20

Function Approximation

Why Study Function Approximation?

- **Solvability**. Many functions are too complex or too 'expensive' to compute in their original form; approximations offer a way to reexpress them in ways that employ simpler and faster arithmetic.

- **Accuracy vs. Efficiency**. Determining a function approximation involves a balance between precision and speed. There's no single right answer but approximations can be easily adjusted to investigate the trade-offs.

- **Range/Operation Restrictions**. The suitability of an approximation greatly depends on the range of data that it must handle, and upon hardware rounding errors.

- **Polynomial Accuracy**. Polynomial approximations, typified by Taylor series and the Lagrange and Chebyshev approximations, combine simple implementations with precision. The Taylor series works best with data close to its expansion point, and is more error-prone at greater distances. However, Lagrange and Chebyshev spread the approximation error evenly across the entire data range.

- **Modeling Singularities**. Rational approximations are a better choice than polynomials for handling curves containing sharp spikes, vertical asymptotes, or functions that flatten out at infinity.

20.1 Dirty Functions as Motivation

Function approximation investigates how complex functions can be approximated by polynomials. 'Complex' may mean that the original function doesn't have a known definition or that it's computationally costly to evaluate. Polynomials may be 'simpler' in the sense of having desirable properties such as continuity and known error bounds, or by being less expensive to calculate. An excellent textbook which covers this topic is *Numerical Analysis* by Burden *et. al* [**BFB16**]. One major area of function approximation, which we won't be discussing here (but will perhaps in the future) are Fourier series.

The second author's interest in this topic started while he was reading up on the early days of microcomputer BASICs. Many of them lacked math functions, and the first ones only offered integer arithmetic. With the arrival of floating-point capabilities, Dennis Allison (the designer of Tiny BASIC) saw the need for better math support, and published "Quick and Dirty Functions for BASIC" in *Dr. Dobb's Journal* in 1978 [**All78**].

The article briefly describes five short but efficient subroutines for $\sqrt{x}$, $\log_{10} x$, 10^x, $\arctan x$, and $\cos x$. The functions utilize various range reducing formulae, and most employ polynomial approximations borrowed from Hart's *Computer Approximations* text [**Har68**]. The approximations are accurate to around six decimal places, which was more than adequate enough for the microcomputer hardware of the time, and the underlying motivation was that multiplication/-division was expensive, available memory was tiny, and floating-point precision was limited. Low-order approximations or truncated power series were a valid alternative to the high-precision algorithms used on mainframes.

This highlights the question of the relevance of function approximation *to-day*, when computing power is plentiful and cheap. One answer can be seen in the current uses of languages similar to BASIC (e.g. MicroPython `https://micropython.org/`) as coding tools on micro-controllers such as the Arduino and low-end Raspberry Pi's. In addition, functions with no simple closed definition can still only be represented approximately.

Allison's article was republished in Volume 3 of the *Dr. Dobb's Journal* reprints, which is available at the Internet Archive at `https://archive.org/details/dr_dobbs_journal_vol_03/`, in issue No. 9. I've also saved a copy of the article locally in dirtyAllison.pdf.

dirty.py is our translation of Allison's BASIC into Python, with the addition of a test-rig that calls each function with different inputs and compares their results to Python's math library. The results are much as you'd expect – Allison's code is accurate to around 6 or 7 decimal places. In the next four subsections, we'll look in detail at Allison's $\sqrt{x}$, $\log_{10} x$, $\arctan x$, and $\cos x$.

20.2 Dirty Square Root

Allison's pseudo-code:

(1) Report $x < 0$ as an error.

(2) Reduce the input range:
$t = 1; repeat\,\{x = x/100; t = t * 10\}$ until $x \leq 1$.
By the end, $0 \leq x \leq 1$ and $x * t * t =$ the input number.

(3) Make an initial guess for $\sqrt{x}$ with

$$y_0 = 0.115442 + 1.15442x$$

(4) Iterate until $|y_{k+1} - y_k| < 10^{-6}$ using the Newton-Raphson method:

$$y_{k+1} = \tfrac{1}{2}\left(y_k + \frac{x}{y_k}\right)$$

(5) The final answer is: $y_{k+1} * t$

It's possible to replace (4) by a for-loop that loops 4 or 5 times since we know that each pass will roughly double the number of correct digits in the result.

The Python code:

```
def sqrtD(x):
  if x < 0:
    raise ValueError("Math domain error")
  if x == 0:
    return 0

  # Scale x down by powers of 100
  t = 1.0
  while x > 1:
    x = x / 100
    t *= 10

  # A linear guess at sqrt(x)
  y = 0.115442 + 1.15442*x

  # Newton-Raphson until convergence
  count = 0
  while True:
    oldY = y
    y = 0.5 * (y + (x / y))
    count += 1
    if abs(y - oldY) < 1e-6:
      break
  return y * t
```

The *Newton–Raphson* method starts with a function f, its derivative f', and an initial guess y_0 for a root of f. If f satisfies certain assumptions and the guess is close to the root, then

$$y_1 = y_0 - \frac{f(y_0)}{f'(y_0)}$$

will be a better approximation of the root than y_0. Geometrically, $(y_1, 0)$ is the intercept of the tangent to f at $(y_0, f(y_0))$, as depicted in Fig. 20.1.

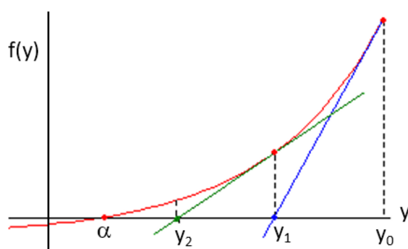


Figure 20.1. Newton-Raphson Approaching a Root at α.

The process repeatedly update y:

$$y_{k+1} = y_k - \frac{f(y_k)}{f'(y_k)}$$

until y_{k+1} is close enough to the root at α.

We use Newton-Raphson to approximate $y = \sqrt{x}$ by rephrasing it as a root-finding problem. Specify f as

$$f(y) = y^2 - x$$

Since $f'(y) = 2y$, the iterator becomes

$$\begin{aligned} y_{k+1} &= y_k - \frac{f(y_k)}{f'(y_k)} \\ &= y_k - \frac{y_k^2 - x}{2y_k} \\ &= \frac{1}{2}(y_k + \frac{x}{y_k}). \end{aligned}$$

The number of correct digits in y_k roughly doubles with each step. To see this, consider each y_k as the root we seek (α) plus some error term e_k, which is its distance from the root (e.g. as pictured in Fig. 20.1). Consider how the error changes from one iteration, $y_k = \alpha + e_k$, to the next, $y_{k+1} = \alpha + e_{k+1}$. These expressions, and the fact that $\alpha^2 = x$, can be substituted into the Newton-Raphson formula to give:

$$\alpha + e_{k+1} = \frac{1}{2}\left(\alpha + e_k + \frac{\alpha^2}{\alpha + e_k}\right).$$

After some rearranging, this becomes:

$$e_{k+1} = \frac{e_k^2}{2(\alpha + e_k)} = \frac{e_k^2}{2(\sqrt{x} + e_k)}.$$

When y_k gets close to the root α, the error term becomes small enough so that it can be omitted from the denominator, yielding a quadratic approximation

$$e_{k+1} \approx \frac{e_k^2}{2\sqrt{x}}.$$

Since the error is being *squared* in each iteration, the number of correct decimal digits doubles. For example, if y_k possesses k correct digits then $e_k \approx 10^{-k}$, and

$$e_{k+1} \approx \frac{(10^{-k})^2}{2\sqrt{x}} \approx 10^{-2k},$$

resulting in roughly $2k$ correct digits. The constant factor $1/(2\sqrt{x})$ affects this progression but will become negligible after the first few steps. This is confirmed by the results of errDigits.py described below.

Stage (2) of the pseudo-code reduces the initial error by constraining x between 0 and 1. The range-reduction loop keeps dividing x by 100 and multiplying a scale factor t by 10 until $x \leq 1$. The domain is therefore the interval $(1/100, 1]$.

Stage (3) starts the Newton-Raphson process with a value obtained from $y_0 = a + bx$. A good initial value should fix a and b in order to minimize the maximum relative error:

$$\epsilon(x) = \frac{(a + bx) - \sqrt{x}}{\sqrt{x}}.$$

20.2.0.1 Determining a and b. Deciding on suitable values for a and b is a *minimax* approximation problem.

The *Chebyshev equioscillation theorem* states that a minimax approximation of a curve of degree n is characterized by *at least* $n + 2$ points where the relative error $\epsilon(x)$ has a maximum magnitude E with alternating signs. So for our linear polynomial, we need at least three points to determine suitable values for a and b.

