Clenshaw and Curtis (who published their rule in 1960), and go back to the work of Lipot Fejér in the 1930s. All of them use cosine values over the interval $[-1, 1]$; some use the endpoints, see Figure 3. Note that each rule, because of its cosine values, has nodes which are projections of equally spaced points around the unit half-circle, and are thus more clustered near the endpoints. We will refer to the use of any of these rules as Clenshaw-Curtis quadrature, although we will restrict ourselves to Fejér's first rule.

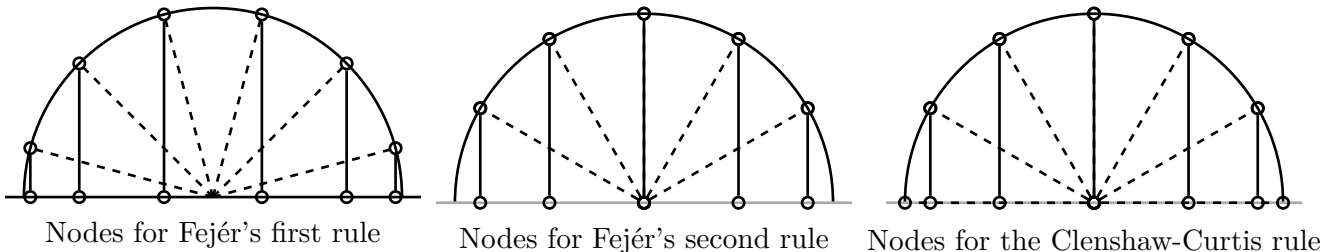


Figure 3: Different nodes for different rules

The weights can be obtained either by integrating the interpolating polynomial, or by use of the discrete fourier (or cosine) transform. However, a simple way is to compute the weights so that the integration rule is exact for all polynomials of degree up to the number of points.

But first we need another set of orthogonal polynomials; the *Chebyshev polynomials* (formally the *Chebyshev polynomials of the first kind*) $T_k(x)$ which can be defined by the recursion:

$$T_0(x) = 1, T_1(x) = x, T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x) \quad \text{for } k \geq 1$$

or by the simple trigonometric definition

$$T_k(\cos \theta) = \cos(k\theta).$$

This works because by de Moivre's theorem, the real part of $(\cos \theta + i \sin \theta)^k$ consists of powers of $\cos \theta$ and even powers of $\sin \theta$. But we can write $\sin^{2m} \theta$ as $(1 - \cos^2 \theta)^m$ which means that the real part can be written entirely in powers of $\cos \theta$. For example, with $k = 5$:

$$\begin{aligned} \cos(5\theta) &= \operatorname{Re}((\cos \theta + i \sin \theta)^5) \\ &= \cos^5 \theta - 10 \cos^3 \theta \sin^2 \theta + 5 \cos \theta \sin^4 \theta \\ &= \cos^5 \theta - \cos^3 \theta (1 - \cos^2 \theta) + \cos \theta (1 - \cos^2 \theta)^2 \\ &= 16 \cos^5 \theta - 20 \cos^3 \theta + 5 \cos \theta \end{aligned}$$

Thus $T_5(x) = 16x^5 - 20x^3 + 5x$. The Chebyshev polynomials are extremely useful in approximation; they are also orthogonal over the interval $[-1, 1]$ with respect to the weight function $(1 - x^2)^{-1/2}$:

$$\int_{-1}^1 \frac{1}{\sqrt{1-x^2}} T_m(x) T_n(x) dx = 0 \quad \text{if } m \neq n.$$

They also provide a basis for the vector space of polynomials, so that any power x^n can be written in terms of Chebyshev polynomials as shown in Table 3

k	Chebyshev polynomials $T_k(x)$	Expansion of x^k
0	1	T_0
1	x	T_1
2	$2x^2 - 1$	$(T_2 + T_0)/2$
3	$4x^3 - 3x$	$(T_3 + 3T_1)/4$
4	$8x^4 - 8x^2 + 1$	$(T_4 + 4T_2 + 3T_0)/8$
5	$16x^5 - 20x^3 + 5x$	$(T_5 + 5T_3 + 10T_1)/16$
6	$32x^6 - 48x^4 + 18x^2 - 1$	$(T_6 + 6T_4 + 15T_2 + 10T_0)/32$

