

INTRODUCTION

Here are a few interesting, and sometimes difficult to find, formulas, algorithms and constants that I have found useful over a long career of designing and writing computer systems. Some of these will likely be familiar to you, some may not. All have been thoroughly tested and work correctly.

The latest version of this document, source code in various languages for the included algorithms, and much more, can be downloaded free from my website below.

I would be happy to receive comments on this material, especially any new formulas or algorithms! There is an email link to me on my website below. Enjoy.

Judson D. McClendon
Sun Valley Systems
4522 Shadow Ridge Pkwy
Pinson, AL 35126

sunvaley.com

Newton's Method for Finding Roots:

Newton's Method for extracting positive integral roots of positive real numbers is as follows:

The r_{th} root of the positive number n is obtained as a root of the function:

$$f(a) = a^r - n$$

by means of the iterated equation:

$$\begin{aligned} a_j &= a_i - (a_i^r - n) / (r * a_i^{(r-1)}) \\ &= a_i * (1 - 1 / r) + (n / (r * a_i^{(r-1)})) \end{aligned}$$

For square root, the equation simplifies to

$$a_j = (a_i + (n / a_i)) / 2$$

Where: a_i = previous approximation
 a_j = next approximation
 r = root
 n = number

To use Newton's Method, begin by obtaining a first approximation to the root. For example, to get the square root of very high precision (long) numbers, you can divide the exponent of the number by 2 to get the exponent of the first approximation, then use 3 as the first digit of the mantissa (e.g. if $n = 5.0 \times 10^{24}$ then 1st app. = 3.0×10^{12} ; if $n = 5.0 \times 10^{23}$ then 1st app. = 3.0×10^{11} ; if $n = 5.0 \times 10^{-4}$ then 1st app. = 3.0×10^{-2}). This gives a first approximation reasonably close to the square root. Then iterate the equation above until the desired accuracy is obtained. You can compare successive approximations to determine when to stop, or simply iterate a pre-determined number of times. The precision (number of accurate digits) roughly doubles each iteration, so Newton's method converges very rapidly if a reasonable first approximation is obtained. What else would we expect from Newton?

Integer Power Function:

This method calculates y^x (y is a real number, x is a positive integer) using the Square-Multiply algorithm. This algorithm is extremely efficient, much faster than using logarithms. I used this algorithm in my extended precision calculator program BIGCALC (sunvaley.com).

```
if (x = 0)
  p = 1
  exit
```

```
if (x < 0)
  x = -x
  r = true
```

```
shift x left 1 bit
```

```
set rightmost bit of x to 1 (flag bit)
```

```
shift x left until high order bit is 1
```

```
shift x left again
```

```
set p = y
```

```
while x <> (high order bit = 1, remaining bits = 0)
```

```
  square p (p = p * p)
```

```
  if high order bit of x = 1
```

```
    multiply y times p (p = p * y)
```

```
  shift x left 1 bit
```

```
if (r = true)
  p = 1 / p
```

Where: y = number
 x = integer power
 r = reciprocal Boolean, for negative powers
 p = working variable, will contain y^x on completion

If y is real, then p should be real, to sufficient precision.

Number N is base B to what power P?

$$P = \log(N) / \log(B) \quad (\text{logarithms in any base may be used})$$

Derivation:

$$B^P = N$$

$$\log(B^P) = \log(N)$$

$$\log(B)P = \log(N)$$

$$P = \log(N) / \log(B)$$

Example: What power of 2 is 32,768?

$$P = \log(32768) / \log(2)$$

$$P = 4.51545 / 0.30103 \quad (\text{base 10 logs})$$

$$P = 15$$

Weekday from Date - Zeller's Congruence:

Derived by Julius Christian Johannes Zeller in 1883, Zeller's Congruence can be used to determine the day of the week of any date in the Gregorian calendar (started October 15, 1582, but was adopted at various times in different places). Below are two variants, (A) using real variables, and (B) using integer variables and the “\” integer division operator (B).

$$A: w = (\text{int}((13 * m + 3) / 5) + d + y + \text{int}(y / 4) - \text{int}(y / 100) + \text{int}(y / 400) + 1) \bmod 7$$

$$B: w = ((13 * m + 3) \setminus 5 + d + y + y \setminus 4 - y \setminus 100 + y \setminus 400 + 1) \bmod 7$$

Where: w = weekday (0 = Sun, 1 = Mon, 2 = Tue, 3 = Wed, 4 = Thu, 5 = Fri, 6 = Sat)
d = day of the month
m = month of year (Jan & Feb = 13 & 14 of previous year)
y = year (year - 1 if month is Jan or Feb)
\ = Integer division operator (e.g. $4 \setminus 3 = 1$)
int = integer part (e.g. $\text{int}(3.5) = 3$)
mod = modulus or remainder part (e.g. $10 \bmod 7 = 3$)