Three suitable points within $(1/100, 1]$ are the left boundary $x_L = 1/100$, the *interior critical* point x^*, and the right endpoint at $x = 1$. The equioscillation condition means that:

$$\epsilon(x_L) = +E, \qquad \epsilon(x^*) = -E, \qquad \epsilon(1) = +E.$$

The interior critical point satisfies $\epsilon'(x) = 0$, and so for our square root problem:

$$\frac{d\epsilon}{dx} = \left(b\sqrt{x} - \frac{a + bx}{2\sqrt{x}}\right)/x = 0.$$

Multiplying through by $x\sqrt{x}$, and focusing on the numerator:

$$bx - \frac{a+bx}{2} = 0 \implies bx = a \implies x^* = \frac{a}{b}.$$

Now we substitute x_L, x^*, and $x = 1$ into the relative error function $\epsilon(x)$:

$$\begin{aligned}
\epsilon(x_L) &= \epsilon(1/100) = \frac{a+(b/100)-(1/10)}{1/10} = 10a + (b/10) - 1 = +E. && (1)\\
\epsilon(x^*) &= \frac{a+b\cdot(a/b)}{\sqrt{a/b}} - 1 = \frac{2a}{\sqrt{a/b}} - 1 = 2\sqrt{ab} - 1 = -E. && (2)\\
\epsilon(1) &= a + b - 1 = +E. && (3)
\end{aligned}$$

Solve for a and b in $y_0 = a + bx$ by adding equs. (3) and (2):

$$\begin{aligned}
(a+b-1) + (2\sqrt{ab}-1) = 0 &\implies a + b + 2\sqrt{ab} = 2\\
&\implies \left(\sqrt{a}+\sqrt{b}\right)^2 = 2,
\end{aligned}$$

or

$$\sqrt{a} + \sqrt{b} = \sqrt{2}. \qquad (4)$$

Also, let's equate equ. (1) and equ. (3) since they both equal $+E$:

$$\begin{aligned}
10a + \frac{b}{10} - 1 = a + b - 1 &\implies 9a = b - \frac{b}{10} = \frac{9b}{10}\\
&\implies a = \frac{b}{10}.
\end{aligned}$$

Substituting $a = b/10$ into equ. (4):

$$\begin{aligned}
\sqrt{\frac{b}{10}} + \sqrt{b} = \sqrt{2} &\implies \sqrt{b}\left(\frac{1}{\sqrt{10}} + 1\right) = \sqrt{2}\\
&\implies \sqrt{b} = \frac{\sqrt{2}}{1 + 1/\sqrt{10}} = \frac{\sqrt{20}}{\sqrt{10}+1}.
\end{aligned}$$

Let's assume that $\sqrt{10} = 3.16228$ in order to get some simple values for a and b:

$$\sqrt{b} = \frac{4.47214}{4.16228} = 1.07441, \qquad b = 1.15436, \qquad a = \frac{b}{10} = 0.115436.$$

The linear polynomial is therefore

$$y_0 = 0.115436 + 1.15436\,x,$$

which agrees with Allison's equation to five significant figures.

Aside from the equation, we can also substitute $a = 0.115442$ and $b = 1.15442$ into equ. (2) to give us a value for the peak relative error E:

$$\begin{aligned}
E &= 1 - 2\sqrt{0.115442 \times 1.15442} = 1 - 2\sqrt{0.133284}\\
&= 1 - 2(0.36509) = 0.2698.
\end{aligned}$$

Therefore, the number of correct decimal digits is

$$d = -\log_{10}(E) = -\log_{10}(0.2698) \approx 0.569 \text{ digits},$$

which agrees with the 0.56 digits of accuracy mentioned in Allison's description of the square-root function.

20.2.0.2 Why Bother with $y_0 = a + bx$? The previous subsection seems to be a lot of work just to set the Newton-Raphson starting value somewhere between 1/100 and 1. To see what advantage it gives use, let's compare this linear approximation with simply fixing $y_0 = 1$ or 0.5 in the relative error equation:

$$\epsilon(x) = \frac{y_0 - \sqrt{x}}{\sqrt{x}}.$$

Case 1. When $y_0 = 1$:

$$\epsilon_0(x) = \frac{1-\sqrt{x}}{\sqrt{x}} = \frac{1}{\sqrt{x}} - 1.$$

This is a decreasing function of x, so the worst case is at the left boundary:

$$\epsilon_0\left(\frac{1}{100}\right) = \frac{1}{1/10} - 1 = 9.0, \qquad d_0 = -\log_{10}(9.0) \approx -0.954 \text{ digits}.$$

The negative indicates that the worst-case relative error exceeds 100%, meaning that y_0 doesn't contain even one correct digit.

Case 2. $y_0 = 0.5$:

$$\epsilon_0(x) = \frac{0.5}{\sqrt{x}} - 1.$$

The worst case is again at the left boundary:

$$\epsilon_0\left(\frac{1}{100}\right) = \frac{0.5}{0.1} - 1 = 4.0, \qquad d_0 = -\log_{10}(4.0) \approx -0.602 \text{ digits}.$$

Although $y_0 = 1$ and 0.5 are both worse than using $y_0 = a + bx$ in terms of the number of correct digits in the starting value of Newton-Raphson, how does this affect its subsequent performance? In particular, how many more iterations does it take for the method to reach a good level of digit accuracy?

Recall that the recursive error definition is:

$$e_{k+1} = \frac{e_k^2}{2(\sqrt{x} + e_k)}.$$

which models how the error in Newton-Raphson reduces as the method closes in upon an answer.

For $y_0 = 0.115436 + 1.15436\,x$, we found that $e_0 = 0.2698$, and for the fixed starting values, we got $e_0 = 9$ for $y_0 = 1$ and $e_0 = 4$ for $y_0 = 0.5$. Evaluate the

recursive error equation starting with each of these e_0's, and keep going until the number of correct digits d exceeds some threshold (recall that $d = -log_{10}(e)$). Report the number of iterations required to reach that value.

Table 20.1 does all of that; its data was obtained by running `errDigits.py` with the different e_0's.

Iteration n	Linear y_0 $e_0 = 0.27$	Fixed $y_0 = 0.5$ $e_0 = 4.0$	Fixed $y_0 = 1$ $e_0 = 9.0$
0	0.569	−0.602	−0.954
1	1.542	−0.204	−0.607
2	3.398	0.308	−0.211
3	7.096	1.090	0.299
4	14.494	2.516	1.075
5	29.288	5.334	2.487
6	—	10.969	5.277
7	—	22.239	10.854
8	—	—	22.009

Table 20.1. Iterations to achieve a digit accuracy of $d > 15$ for different e_0's

Newton-Raphson's quadratic convergence only kicks in after e_k has become positive. Without $y_0 = a + bx$, the number of iterations increases from about 4 to 6 or 7. So, employing a linear approximation reduces the running time of the square-root function quite significantly.

20.3 Dirty Log Base 10

Allison's pseudo-code:

(1) $x \leq 0$ is treated as an error.

(2) If $x < 1$ then {sign $= -1; x = 1/x$} else sign $= 1$, relying on the the fact that $\log(1/x) = -\log(x)$.

(3) Reduce the range of x by dividing by 10^n, such that $1/10 \leq \frac{x}{10^n} < 1$, and redefine x as $\frac{x}{10^n}$.

(4) Introduce the variable $s = \sqrt{10}\,x$, which fixes the input range to $1/\sqrt{10} \leq s < \sqrt{10}$.

(5) Change variables yet again so that $z = (s - 1)/(s + 1)$, and calculate the rational polynomial, $r(z)$, using the supplied P and Q coefficients, with z^2 as the argument, and multiplying by z:

$$r(z) = z\,\frac{P(z^2)}{Q(z^2)}$$

The coefficients:

P_0	3.1878	22082	024
P_1	−2.6558	08794	66
P_2	0.26686	32700	47
Q_0	3.6701	15625	115
Q_1	−4.2809	73292	83
Q_2	1.0		

(6) The $log_{10}(x)$ result is calculated using $(n + r - 0.5) \cdot$ sign.

The corresponding code in `dirty.py`:

```
SQRT10 = 3.162278

def log10D(x):
  if x <= 0:
    raise ValueError("Math domain error")

  sign = 1
  if x < 1:
    sign = -1
    x = 1/x

  # range reduction to [0.1, 1.0]
  n = 0
  while x >= 1:
    x = x/10
    n += 1      # count powers of 10

  # set to range [1/sqrt(10, sqrt(10)]
  s = SQRT10 * x
  z = (s-1)/(s+1)
  z2 = z*z

  # Coeffs for the rational approx
  p = 3.187822 + z2*(-2.655809 + z2*0.2668633)
  q = 3.670116 + z2*(-4.280973 + z2)
  y = (n + z*p/q - 0.5) * sign
  return y
```

Step (5) in the pseudo-code depends on the hyperbolic arctangent (atanh) identity:

$$\ln(s) = 2 \cdot \operatorname{atanh}\left(\frac{s-1}{s+1}\right)$$

See section 20.3.1 to see why this identity is true.

By defining $z = \frac{s-1}{s+1}$, the Maclaurin series for atanh(z) is simply:

$$\ln(s) = 2\left(z + \frac{z^3}{3} + \frac{z^5}{5} + \cdots\right) = 2z\left(1 + \frac{z^2}{3} + \frac{z^4}{5} + \cdots\right).$$

If you're unsure of how atanh(z) is expanded into a series, look at section 20.3.2.

We want base-10 logarithms, but that's only a change of base:

$$\log_{10}(s) = \frac{\ln(s)}{\ln(10)} = \frac{2}{\ln(10)} z\left(1 + \frac{z^2}{3} + \frac{z^4}{5} + \cdots\right).$$

Since the rational polynomial approximates the series:

$$\frac{P(w)}{Q(w)} = \frac{2}{\ln(10)}\left(1 + \frac{w}{3} + \frac{w^2}{5} + \frac{w^4}{7} + \cdots\right),$$

then it is called like so:

$$\log_{10}(z) = z\frac{P(z^2)}{Q(z^2)}.$$

The advantage of evaluating z^2 instead of z, is that it halves the number of multiplications within the series calculation.

20.3.1 Deriving the $\ln(x) = \operatorname{atanh}(x)$ Identity.

Why does

$$\ln(s) = 2 \cdot \operatorname{atanh}\left(\frac{s-1}{s+1}\right)?$$

Let's assume the exponential identities for the hyperbolic cosine and sine:

$$\cosh(t) = \frac{e^t + e^{-t}}{2}, \qquad \sinh(t) = \frac{e^t - e^{-t}}{2}.$$

So,

$$\tanh(t) = \frac{\sinh(t)}{\cosh(t)} = \frac{e^t - e^{-t}}{e^t + e^{-t}}.$$

Set $y = \tanh(t)$, and multiply the numerator and denominator by e^t:

$$y = \tanh(t) = \frac{e^{2t} - 1}{e^{2t} + 1}.$$

Rearrange:

$$e^{2t} = \frac{1+y}{1-y}.$$

Take the natural logarithm of both sides:

$$2t = \ln\left(\frac{1+y}{1-y}\right) \implies t = \frac{1}{2}\ln\left(\frac{1+y}{1-y}\right).$$

But $t = \text{atanh}(y)$, so:

$$\text{atanh}(y) = \frac{1}{2}\ln\left(\frac{1+y}{1-y}\right), \qquad |y| < 1 \qquad (1)$$

The constraint $|y| < 1$ is due to $\tanh(t)$'s range.