Table 3: Chebyshev polynomials

It can be seen that if n is even, then

$$x^n = \binom{n}{0}T_n + \binom{n}{1}T_{n-2} + \dots + \frac{1}{2}\binom{n}{n/2}T_0$$

and if n is odd, then

$$x^n = \binom{n}{0}T_n + \binom{n}{1}T_{n-2} + \dots + \binom{n}{(n-1)/2}T_1.$$

It can be easily shown that the roots of $T_n(x)$ are the values

$$\cos\left(\left(k - \frac{1}{2}\right)\frac{\pi}{n}\right) \text{ for } 1 \leq k \leq n.$$

These are called *Chebyshev nodes* and are the nodes used in Fejér's first rule. In fact we will explore this rule rather than the Clenshaw-Curtis rule; this has the advantage of not including the endpoints, so can deal with integrals with endpoint singularities. We want to choose the weights so that the resulting quadrature rule is accurate for all polynomials of degree up to and including n . But since the Chebyshev polynomials form a basis, this is equivalent to the quadrature rule being accurate for all Chebyshev polynomials of order up to n . That is, for nodes x_k and weights w_k we want

$$w_1T_m(x_1) + w_2T_m(x_2) + \dots + w_nT_m(x_n) = \int_{-1}^1 T_m(x) dx$$

for all $m = 0, 2, \dots, n-1$. The right hand side is equal to 0 for odd m (since Chebyshev polynomials of odd order are odd), and is equal to 2 for $m = 0$. For even $m = 2q$ for $q \geq 1$ then the integral is equal to $-2/(4q^2 - 1)$.

The nodes are chosen to be $x_k = \cos\left(\left(k - \frac{1}{2}\right)\frac{\pi}{n}\right)$, and since by definition $T_m(\cos \theta) = \cos(m\theta)$, we have

$$T_m(x_k) = T_m\left(\cos\left(\left(k - \frac{1}{2}\right)\frac{\pi}{n}\right)\right) = \cos\left(m\left(k - \frac{1}{2}\right)\frac{\pi}{n}\right).$$

For indices starting at 1 (as PARI/GP does), instead of at 0, this means the weights \mathbf{w} can be determined [11] by the matrix equation

$$\mathbf{V}\mathbf{w} = \mathbf{b}$$

where \mathbf{V} is the matrix defined by

$$V_{ij} = \cos\left((i-1)\left(j - \frac{1}{2}\right)\frac{\pi}{n}\right) \text{ for } 1 \leq i, j \leq n$$

and the elements of \mathbf{b} are the integrals of the Chebyshev polynomials over the interval $[-1, 1]$; from above these are:

$$\left[2, 0, -\frac{2}{3}, 0, -\frac{2}{15}, 0, -\frac{2}{35}, 0, -\frac{2}{63}, \dots\right]$$

Here's an example, applied to the integral for Catalan's constant:

```
\p100
f = x -> Pi^2/32*(x+1)/sin(Pi/4*(x+1))
N = 99
xs = vector(N,k,cos((k-1/2)*Pi/N));
V = matrix(N,N,i,j,cos((i-1)*(j-1/2)*Pi/N));
b = vector(N); b[1] = 2
{ for(k = 1, (N-1)/2,
b[2*k+1] = -2/(4*k^2-1)) }
w = matsolve(V,b~);
s = sum(i = 1,N,f(xs[i])*w[i]);
printf("%.10g\n",abs(s - Catalan))
```

This produces the output

```
1.301500596 e-80
```

which shows a very high accuracy. Increasing the values of N and the precision will result in even closer approximations. Note that this method of computing the weights is not very efficient; better methods use the discrete Fourier transform [12] or the discrete cosine transform [13]. But for simple experimentation, using matrix arithmetic is quite sufficient.