Examples Feb 12, 1809: d = 12, m = 14, y = 1808 (w = 0 Sun)

Jul 4, 1776: d = 4, m = 7, y = 1776 (w = 4 Thu)

Date Day Number:

Date day number calculates a number which is one greater for each succeeding date. The date day number can be used to determine the number of days between any two dates in the Gregorian calendar (started October 15, 1582, but was adopted at various times in different places). Note that 16 bit integer variables are not sufficient; use form (A) with real variables => 6 digits, or form (B) with minimum 32 bit integer variables and the “\” integer division operator.

$$A: n = y * 365 + \text{int}(y / 4) - \text{int}(y / 100) + \text{int}(y / 400) + \text{int}(m * 30.6001) + d$$

$$B: n = y * 365 + y \setminus 4 - y \setminus 100 + y \setminus 400 + m * 306001 \setminus 10000 + d$$

Where: n = day number (Note that day number can be 6 digits or more)
 d = day of the month
 m = month + 13 (Jan & Feb)
 month + 1 (Mar - Dec)
 y = year - 1 (Jan & Feb)
 year (Mar - Dec)
 \ = Integer division operator (e.g. 4 \ 3 = 1)
 int = integer part (e.g. int(3.5) = 3)

Example: Feb 12,1809: d = 12, m = 15, y = 1808 (n = 660,829)

Jul 4, 1776: d = 4, m = 8, y = 1776 (n = 648,919)

days between = 11,910

Note: The result can be adjusted by a constant to make the date day number sequence begin on any date desired. For example, subtract 578,607 to make January 1, 1584 day zero:

$$A: n = y * 365 + \text{int}(y / 4) - \text{int}(y / 100) + \text{int}(y / 400) + \text{int}(m * 30.6001) + d - 578607$$

$$B: n = y * 365 + y \setminus 4 - y \setminus 100 + y \setminus 400 + m * 306001 \setminus 10000 + d - 578607$$

This is a good choice, because January 1, 1584 was a Sunday, making the following true:

$$\text{DayOfWeek} = \text{DateDayNbr}(\text{date}) \text{ MOD } 7$$

Where DayOfWeek = 0 for Sunday, 1 for Monday, ..., 6 for Saturday.

And 1584 was a leap year less than 2 years after the start of the Gregorian Calendar.

Gregorian Easter Date Computation:

Computation of the date of Easter (sometimes called “Computus”, Latin for “computation”), is somewhat complex. It is also unusual, in that the calculation makes use of the “calendar moon”, which does not exactly correspond to the celestial moon. Below are four algorithms for calculating Easter. All of the algorithms have been tested and verified to return the same dates for Easter throughout the range of years from 1583 (the first Easter in the Gregorian Calendar) to well beyond 9999, which is more than adequate for any practical purpose. (Though of mathematical curiosity only, using 64 bit integer variables they are consistent to beyond 10^{10} .)

All these algorithms use integer variables, and the operator ‘\’ indicates an integer divide. If you use the Lilius-Clavius algorithm for years > 9000, you should read the notes about the epact calculation and “mod” operator. For the next few millennia, 16 bit integers are sufficient for these algorithms. But for years > ~6,000, they can overflow 16 bit integers. It’s very unlikely the Gregorian Calendar will remain unchanged beyond 4000, because it will differ a whole day from the celestial year by then, and a “4000 year multiple” leap year correction has already been proposed (i.e. leap years would be years evenly divisible by 4, but not 100, unless 400, but not 4000). Also, the Earth’s rotation is continually slowing due to tidal friction, and this will continue to prompt further calendrical corrections in the far future. Further, a number of proposals have been made to change the Easter date to match the true lunar cycle, and to simplify the calculation in various ways.

The algorithms below are ordered by date created/published.

Lilius-Clavius Easter Algorithm:

The following algorithm, by Aloysius Lilius and Christopher Clavius from the late 16th century, with notes to the right, comes from volume one of “The Art of Computer Programming”, Second Edition, by Donald E. Knuth, pages 155-156. Let y be the year for which the date of Easter is desired.