We want the equality to be in terms of $\ln(x)$, which requires some variable renaming. Let $y = (x-1)/(x+1)$, and use it to simplify $(1+y)/(1-y)$:

$$\text{The numerator:} \quad (1+y) = 1 + \frac{x-1}{x+1} = \frac{(x+1)+(x-1)}{x+1} = \frac{2x}{x+1}$$

$$\text{The denominator:} \quad (1-y) = 1 - \frac{x-1}{x+1} = \frac{(x+1)-(x-1)}{x+1} = \frac{2}{x+1}$$

Plug these back into the logarithm's ratio:

$$\frac{1+y}{1-y} = \frac{2x/(x+1)}{2/(x+1)} = x.$$

Now we can rewrite the y arguments in equ. (1)'s atanh = ln equality in terms of x:

$$\text{atanh}\left(\frac{x-1}{x+1}\right) = \frac{1}{2}\ln(x).$$

So:

$$\ln(x) = 2\ \text{atanh}\left(\frac{x-1}{x+1}\right), \qquad x > 0.$$

The $x > 0$ domain restriction follows from $|y| < 1$.

20.3.2 Deriving the atanh(x) Series.

Begin with

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots, \qquad |x| < 1.$$

Integrate term-by-term from 0 to x,

$$\int_0^x \frac{dt}{1-t} = \int_0^x (1 + t + t^2 + t^3 + \cdots)\, dt,$$

to get

$$-\ln(1-x) = x + \frac{x^2}{2} + \frac{x^3}{3} + \frac{x^4}{4} + \cdots.$$

Therefore

$$\ln(1-x) = -\left(x + \frac{x^2}{2} + \frac{x^3}{3} + \frac{x^4}{4} + \cdots\right).$$

Now replace x by $-x$ in the original series:

$$\frac{1}{1+x} = 1 - x + x^2 - x^3 + x^4 - \cdots.$$

Once again integrate term-by-term,

$$\int_0^x \frac{dt}{1+t} = \int_0^x (1 - t + t^2 - t^3 + t^4 - \cdots) dt,$$

which gives us

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots.$$

Substitute these two series into the identity:

$$\text{atanh}(y) = \frac{1}{2}\ln\left(\frac{1+y}{1-y}\right),$$

producing

$$\text{atanh}(x) = \frac{1}{2}[(x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots) - (-x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \frac{x^5}{5} - \cdots)].$$

The terms on the right-hand side either cancel or double up, leaving:

$$\text{atanh}(x) = \frac{1}{2}\left(2x + \frac{2x^3}{3} + \frac{2x^5}{5} + \frac{2x^7}{7} + \cdots\right),$$

so

$$\text{atanh}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \cdots.$$

20.3.3 Checking the Polynomial's Coefficients. The rational polynomial uses P_0, P_1, P_2, Q_0, Q_1, and Q_2:

$$r(x) = \frac{P_0 + P_1x + P_2x^2}{Q_0 + Q_1x + Q_2x^2}$$

This polynomial models the power series:

$$r(x) = \frac{2}{\ln(10)}\left(1 + \frac{x}{3} + \frac{x^2}{5} + \frac{x^4}{7} + \ldots\right).$$

One simple way to test if the rational and series are equivalent is to consider their values at $x = 0$. The rational becomes

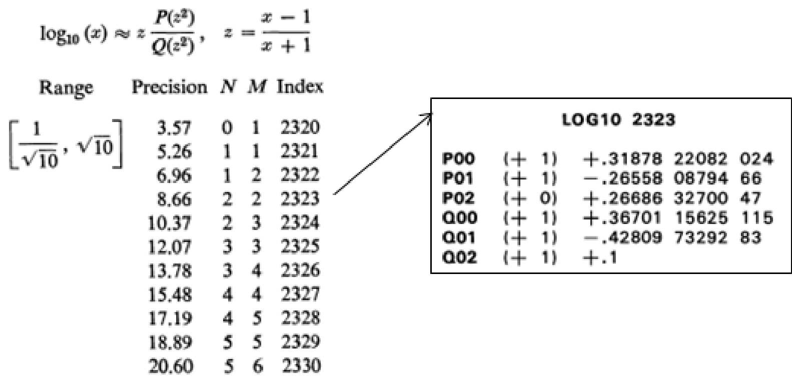
$$\frac{P_0}{Q_0} = \frac{3.187822082024}{3.670115625115} \approx 0.86859$$

while the series is:

$$\frac{2}{\ln(10)} \approx \frac{2}{2.302585093} \approx 0.86859$$

So the same (approximate) result, which is some indication that the rational does approximate the power series.

20.3.4 Why use a Rational Polynomial? Allison obtained his coefficients from Hart's *Computer Approximations* [**Har68**], but why these particular ones, especially when Hart offers a range of approximations for every function, in the form of polynomials and as rationals? Allison chose the rational listed on p.110 of Hart's text, and coefficients that offer a precision of 8.66 (8 or 9 correct decimal digits). Hart's coefficients table 2323 is printed on p.221, but Fig. 20.2 shows the rational and coefficients together.



$$\log_{10}(x) \approx z \frac{P(z^2)}{Q(z^2)}, \quad z = \frac{x-1}{x+1}$$

Range	Precision	N	M	Index
$\left[\frac{1}{\sqrt{10}}, \sqrt{10}\right]$	3.57	0	1	2320
	5.26	1	1	2321
	6.96	1	2	2322
	8.66	2	2	2323
	10.37	2	3	2324
	12.07	3	3	2325
	13.78	3	4	2326
	15.48	4	4	2327
	17.19	4	5	2328
	18.89	5	5	2329
	20.60	5	6	2330

LOG10 2323

P00	(+ 1)	+.31878 22082 024
P01	(+ 1)	−.26558 08794 66
P02	(+ 0)	+.26686 32700 47
Q00	(+ 1)	+.36701 15625 115
Q01	(+ 1)	−.42809 73292 83
Q02	(+ 1)	+.1

Figure 20.2. A Hart Rational for $\log_{10} x$

One reason for choosing a rational is that it gives fairly good answers even near asymptotes in the function. This is an issue for atanh(x) near $x = -1$ and 1, as shown in Fig. 20.3.

However, such concerns are avoided in Allison's code by restricting the input range in

$$r = z\,\frac{P(z^2)}{Q(z^2)}.$$

$z = \frac{s-1}{s+1}$ and s is in the interval $[\frac{1}{\sqrt{10}}, \sqrt{10}]$, meaning that:

- when $s = \frac{1}{\sqrt{10}}$, $z = \frac{(1/\sqrt{10})-1}{(1/\sqrt{10})+1} = \frac{1-\sqrt{10}}{1+\sqrt{10}} \approx -0.51949$;
- when $s = 1$, $z = 0$;
- when $s = \sqrt{10}$, $z = \frac{\sqrt{10}-1}{\sqrt{10}+1} \approx +0.51949$.

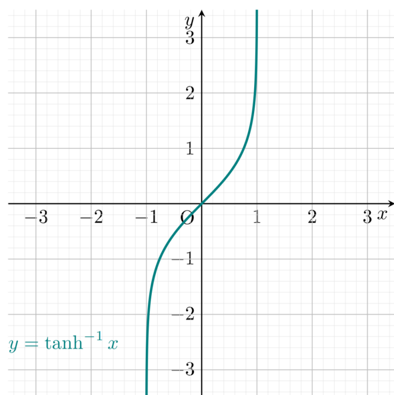


Figure 20.3. The atanh(x) Curve

20.3.5 Scaling the Result in Step (6). Step (6) specifies that the function returns $(n + r - 0.5) \cdot$ sign. r is the polynomial result, so equivalent to $\log_{10}(s)$, and s is a scaled version of the x input, such that $s = (\sqrt{10}/(10^n))x$. Putting these together:

$$
\begin{aligned}
r &= \log_{10}(s) \\
&= \log_{10}\left(\frac{\sqrt{10}}{10^n}x\right) \\
&= \log_{10}(\sqrt{10}) - \log_{10}(10^n) + log_{10}(x) \\
&= 0.5 - n + log_{10}(x)
\end{aligned}
$$

So $(r + n - 0.5)$ undoes the scaling to produce $\log_{10}(x)$, with the additional step of applying the correct sign.

20.3.6 `dirty.py` in Action. dirty.py includes a test-rig which compares the results when calling `log10D(x)` versus `math.log10(x)` for $x = 0.1, 1, 10, 50$, and 1000. The output is shown below:

```
Func    | Input       | Dirty        | Python       | Error
----------------------------------------------------------------
log10   |  0.100000   | -1.000000    | -1.000000    | 1.64e-07
log10   |  1          |  0.000000    |  0.000000    | 1.64e-07
log10   |  10         |  1.000000    |  1.000000    | 1.64e-07
log10   |  50         |  1.698970    |  1.698970    | 1.61e-08
log10   |  1000       |  3.000000    |  3.000000    | 1.64e-07
```

Errors start to appear at around the seventh decimal place, within the expected range.