4 Conclusions

We have shown how to compute roots and integrals to a very high accuracy using relatively simple means, and the software PARI/GP. We note that the great advantages of PARI/GP are its accessibility, and its ability to manage arbitrary precision arithmetic. In the numerical methods class taught by the author, students could choose to copy and paste scripts, and to experiment with these scripts, to explore computations to arbitrary precision, or to explore rates of convergence. Students whose analytical skills might be insufficient for understanding the formal definition and proofs of rates of convergence, or of the formal relative accuracies of different quadrature methods, could obtain a real insight by experimentation and exploration. This insight provided a deep appreciation of some modern mathematical techniques, and in the author's opinion allowed the students to think of themselves in a very real sense as being genuine mathematicians.

Acknowledgements

The author gratefully acknowledges the reviewers, who pointed a number of typos, errors and inconsistencies. The author has attempted to make all corrections, amendments, and additions required.

References

- [1] The PARI~Group, “PARI/GP version 2.17.2,” PARI/GP. [Online]. Available: <http://pari.math.u-bordeaux.fr/>
- [2] A. M. Ostrowski, “Appendix G, Some Modifications and Improvements of the Newton-Raphson Method,” in *Solutions of Equations in Euclidean and Banach Spaces*, 3rd ed., Academic Press, 1973.
- [3] J. N. Snyder, “Inverse interpolation, a real root of $f(x)=0$,” University of Illinois Digital Computer Laboratory, ILLIAC I Library Routine H1-71 4, 1956.
- [4] Mark Dowell and Peter Jarratt, “A modified regula falsi method for computing the root of an equation,” *BIT Numerical Mathematics*, pp. 168–174, 1971.
- [5] J. A. Ford, “Improved algorithms of Illinois-type for the numerical solution of nonlinear equations,” University of Essex, Department of Computer Science, 1995.
- [6] N. Anderson and A. Björck, “A new high-order method of Regula Falsi type for computing the root of an equation,” *BIT Numerical Mathematics*, pp. 253–264, 1973.
- [7] David H. Bailey, Karthik Jeyabalan, and Xiaoye S. Li, “A Comparison of Three High-Precision Quadrature Schemes,” *Experimental Mathematics*, vol. 14, no. 3, pp. 317–329, 2005.
- [8] L. Gatteschi and G. Pittaluga, “An asymptotic expansion for the zeros of Jacobi polynomials,” *Mathematical Analysis*, vol. 79, pp. 70–86, 1985.
- [9] Hidetosi Takahasi and Masatake Mori, “Double Exponential Formulas for Numerical Integration,” *Publications of the Research Institute for Mathematical Sciences, Kyoto University*, vol. 9, no. 3, pp. 721–741, 1974, [Online]. Available: https://www.jstage.jst.go.jp/article/kyotoms1969/9/3/9_3_721/_pdf
- [10] Lloyd N. Trefethen, “Is Gauss quadrature better than Clenshaw-Curtis?,” *SIAM Review*, vol. 50, no. 1, pp. 67–87, 2008, [Online]. Available: https://people.maths.ox.ac.uk/trefethen/publication/PDF/2008_127.pdf
- [11] Jörg Waldvogel, “Fast Construction of the Fejér and Clenshaw-Curtis Quadrature Rules, Revisited,” in *Chebfun and Beyond*, 2012. [Online]. Available: <https://people.math.ethz.ch/~joergw/Papers/oxford.pdf>
- [12] Jörg Waldvogel, “Fast Construction of the Fejér and Clenshaw-Curtis Quadrature Rules,” *BIT Numerical Mathematics*, vol. 43, no. 1, pp. 1–18, 2003.
- [13] Alvise Sommariva, “Fast construction of Fejér and Clenshaw–Curtis rules for general weight functions,” *Computers & Mathematics with Applications*, vol. 65, no. 4, pp. 682–693, 2013.