$g = (y \bmod 19) + 1$	(g is the so-called “golden number” of the year in the 19-year Metonic cycle.)
$c = \text{int}(y / 100) + 1$	(c is the century number.)
$x = \text{int}(3 * c / 4) - 12$	(x is the number of years, such as 1900, in which leap year was dropped to keep in step with the sun.)
$z = \text{int}((8 * c + 5) / 25) - 5$	(z is a special correction designed to synchronize Easter with the moon's orbit.)
$d = \text{int}(5 * y / 4) - x - 10$	(d is a factor to adjust the date to the following Sunday.)
$e = (11 * g + 20 + z - x) \bmod 30$ if ($e = 25$ and $g > 11$) or ($e = 24$) $e = e + 1$	(e is the so-called “epact” which specifies when a full moon occurs. For years in the distant future, see *note below.) (Easter is supposedly “the first Sunday following the first full moon which occurs on or after March 21.” Actually

Formulas 2.8 - by Judson D. McClendon – 2008-09-21

$n = 44 - e$
 if ($n < 21$)
 $n = n + 30$

perturbations in the moon's orbit make this not strictly true, but we are concerned here with the “calendar moon” rather than the actual moon. The n'th of March is a calendar full moon.)

$n = n + 7 - ((d + n) \bmod 7)$ (Advance to following Sunday.)

if ($n > 31$) (If $n > 31$ then Easter falls in April instead of March.)
 EasterMonth = 4
 EasterDay = $n - 31$
 else
 EasterMonth = 3
 EasterDay = n

*Note: As Knuth points out, beginning with year 9006, the expression for Epact can become negative, and many computer language built-in “mod” operators do not yield the desired result for this purpose. Below are three possible solutions to this problem:

Knuth’s solution (based on analysis of his intentionally obfuscated MIX assembly solution on pages 511-513) is to calculate the Epact as above then, if it is negative, add 30:

$e = (11 * g + 20 + z - x) \bmod 30$
 if ($e < 0$)
 $e = e + 30$

My own solution, which is simple and has zero performance cost, is to increase the constant ‘20’ in the Epact calculation by a multiple of 30 (and thus ‘invisible’ to the ‘mod 30’ operation) sufficient to prevent the Epact going negative. A constant of 50 is sufficient through year 9999, but I prefer using 320 so the original 20 and the 300 multiple of 30 are apparent. Below is a table that should satisfy even the most avid Easter calculators, showing maximum year ranges, minimum Epact values, suggested correction factors, and appropriate replacements for the Epact constant of 20. The minimum Epact values in blue are estimated, based on the pattern for lower limits.

Max Year	Minimum Epact	Correction Factor	Epact Constant
9,999	-5	30 or 300	50 or 320
99,999	-392	600	620
999,999	-4,262	6,000	6020
9,999,999	-42,962	60,000	60020
99,999,999	-429,962	600,000	600020
999,999,999	-4,299,962	6,000,000	6000020
9,999,999,999	-42,999,962	60,000,000	60000020
99,999,999,999	-429,999,000	600,000,000	600000020
999,999,999,999	-4,299,999,000	6,000,000,000	6000000020
9,999,999,999,999	-42,999,999,000	60,000,000,000	60000000020
99,999,999,999,999	-429,999,999,000	600,000,000,000	600000000020
999,999,999,999,999	-4,299,999,999,000	6,000,000,000,000	6000000000020

Another, perhaps more elegant, solution was suggested to me by Edward Moneo, in which the mod function is 'corrected' to produce the desired result, and works for any range of years. There is no single "official" definition of the 'mod' function, but for this purpose, the following definition works:

$$A \bmod B = A - B * \text{int}(A / B)$$

So, instead of coding the epact calculation directly as:

$$E = (11 * G + 20 + Z - X) \bmod 30$$

use this instead:

$$\begin{aligned} \text{temp} &= (11 * G + 20 + Z - X) \\ E &= \text{temp} - 30 * \text{int}(\text{temp} / 30) \end{aligned}$$

Butcher Easter Algorithm:

This Easter algorithm was submitted anonymously by letter to Nature Magazine and published in April 1876. Bishop Samuel Butcher ([hence the name](#)) showed that this algorithm followed from Delambre's analytical solutions and produces the date of Easter for all years. Let y be the year for which the date of Easter is desired.

$$\begin{aligned} a &= y \bmod 19 \\ b &= y \setminus 100 \\ c &= y \bmod 100 \\ d &= b \setminus 4 \\ e &= b \bmod 4 \\ f &= (b + 8) \setminus 25 \\ g &= (b - f + 1) \setminus 3 \\ h &= (19 * a + b - d - g + 15) \bmod 30 \\ i &= c \setminus 4 \\ k &= c \bmod 4 \\ l &= (32 + 2 * e + 2 * i - h - k) \bmod 7 \\ m &= (a + 11 * h + 22 * l) \setminus 451 \\ \text{EasterMonth} &= (h + l - 7 * m + 114) \setminus 31 \\ \text{EasterDay} &= (h + l - 7 * m + 114) \bmod 31 + 1 \end{aligned}$$

Original Oudin Easter Algorithm:

In 1940 J.M. Oudin published this Easter algorithm as a correction to Carl Friedrich Gauss' Easter computation of 1800, which was flawed. Let y be the year for which the date of Easter is desired.