20.3.7 Comparison with a Modern Approach. Although Allison's article dates from 1978, its scaling of inputs, followed by the use of a polynomial approximation, is still popular, albeit in a more complicated form. One fairly easy comparison is to look at MicroPython (`https://micropython.org/`) which runs on a range of micro-controllers, including its own development board. MicroPython's math library is documented at `https://docs.micropython.org/en/latest/library/math.html`, and includes a $\log_{10}(x)$ function:

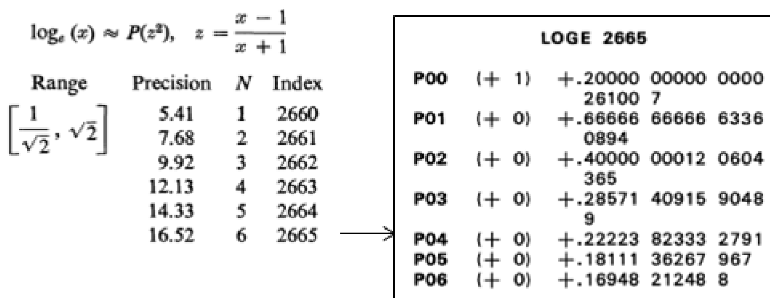
```
#include <math.h>

static const double _M_LN10 = 2.302585092994046;

double log10(double x)
{   return log(x) / (double)_M_LN10;  }
```

The code for $\log(x)$ is at `https://github.com/micropython/micropython/blob/master/lib/libm_dbl/log.c`, and is a bit too complex to describe here in detail. It essentially follows the same steps as Allison's code: it scales its input, then employs a 7-term polynomial approximation of $\log(1 + x)$ constrained to the range $\sqrt{2}/2 < x < \sqrt{2}$. It sets $s = x/(2 + x)$, which allows $log(1 + x)$ to be expressed as $log(1 + s) - log(1 - s)$, which let's it employ the atanh() series.

Amazingly, Hart's textbook (from 1968) contains a very similar polynomial on p.111 and in table 2665 (p.227), as shown in Fig. 20.4.



$\log_e(x) \approx P(z^2), \quad z = \frac{x-1}{x+1}$

Range	Precision	N	Index
$\left[\frac{1}{\sqrt{2}}, \sqrt{2}\right]$	5.41	1	2660
	7.68	2	2661
	9.92	3	2662
	12.13	4	2663
	14.33	5	2664
	16.52	6	2665

LOGE 2665

P00	(+ 1)	+.20000 00000 0000 26100 7
P01	(+ 0)	+.66666 66666 6336 0894
P02	(+ 0)	+.40000 00012 0604 365
P03	(+ 0)	+.28571 40915 9048 9
P04	(+ 0)	+.22223 82333 2791
P05	(+ 0)	+.18111 36267 967
P06	(+ 0)	+.16948 21248 8

Figure 20.4. Hart Rational Polynomial for log

MicroPython runs on 32-bit architectures and, as explained in Appendix E, 32-bit IEEE 754 floating-point employ 1 sign bit, 8 exponent bits, and 23 stored fraction bits. This fractional part supports 24 bits of precision (after including the hidden leading 1), equivalent to $\log_{10}(2^{24}) \approx 7$ decimal digits. However, the

choice of coefficients in the MicroPython code means that it offers a precision of around 16 digits, making it suitable for 64-bit architectures as well.

20.4 Dirty Arctangent

Allison's pseudo-code:

(1) If $x < 0$ then return $-\arctan(|x|)$ since $\arctan(-x) = -\arctan(x)$.

(2) If $x > 1$ then return $(\pi/2 - \arctan(1/x))$ since $\arctan(|x|) = (\pi/2 - \arctan(1/|x|))$ and $x > 0$.

(3) Compute $\arctan(x)$ in the range $[0, \tan(\pi/4) = 1]$ by setting $z = x * x$ and calling the polynomial:

$$\arctan(x) = x \frac{P_0 + P_1 z + P_2 z^2}{Q_0 + Q_1 z + Q_2 z^2 + Q_3 z^3}$$

The coefficients are:

P_0	80.78998484078
P_1	72.5814378046
P_2	11.11774163116
Q_0	80.78998647615
Q_1	99.5112677199
Q_2	28.13285868067
Q_3	1.0

The corresponding code in dirty.py:

```
PI = 3.141593
HALF_PI = PI/2

def atanD(x):
  sign = 1
  if x < 0:
    sign = -1
    x = -x
  isInvert = (x > 1)
  if isInvert:
    x = 1/x

  # range reduction to [0, tan pi/4 = 1]
  x2 = x * x
  # Rational approximation coefficients
```

```
# Hart ARCTN 5092
p = 80.78998 + x2 * (72.58144 + x2 * 11.11774)
q = 80.78999 + x2 * (99.51128 + x2 * (28.13286 + x2))
y = x * p/q

if isInvert:
  y = HALF_PI - y    # atan(x) = pi/2 - atan(1/x)
return y * sign
```

Step (2) is used to restrict the range of x to $[0, \tan(\pi/4) = 1]$, and is best understood geometrically via Fig. 20.5.

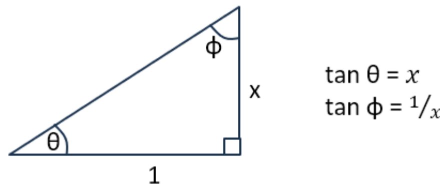


Figure 20.5. Complementary Tangents in a Right-angled Triangle

The right triangle in Fig. 20.5 has legs x and 1. If θ is the angle opposite the side of length x, then $\tan\theta = x$. The other angle ϕ satisfies $\tan\phi = 1/x$. Since the two angles of a right triangle add to $\pi/2$ radians, then

$$\theta + \phi = \pi/2,$$

which gives us

$$\arctan(x) = \pi/2 - \arctan(1/x).$$

Also, since $\theta > \pi/4$, then $\phi < \pi/4$, which allows us to constrain the polynomial's range by only passing it input equivalent to $\tan(\phi) < 1$.

The Maclaurin series expansion for $\arctan(x)$ is

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots,$$

and if we factor out an x, the terms all become even powers:

$$\arctan(x) = x\left(1 - \frac{x^2}{3} + \frac{x^4}{5} - \frac{x^6}{7} + \ldots\right).$$

If we apply the substitution $z = x^2$ to the bracketed sequence, then only sequential powers of z are needed inside it:

$$\arctan(x) = x\left(1 - \frac{z}{3} + \frac{z^2}{5} - \frac{z^3}{7} + \ldots\right)$$

It's this inner sequence that's modeled by the rational polynomial:

$$\frac{P(z)}{Q(z)} = 1 - \frac{z}{3} + \frac{z^2}{5} - \frac{z^3}{7} + \ldots$$
$$= \frac{P_0+P_1z+P_2z^2}{Q_0+Q_1z+Q_2z^2+Q_3z^3}$$

We can obtain some evidence for this by considering the $z = 0$ case. The series will equal 1 while the polynomial will be:

$$\frac{P_0}{Q_0} = \frac{80.78998484078}{80.78998647615} \approx 1.0$$

There's also the question of why a rational polynomial is being used at all. It's for the same reason as atanh() utilization in the previous section – arctan() has asymptotes, as shown in Fig. 20.6, and a rational can better deal with inputs near those extremes.

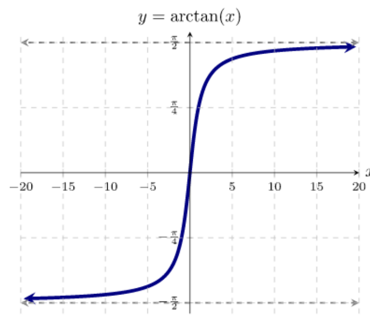


Figure 20.6. The arctan(x) Curve

As usual, the coefficients come from Hart's *Computer Approximations* [**Har68**]: the rational from p.130 and table 5092 on p.275; Fig. 20.7 shows them together.

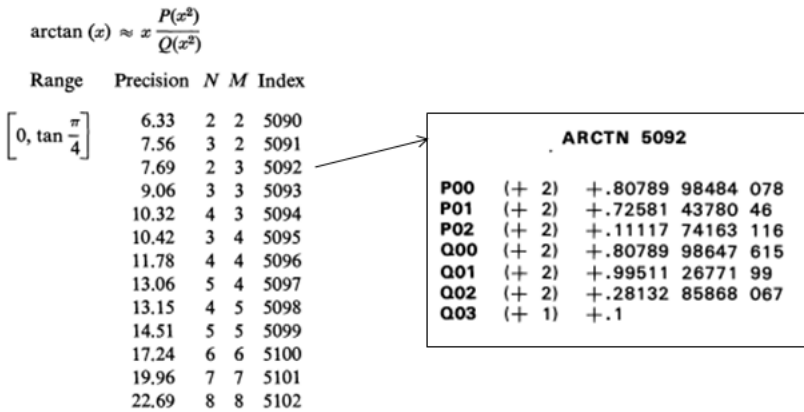
As with the atanh() polynomial, Allison chose a precision close to 8 decimal digits.

20.4.1 Deriving the arctan() **Series.** Let's define $y = \arctan(x)$ so that $x = \tan(y)$, and then differentiate both sides with respect to x:

$$\frac{d}{dx}x = \frac{d}{dx}\tan(y),$$

After applying the chain rule, and some rearranging of terms, we get

$$\frac{dy}{dx} = \frac{1}{\sec^2(y)}.$$



$\arctan(x) \approx x \frac{P(x^2)}{Q(x^2)}$

Range	Precision	N	M	Index
$\left[0, \tan\frac{\pi}{4}\right]$	6.33	2	2	5090
	7.56	3	2	5091
	7.69	2	3	5092
	9.06	3	3	5093
	10.32	4	3	5094
	10.42	3	4	5095
	11.78	4	4	5096
	13.06	5	4	5097
	13.15	4	5	5098
	14.51	5	5	5099
	17.24	6	6	5100
	19.96	7	7	5101
	22.69	8	8	5102

ARCTN 5092

P00	(+ 2)	+.80789 98484 078
P01	(+ 2)	+.72581 43780 46
P02	(+ 2)	+.11117 74163 116
Q00	(+ 2)	+.80789 98647 615
Q01	(+ 2)	+.99511 26771 99
Q02	(+ 2)	+.28132 85868 067
Q03	(+ 1)	+.1

Figure 20.7. A Hart Rational Polynomial for arctan

With the help of the trigonometric identity $\sec^2(y) = 1 + \tan^2(y)$, this is

$$\frac{dy}{dx} = \frac{1}{1 + \tan^2(y)},$$

and since we started with $x = \tan(y)$, then

$$\frac{d}{dx} \arctan(x) = \frac{1}{1 + x^2}.$$

The Maclaurin series for arctan() appears by expanding the right-hand side of this derivative. Let's assume that

$$\frac{1}{1+t} = 1 - t + t^2 - t^3 + t^4 - \cdots \qquad |t| < 1.$$

Substituting $t = x^2$ gives

$$\frac{1}{1+x^2} = 1 - x^2 + x^4 - x^6 + x^8 - \cdots \qquad |x| < 1.$$

Plug this into the right-hand side, and integrate term-by-term from 0 to x:

$$\arctan(x) = \int_0^x \frac{1}{1+t^2}\, dt = \int_0^x (1 - t^2 + t^4 - t^6 + t^8 - \cdots)\, dt.$$

Integrating each term yields

$$\arctan(x) = \left[t - \frac{t^3}{3} + \frac{t^5}{5} - \frac{t^7}{7} + \frac{t^9}{9} - \cdots \right]_0^x$$

which evaluates to

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \cdots$$

20.4.2 `dirty.py` in Action. The dirty.py test-rig compares the results when calling `atanD(x)`versus `math.atan(x)` for $x = 0, 0.5, 1, 5, -1$. The output is shown below:

```
Func    | Input       | Dirty          | Python         | Error
---------------------------------------------------------------
atan    |  0          |  0.000000      |  0.000000      | 0.00e+00
atan    |  0.500000   |  0.463648      |  0.463648      | 5.49e-08
atan    |  1          |  0.785398      |  0.785398      | 1.01e-07
atan    |  5          |  1.373401      |  1.373401      | 1.90e-07
atan    | -1          | -0.785398      | -0.785398      | 1.01e-07
```

Errors start to appear at around the seventh decimal place, within the expected range.

20.5 Dirty Sine

Allison's pseudo-code for $sin(x)$ is really aimed at implementing cosine, and starts by switching to that function:

$$\sin(x) = \cos(\frac{\pi}{2} - x)$$

The remaining pseudo-code has five stages:

(1) Input that's too large is detected by looking at the number of decimal digits it uses. If $|x| > 10^{d-3}$, then an error is reported.

(2) The cosine's even property, $\cos(-x) = \cos(x)$, is employed to deal with negative values.

(3) The cosine's periodicity, $\cos(2\pi + x) = \cos x$, is used to constrain the input range to $0 < x < 2\pi$.

(4) The cosine's symmetry across the four quadrants of the unit circle allows its input range to be limited to quadrant 1, along with suitable sign changes:

$$
\begin{aligned}
0 \le x < \frac{\pi}{2}: &\quad s = +1 \quad x = x \\
\frac{\pi}{2} \le x < \pi: &\quad s = -1 \quad x = |\pi - x| \\
\pi \le x < \frac{3\pi}{2}: &\quad s = -1 \quad x = |\pi - x| \\
\frac{3\pi}{2} \le x < 2\pi: &\quad s = +1 \quad x = |x - 2\pi|
\end{aligned}
$$

s stores the sign required by the result.

(5) A power series polynomial is called:

$$
\begin{aligned}
z &= x^2 \\
y &= P_0 + P_1 z + P_2 z^2 + P_3 z^3 + P_4 z^4 \\
\cos &= y * s
\end{aligned}
$$

with the coefficients:

P_0	0.999999953464
P_1	−0.499999053455
P_2	0.0416635846769
P_3	−0.0013853704264
P_4	0.00002315393167

The corresponding code in `dirty.py`:

```
def sinD(x):
  # Shift phase to use cosine approximation
  x = HALF_PI - x
  if abs(x) > 1000:
    raise ValueError("Input too large")
  if x < 0:
    x = -x

  # Range reduction to [0, 2*pi)
  x = x % TWO_PI

  # Determine quadrant (0 to 3; ccw) and sign
  quadrant = int(x / HALF_PI)
  sign = 1
  if quadrant == 1 or quadrant == 2:
    sign = -1
    x = PI - x
  elif quadrant == 3:
```

```
    x = x - TWO_PI

  x2 = x * x
  # Polynomial approximation for cosine
  # Hart p.118, COS 3502
  y = 0.9999999 + x2*(-0.4999991 + x2*(0.04166359 + \
      x2*(-0.001385370 + x2*(0.00002315393))))

  return y * sign
```

In step (1), the check $|x| > 10^{d-3}$ deals with floating-point precision loss during argument reduction, specifically in step (3) when the range is reduced to $[0, 2\pi)$ with $x = x \% 2\pi$. A modulo operation may work by repeatedly subtracting 2π. Since π can't be represented exactly in memory, all of these subtractions can cause rounding errors to accumulate.

The x threshold 10^{d-3} uses d, the chosen number of digits of precision, with -3 as a safety margin. I've hardwired $d = 7$ into the code, by checking for $|x| > 1000$.

Steps (2), (3), and (4) use reflectivity, periodicity, and symmetry as they apply to the cosine (see Fig. 20.8) to reduce the input range to $[0, \pi/2)$.

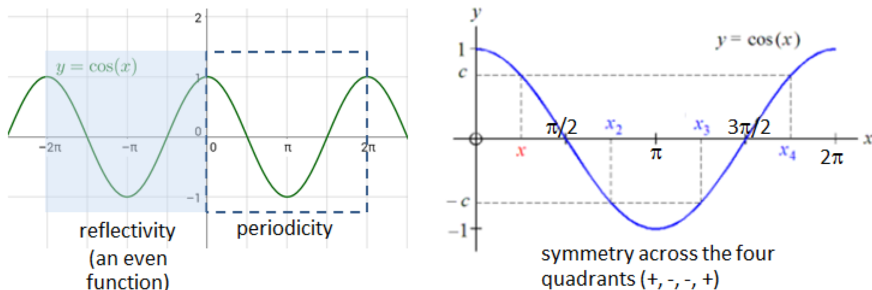


Figure 20.8. Reflectivity, periodicity, and symmetry for cosine

The Maclaurin series expansion for $\cos(x)$ is

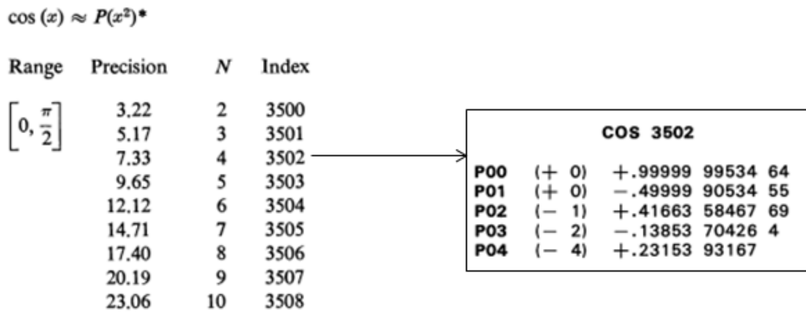
$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \ldots$$

The substitution $z = x^2$ turns the polynomial into an integer sequence of powers:

$$\cos(x) = 1 - \frac{z}{2} + \frac{z^2}{24} - \frac{z^3}{720} + \frac{z^4}{40320} - \ldots$$

This is the first time our approximation has *not* been a rational polynomial. There's no need since cosine doesn't have asymptotes, unlike atanh() and arctan().

Once again, the coefficients come from Hart's *Computer Approximations* [**Har68**]: the polynomial from p.123 and its table 3502 from p.241; Fig. 20.9 shows them together.



$\cos(x) \approx P(x^2)$*

Range	Precision	N	Index
$\left[0, \frac{\pi}{2}\right]$	3.22	2	3500
	5.17	3	3501
	7.33	4	3502
	9.65	5	3503
	12.12	6	3504
	14.71	7	3505
	17.40	8	3506
	20.19	9	3507
	23.06	10	3508

COS 3502

P00	(+ 0)	+.99999 99534 64
P01	(+ 0)	−.49999 90534 55
P02	(− 1)	+.41663 58467 69
P03	(− 2)	−.13853 70426 4
P04	(− 4)	+.23153 93167

Figure 20.9. A Hart Power Series Polynomial for cos

Allison wanted precision close to 7 decimal digits (i.e. $d = 7$), which backs up his test that $x > 1000$ or 10^{d-3}.

The coefficients in table 3502 are 'related' to the Maclaurin expansion for cosine, as shown in Table 20.2. The differences are due to Hart's use of Remez approximation which tends to modify the series coefficients depending on the input range constraints.

Coefficient	Coef Float	Series Fraction	Series Float
P_0	0.999999953464	+1	1.0
P_1	−0.499999053455	−1/2!	−0.5
P_2	0.0416635846769	+1/4!	0.04166666666 ...
P_3	−0.0013853704264	−1/6!	−0.00138888888 ...
P_4	0.00002315393167	+1/8!	0.00002480158 ...

Table 20.2. Comparison of Polynomial and Series Coefficients

The dirty.py test-rig compares `sinD(x)` versus `math.sin(x)`. The output:

```
Func    | Input         | Dirty          | Python         | Error
-----------------------------------------------------------------
sin     |  0            |  0.000000      |  0.000000      | 2.56e-07
sin     |  0.785398     |  0.707107      |  0.707107      | 2.26e-07
sin     |  1.570796     |  1.000000      |  1.000000      | 1.00e-07
sin     |  3.141593     |  0.000000      |  0.000000      | 9.02e-08
sin     |  6.283185     | -0.000000      | -0.000000      | 4.37e-07
```

Errors start to appear at around the seventh decimal place.