$$\begin{aligned} c &= y \setminus 100 \\ n &= y - 19 * (y \setminus 19) \\ k &= (c - 17) \setminus 25 \\ i &= c - c \setminus 4 - (c - k) \setminus 3 + 19 * n + 15 \end{aligned}$$

$$i = i - 30 * (i \setminus 30)$$

$$i = i - (i \setminus 28) * (1 - (i \setminus 28)) * (29 \setminus (i + 1)) * ((21 - n) \setminus 11)$$

$$j = y + y \setminus 4 + i + 2 - c + c \setminus 4$$

$$j = j - 7 * (j \setminus 7)$$

$$x = i - j$$

$$\text{EasterMonth} = 3 + (x + 40) \setminus 44$$

$$\text{EasterDay} = x + 28 - 31 * (\text{EasterMonth} \setminus 4)$$

Tondering Modification of Oudin Easter Algorithm:

This simplified form of Oudin's algorithm is by [Claus Tondering](#). Easter computation for a full range of years without using a table doesn't get much shorter than this. Let y be the year for which the date of Easter is desired.

$$c = y \setminus 100$$

$$g = y \bmod 19$$

$$h = (c - c \setminus 4 - (8 * c + 13) \setminus 25 + 19 * g + 15) \bmod 30$$

$$i = h - (h \setminus 28) * (1 - (29 \setminus (h + 1))) * ((21 - g) \setminus 11)$$

$$j = (y + y \setminus 4 + i + 2 - c + c \setminus 4) \bmod 7$$

$$k = i - j$$

$$\text{EasterMonth} = 3 + (k + 40) \setminus 44$$

$$\text{EasterDay} = k + 28 - 31 * (\text{EasterMonth} \setminus 4)$$

Where: c = Century Number - 1
 g = Golden Number - 1
 h = (23 - Epact) mod 30
 i = Days from 21 Mar to the Paschal full moon
 j = Weekday for the Paschal full moon (0=Sun, 1=Mon, ..., 6=Sat)
 k = Days from 21 Mar to the Sunday on or before the Paschal full moon (-6 to 28)

Some Important Constants to 100+ digits:

π :	3.14159	26535	89793	23846	26433	83279	50288	41971	69399	37510	
	58209	74944	59230	78164	06286	20899	86280	34825	34211	70679	82
$1/\pi$:	0.31830	98861	83790	67153	77675	26745	02872	40689	19291	48091	
	28974	95334	68811	77935	95268	45307	01802	27605	53250	61719	12
π^2 :	9.86960	44010	89358	61883	44909	99876	15113	53136	99407	24079	
	06264	13349	37622	00448	22419	20524	30017	73403	71855	22318	24
$\pi/180$:	0.01745	32925	19943	29576	92369	07684	88612	71344	28718	88541	
	72545	60971	91440	17100	91146	03449	44368	22415	69634	50948	22
$180/\pi$:	57.29577	95130	82320	87679	81548	14105	17033	24054	72466	56432	
	15491	60243	86120	28471	48321	55263	24409	68995	85111	09441	86
e :	2.71828	18284	59045	23536	02874	71352	66249	77572	47093	69995	
	95749	66967	62772	40766	30353	54759	45713	82178	52516	64274	27
$1/e$:	0.36787	94411	71442	32159	55237	70161	46086	74458	11131	03176	
	78345	07836	80169	74614	95744	89980	33571	47274	34591	96437	46
e^2 :	7.38905	60989	30650	22723	04274	60575	00781	31803	15570	55184	
	73240	87127	82252	25737	96079	05776	33843	12485	07912	17947	73
$\sqrt{2}$:	1.41421	35623	73095	04880	16887	24209	69807	85696	71875	37694	
	80731	76679	73799	07324	78462	10703	88503	87534	32764	15727	35
$\sqrt{3}$:	1.73205	08075	68877	29352	74463	41505	87236	69428	05253	81038	
	06280	55806	97945	19330	16908	80003	70811	46186	75724	85756	75
Φ :	1.61803	39887	49894	84820	45868	34365	63811	77203	09179	80576	
	28621	35448	62270	52604	62818	90244	97072	07204	18939	11374	84

Note: $\Phi = (\sqrt{5} + 1) / 2$ (Golden Ratio)

If you are interested in other constants, or constants to greater precision, you might want to try my BIGCALC program, which was used to derive the constants above. BIGCALC emulates a 'generic' Hewlett-Packard RPN calculator, with a four register stack and 10 memory registers, with precision up to 1075 digits. The constants π and e are provided to full precision, and many others (e.g. Golden Ratio $\Phi = (\sqrt{5} + 1) / 2$) can be easily calculated. BIGCALC and other goodies can be downloaded free from my website: sunvaley.com.