20.5.1 Deriving the Polynomial Coefficients.

The majority of the polynomials calculated in Hart's *Computer Approximations* utilize the Remez exchange algorithm. While the power series version of that algorithm is fairly straightforward to implement, the rational Remez requires non-linear optimization. The only Python library I could find that offered both forms was Baryrat (`https://github.com/c-f-h/baryrat`), which actually utilizes a modern replacement for Remez.

For the power series Remez used by `cosD()`, there are several Python implementations to choose from. I went with `py-remezfit` (`https://github.com/emece67/py-remezfit`), which packages everything into a single file, although it also relies on numpy and scipy. The following call generates 4th-order power series coefficients for cosine in the range $[0, \pi/2)$, with x^2 as the polynomial argument:

```
import numpy as np
from remezfit import remezfit

coefs, prec, _ = remezfit(lambda x: np.cos(x),
                    0, np.pi/2, 4, arg = lambda x: x*x,
                    dtype = np.single)
print("Coefs:", coefs)
print("Correct Digits:", prec)
```

The output:

```
Coefs: [ 9.9999994e-01 -4.9999911e-01  4.1663747e-02
        -1.3854750e-03  2.3172470e-05]
Correct Digits: 1.2458856141206454e-07
```

The generated coefficients aren't quite the same as Hart's 3502 table, probably due to `py-remezfit` using a modern variant of Remez, but closer than the Maclaurin series in Table 20.2.

20.6 The Maths of Function Approximation

The preceding couple of sections focused on functions from Allison's "Quick and Dirty Functions for BASIC" article as concrete examples of the utility of function approximation. However, we didn't address the mathematics behind the generation of the coefficients, instead just utilizing values prepared for us by John F. Hart and his co-authors [**Har68**].

In the following sections, the maths behind the Taylor series and Lagrange polynomials will be considered, followed by the Chebyshev approximation. Chebyshev offers near optimal behavior while being much simpler to understand than the Remez algorithm favored by Hart.

For rational polynomials, we'll consider Padé approximation, which performs well near singularities, and, aside from its reliance on a linear system solver, is much less complicated than the rational Remez method.

20.7 The Taylor Series

Perhaps the simplest way of obtaining an approximation to a function $f(x)$ is via its Taylor series expansion. Suppose you want to approximate $f(x)$ over the interval $[a, b]$, centered at some $x_0 \in [a, b]$, then Taylor's theorem states:

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2} f''(x_0) + \cdots + \frac{(x - x_0)^n}{n!} f^{(n)}(x_0) + \cdots$$

Maclaurin's series is a special case when $x_0 = 0$:

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + \cdots$$

Some examples:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots & x \in \mathbb{R} \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots & x \in \mathbb{R} \\ \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots & x \in \mathbb{R} \end{aligned}$$

However, many series place limits on the x-range, such as:

Function	Maclaurin Series	Input Range	Radius R
$\frac{1}{1+x}$	$1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + \cdots$	$\lvert x\rvert < 1$	1
$\frac{1}{1+x^2}$	$1 - x^2 + x^4 - x^6 + x^8 - x^{10} + x^{12} - x^{14} + \cdots$	$\lvert x\rvert < 1$	1
$\ln(1 + x)$	$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \frac{x^6}{6} + \frac{x^7}{7} - \frac{x^8}{8} + \cdots$	$-1 < x \leq 1$	1
$\sqrt{1 + x}$	$1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{5x^4}{128} + \cdots$	$-1 \leq x \leq 1$	1
$\tan(x)$	$x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \cdots$	$-\frac{\pi}{2} < x < \frac{\pi}{2}$	$\pi/2$
$\sec(x)$	$1 + \frac{x^2}{2} + \frac{5x^4}{24} + \frac{61x^6}{720} + \cdots$	$-\frac{\pi}{2} < x < \frac{\pi}{2}$	$\pi/2$
$\arctan(x)$	$x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \frac{x^{11}}{11} + \frac{x^{13}}{13} - \cdots$	$-1 \leq x \leq 1$	1

For an even longer list, see section 1.3 of Zwillinger's *CRC Standard Mathematical Tables and Formulae* [**Zwi96**] (a copy is available at the Internet Archive

at https://archive.org/details/crcstandardmathe00chem/page/40/mode/2up).

The input range restrictions depend on the distance of the expansion point ($x_0 = 0$) from the nearest singularity, its *radius of convergence*. There's no point passing an x value to a series outside its prescribed range because it won't converge to an answer. Code implementing the series will loop forever unless there's some iteration limit, but the returned value will be meaningless.

By far the easiest way to implement Taylor series in Python is with the Sympy library for symbolic mathematics (https://www.sympy.org/). Expressions can be manipulated algebraically, and Sympy's `series` module (https://docs.sympy.org/latest/modules/series/series.html#sympy.series.series.series) is ideal for coding the previous examples (see seriesGen.py):

```
import math
import sympy as sp

fNames = ["1/(1+x)", "exp(x)", "cos(x)", "sin(x)",
          "log(1+x)", "atan(x)" ]

spFns = [lambda x: 1/(1+x), sp.exp, sp.cos, sp.sin,
         lambda x: sp.log(1+x), sp.atan ]

x = sp.Symbol("x")
for i, fnm in enumerate(fNames):
  print(f"{fnm}")
  series = sp.series(spFns[i](x), x0=0, n=8, dir='-')  # .removeO()
  print(" ", series)
  print()
```

The `x0` argument in the `series()` call sets the expansion center, while `n` is the maximum term power. The output:

```
1/(1+x)
  1 - x + x**2 - x**3 + x**4 - x**5 + x**6 - x**7 + O(x**8)

exp(x)
  1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 + x**6/720 +
   x**7/5040 + O(x**8)

cos(x)
  1 - x**2/2 + x**4/24 - x**6/720 + O(x**8)

sin(x)
  x - x**3/6 + x**5/120 - x**7/5040 + O(x**8)

log(1+x)
  x - x**2/2 + x**3/3 - x**4/4 + x**5/5 - x**6/6 +
```

```
  x**7/7 + O(x**8)

atan(x)
  x - x**3/3 + x**5/5 - x**7/7 + O(x**8)
```

The "O()" at the end of each expansion gives the order of the next term in the series.

faUtils.py contains two functions for utilizing Sympy series – `seriesCoefs()` and `evalPowers()`, which are employed below to generate the Maclaurin series for e^x and to calculate e^2.

```
>>> import sympy as sp
>>> from faUtils import *
>>> coefs  = seriesCoefs(sp.exp, verbose=True)
   Poly(1/362880*_u**9 + 1/40320*_u**8 + 1/5040*_u**7 + 1/720*_u**6 +
   1/120*_u**5 + 1/24*_u**4 + 1/6*_u**3 + 1/2*_u**2 + _u + 1, _u,
   domain='EX')
Coefs in ascending power order:
   [1, 1, 1/2, 1/6, 1/24, 1/120, 1/720, 1/5040, 1/40320, 1/362880]

>>> evalPowers(coefs, 2)
7.3887125220458545
>>> import math   # for comparison
>>> math.exp(2)
7.38905609893065
```

The series value for e^2 is only 'fairly' close to the one produced by `math.exp()`, with the discrepancy due to a combination of round-off and truncation errors, which I'll consider next.

20.7.1 Truncation Errors.

Truncation errors are inherent in the nature of series. For instance, if we only use the first four terms of e^x to calculate e^2, then we only get

$$e^2 \approx 1 + 2 + \frac{2^2}{2!} + \frac{2^3}{3!} = 6.3333.$$

A function $f(x)$ can be written as an n-term Taylor series plus a truncation error:

$$\begin{aligned} f(x) &= T_n(x) + E_n(x) \\ &= \textstyle\sum_{k=0}^{n} \frac{f^{(k)}(x_0)(x-x_0)^k}{k!} + E_n(x) \end{aligned}$$

x_0 is the expansion point, and $f^{(k)}(x)$ is the k-th derivative of $f(x)$.

We can obtain a value for $E_n(x)$ by utilizing the property that if $f(x)$ has $n+1$ derivatives in an interval $[a, b]$ containing x_0, then for each x in $[a, b]$ there exists a z between x and x_0 such that

$$E_n(x) = \frac{f^{(n+1)}(z)(x-x_0)^{n+1}}{(n+1)!}.$$

If we know that M is the maximum value of $|f^{(n+1)}(z)|$ in the interval, then

$$|E_n(x)| \le \frac{M|x - x_0|^{n+1}}{(n+1)!}.$$

Let's employ this bound to estimate the error for e^2 (i.e. when $x = 2$) for a Maclaurin series with $n = 9$ terms:

$$E_n(x) = \frac{f^{(9+1)}(z)(x)^{(9+1)}}{(9+1)!} = \frac{e^z 2^{10}}{10!}.$$

z is between $x_0 = 0$ and 2, and let's say that $e \approx 3$. Therefore,

$$|E_n(x)| \le \frac{3^2 2^{10}}{10!} \approx 0.00254.$$

Returning to the Python session from earlier, this estimate turns out to be too pessimistic:

```
>>> e2Approx = evalPowers(coefs, 2)
>>> e2Approx
7.3887125220458545
>>> abs(e2Approx - math.exp(2))
0.0003435768847959153
```

The Taylor series definition seems to suggest that it's always better to increase the number of terms in order to reduce the truncation error. A visual depiction of the benefit is shown in Fig. 20.10.

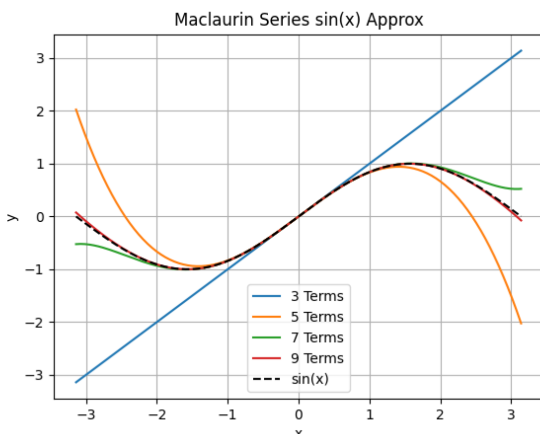


Figure 20.10. Adding More Terms to a Maclaurin Series for $\sin(x)$

sinMac.py generates four versions of the Maclaurin series for $\sin(x)$ with increasing numbers of terms, and plots them against `math.sin(x)` between $-\pi$

and π. Fig. 20.10 highlights how the accuracy of the series around $x_0 = 0$ increases as the number of terms increases, but also shows that it tails off rapidly. sinFar.py examines this property over a wider range (see Fig. 20.11).

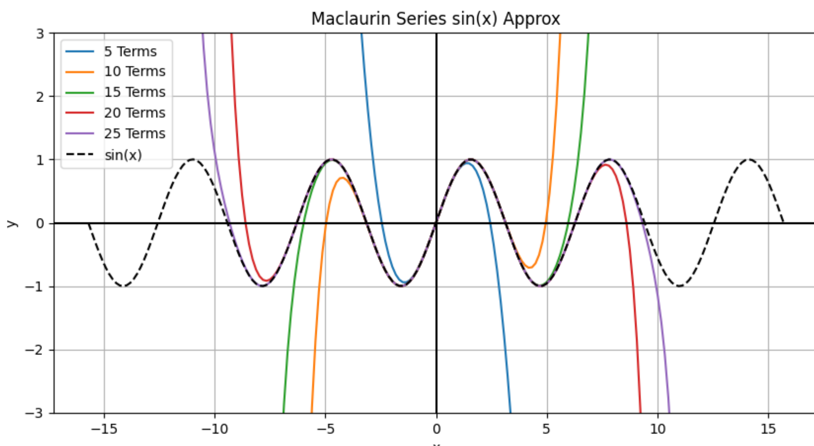


Figure 20.11. A Maclaurin Series for $\sin x$ between -5π and 5π

Adding more terms definitely improves the span of the accuracy, but the number of terms quickly becomes excessive. Also, their coefficients become exceedingly small, raising the issue of rounding errors when they're used in calculations. For example, the coefficients for the 25-term version of $\sin(x)$ are:

```
Using 25 terms:
Coefs in ascending power order:
[0, 1, 0, -1/6, 0, 1/120, 0, -1/5040, 0, 1/362880, 0, -1/39916800, 0,
1/6227020800, 0, -1/1307674368000, 0, 1/355687428096000, 0,
-1/121645100408832000, 0, 1/51090942171709440000, 0,
-1/25852016738884976640000, 0 ]
```

20.7.2 Round-off Errors. Fig. 20.12 shows Maclaurin series expansions for e^{-x} as generated by expFar.py.

As with $\sin(x)$, version of the e^{-x} series with a large number of terms have very small coefficients:

```
Using 25 terms:
Coefs in ascending power order:
[1, -1, 1/2, -1/6, 1/24, -1/120, 1/720, -1/5040, 1/40320, -1/362880,
1/3628800, -1/39916800, 1/479001600, -1/6227020800, 1/87178291200,
-1/1307674368000, 1/20922789888000, -1/355687428096000,
1/6402373705728000, -1/121645100408832000, 1/2432902008176640000,
-1/51090942171709440000, 1/1124000727777607680000,
-1/25852016738884976640000,  1/620448401733239439360000 ]
```

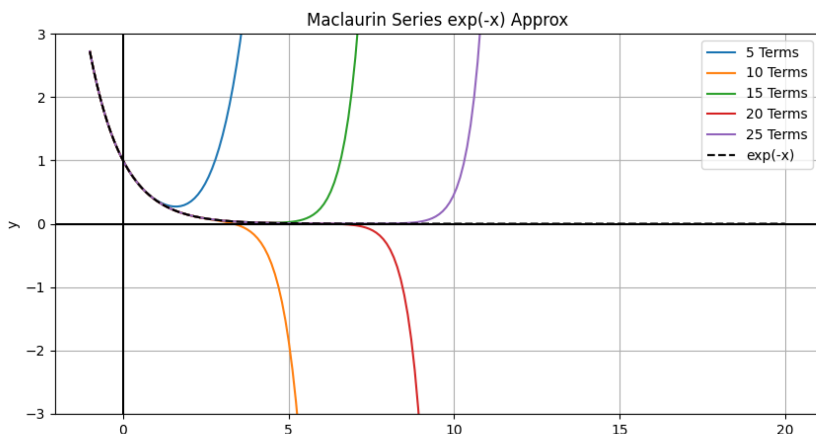


Figure 20.12. A Maclaurin Series for e^{-x} between -1 and 20

Numerical evidence for the resulting rounding errors is produced by expRound.py which calls `expApprox()` to evaluate n terms of e^x:

```
def expApprox(x, n):
  exp = 0
  for i in range(n+1):
    exp += (x**i)/math.factorial(i)
  return exp
```

The program output below shows the result of passing $x = -10$ and $x = -30$ to `expApprox()` to calculate e^{-10} and e^{-30} with different n's. `math.exp()` is also called for the sake of comparison:

```
x == -10
Math e^-10 == 4.53999e-05
Using 10 terms, approx == 1.34259e+03
Using 200 terms, approx == 4.53999e-05

x == -30
Math e^-30 == 9.35762e-14
Using 200 terms, approx == -8.55302e-05
Using 1000 terms, approx == -8.55302e-05
```

A reasonable answer for e^{-10} is produced, but only when using 200 terms. However, rounding errors means that a good result for e^{-30} is out of reach.

The problem is exacerbated by evaluating e^x with negative values which causes the series to add and subtract alternating terms (i.e. $e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$). As soon as a calculation involves the subtraction of similar-sized values then the likelihood of rounding errors increases substantially.

We've already seen the best solution to this problem in dirty.py: the input should be constrained to a narrow range, which in the Taylor series case should be around its x_0 value. `expTaylor()` in expApprox.py implements that strategy:

```
LN2 = math.log(2)   # 0.6931471805599453094

def expTaylor(x, maxIters=16, eps=1e-15):
  if x == 0.0:
    return 1.0

  k = int(round(x/LN2))
  xR = x - k*LN2  # subtract nearest multiple
  term = 1.0
  tot = 1.0
  for n in range(1, maxIters):
    term *= xR/n
    tot += term
    if abs(term) <= eps * max(1.0, abs(tot)):
      # print(n)   # usually about 12
      break
  return tot * (2.0**k)
```

The input is reduced by subtracting the nearest k-th multiple of ln(2) to guarantee that `xR` is in the range $[-\ln(2)/2, \ln(2)/2]$. After the series for e^{xR} has been calculated (using incremental term building rather than expensive calls to `math.factorial()`), then the result for the original x input is constructed with $e^x = 2^k * e^{\mathrm{xR}}$.

The test-rig in expApprox.py compares the results of `expTaylor(x)` against `math.exp(x)` for a range of values:

```
    x       Approx      math.exp(x)     Error
--------------------------------------------------

  -100.0   3.72008e-44   3.72008e-44   1.740e-15
   -10.0   4.53999e-05   4.53999e-05   5.970e-16
    -1.0   3.67879e-01   3.67879e-01   1.509e-16
    -0.5   6.06531e-01   6.06531e-01   3.661e-16
     0.5   1.64872e+00   1.64872e+00   1.347e-16
     1.0   2.71828e+00   2.71828e+00   1.634e-16
    10.0   2.20265e+04   2.20265e+04   3.303e-16
   100.0   2.68812e+43   2.68812e+43   1.289e-15
```

20.7.3 The Problem of Singularities. As mentioned at the start of this section, many Taylor series have a restricted input range, and a series will only converge when given a value inside that range. The range is determined by the distance of the expansion point (x_0) from the nearest singularity, known as its radius of convergence.

This constraint is made more complicated by the fact that while a singularity *may* be drawn as an asymptote when the curve is plotted, the radius is actually located in the complex plane, and so may not be visible (e.g. see. Sec. 20.7.4 below).

Even when the asymptote is visible, the radius is more restrictive than the graph of the curve would suggest. For example, consider $1/(1+x)$ and $\ln(1+x)$ in Fig. 20.13:

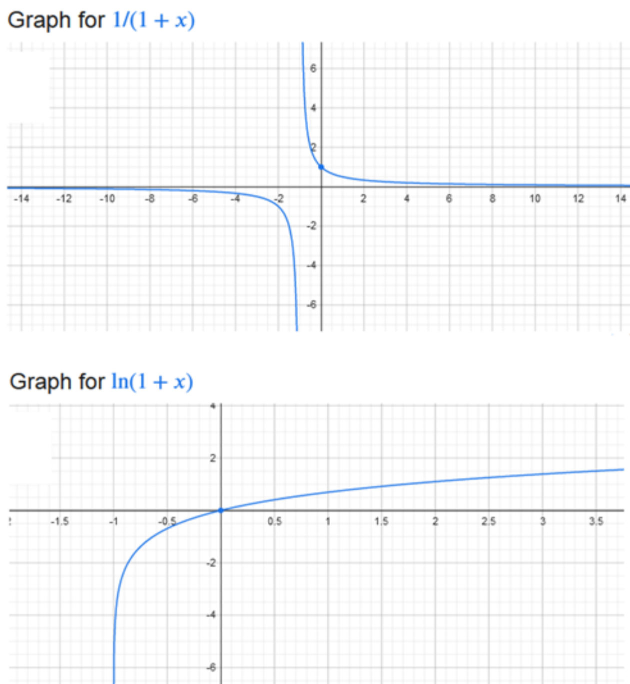


Figure 20.13. Graphs of $1/(1+x)$ and $\ln(1+x)$

$1/(1+x)$ has an asymptote at $x = -1$ which heads towards $\pm\infty$ depending on the direction of approach, while $\ln(1+x)$ has a pole at $x = -1$. The Maclaurin series for $\ln(1+x)$ should clearly not converge for values of $x \leq -1$, but its positive values are also constrained (to the range $-1 < x \leq 1$). The same question arises for $1/(1+x)$ (its range is $|x| < 1$), with the additional issue of why we can't pass larger negative numbers to the series (e.g. $x = -5$).

logCat.py generates five versions of the Maclaurin series for $\ln(1+x)$ with increasing numbers of terms, and plots them against `math.log()` in Fig. 20.14.

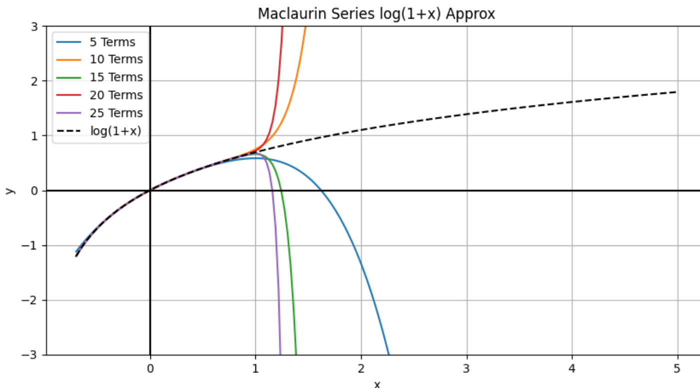


Figure 20.14. Maclaurin Series for $\ln(1+x)$ with Different Numbers of Terms

The range restriction of $-1 < x \leq 1$ makes it impossible to get an accurate answer beyond $x = 1$ no matter how many terms are thrown at the problem.

One way around this limitation is to use identities to rewrite the equation, a strategy we first saw in dirty.py. Recall that `log10D()` utilizes the hyperbolic function $\text{atanh}(x)$. Its Maclaurin series is

$$\text{atanh}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \cdots$$

Its radius of convergence is $R = 1$, restricting it to the range $-1 < x < 1$, which seems to offer no improvements over sticking with $\ln(1+x)$. However, the $\text{atanh}(x)$ series only employs addition, so avoids the cancellation errors related to subtraction that affect $\ln(1+x)$.

`logApprox()` in logApprox.py implements this approach:

```
LN2 = 0.6931471805599453094
SQRT2 = 1.4142135623730950488
SQRT_HALF = SQRT2/2         # sqrt(2/4) == sqrt(2)/2

def logApprox(x, maxIters=16, eps=1e-15):
  if x <= -1.0:
    raise ValueError("logApprox() requires x > -1")
  if x == 0.0:
    return 0.0

  m = 1.0 + x; k = 0
  # m is scaled into the interval
  # sqrt(1/2) <= m <= sqrt(2)
  while m > SQRT2:
    m *= 0.5; k += 1
```

```
  while m < SQRT_HALF:
    m *= 2; k -= 1
  r = m - 1
  z = r/(2 + r)
  z2 = z*z

  term = z; tot = z    # for atanh series
  denom = 3
  for _ in range(maxIters):
    term *= z2
    delta = term/denom
    tot += delta
    if abs(delta) <= eps * max(1.0, abs(tot)):
      break
    denom += 2
  return k*LN2 + 2.0*tot
```

`logApprox()` scales its input before evaluating the atanh() series. The reduction is based on the equality $1 + x = m \cdot 2^k$ which maps x to m in the interval $\sqrt{1/2} \leq m \leq \sqrt{2}$. Then $r = m - 1$ is used to shift the range to either side of 0, and r is utilized in the ln() = atanh() identity:

$$\ln(1 + r) = 2 \cdot \operatorname{atanh}(r/(2 + r))$$

Another simplification is to let $z = r/(2 + r)$, reducing the series to

$$\operatorname{atanh} z = z + z^3/3 + z^5/5 + z^7/7 + \cdots$$

so that

$$\ln(1 + r) = 2 \cdot (z + z^3/3 + z^5/5 + \cdots)$$

All of these renamings and scalings end up placing $|z| \leq 0.171573\ldots$, which is well within the convergence range of the atanh() series.

At the end of `logApprox()`, its result for the original x input is constructed using

$$\ln(1 + x) = (k \cdot \log 2) + \log m.$$

The test-rig in logApprox.py compares the results of `logApprox(x)` against `math.log(1+x)` (actually `math.log1p(x)`):

```
       x         Approx   math.log1p(x)   Rel Error
----------------------------------------------------

 -0.999999  -1.38155e+01  -1.38155e+01   0.000e+00
    -0.999  -6.90776e+00  -6.90776e+00   0.000e+00
     -0.99  -4.60517e+00  -4.60517e+00   1.929e-16
      -0.1  -1.05361e-01  -1.05361e-01   3.952e-16
     -0.01  -1.00503e-02  -1.00503e-02   8.630e-16
    -0.001  -1.00050e-03  -1.00050e-03   8.669e-16
     0.001   9.99500e-04   9.99500e-04   1.100e-13
```

```
  0.01   9.95033e-03   9.95033e-03   1.046e-15
   0.1   9.53102e-02   9.53102e-02   7.280e-16
     1   6.93147e-01   6.93147e-01   0.000e+00
    10   2.39790e+00   2.39790e+00   1.852e-16
   100   4.61512e+00   4.61512e+00   1.924e-16
 1e+06   1.38155e+01   1.38155e+01   0.000e+00
```

20.7.4 Invisible Asymptotes. Since asymptotes can occur in the complex plane for a function, they're not necessarily visible when that function is plotted in the real plane. This is illustrated by $1/(1+x^2)$, shown in Fig. 20.15.

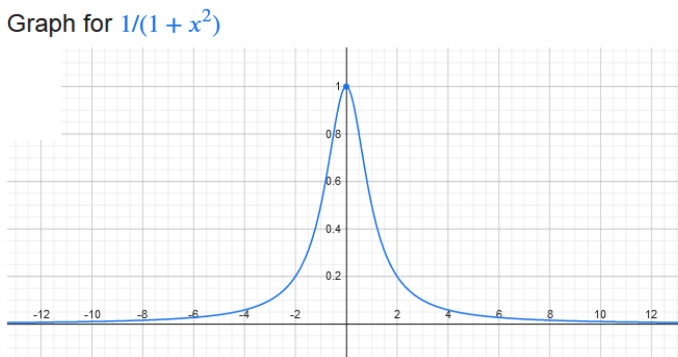


Figure 20.15. $1/(1+x^2)$ Graphed

Its Maclaurin series is:

$$\frac{1}{1+x^2} = 1 - x^2 + x^4 - x^6 + x^8 - x^{10} + x^{12} - x^{14} + \cdots,$$

and has a radius of convergence $R = 1$, limiting is input range to $|x| < 1$. The reason becomes slightly clearer if we factor the function's denominator: $1 + x^2 = (x - i)(x + i)$. Singularities occur when $x = \pm i$, and the function's radius is the distance from the origin to the closest of those points.

limitRadius.py repeatedly plots the Maclaurin series for $1/(1+x^2)$ between $x = -5$ and 5, each time with more terms. Fig. 20.16 shows that the convergence limits become more sharply defined, but never reliably reach beyond the $|x| < 1$ boundaries.

There's nothing we can do to change the singularities but a Taylor series can utilize any expansion point. In movedCenter.py, $1/(1+x^2)$ is again plotted between $x = -5$ and 5, but expanded around $x = 2$. The code changes are minimal:

```
for n in [4, 5, 8, 9]:
  print(f"\nUsing {n} terms:")
  fCoeffs = faUtils.seriesCoefs(fn, x0=2, nTerms=n,
```

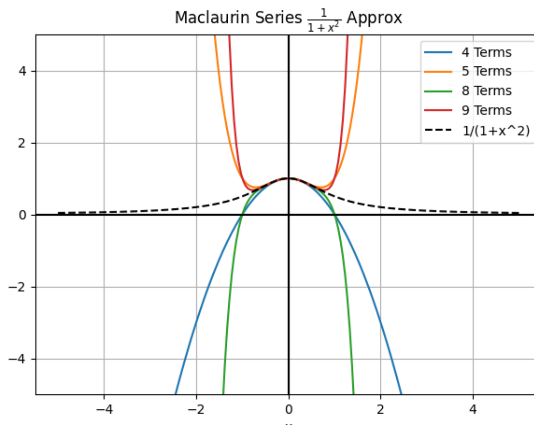


Figure 20.16. Plots of Maclaurin Series for $1/(1+x^2)$

```
                                verbose=True)
ys = [ faUtils.evalPowers(fCoeffs, x, x0=2) for x in xs]
plt.plot(xs, ys, label= f"{n} Terms at x0=2")
```

The four plots are shown in Fig. 20.17.

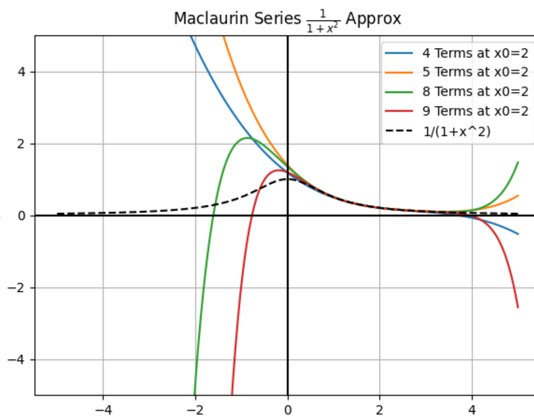


Figure 20.17. Plots of Taylor Series at $x_0 = 2$ for $1/(1+x^2)$

To expand $f(x) = 1/(1+x^2)$ about $x = 2$, let $h = x - 2$, and substitute it into the equation:

$$f(x) = \frac{1}{1+(2+h)^2} = \frac{1}{5+4h+h^2}.$$

Factoring out 5,

$$f(x) = \frac{1}{5}\left(\frac{1}{1 + {}^4\!/_5 h + {}^1\!/_5 h^2}\right).$$

The right-hand side can be expanded by using

$$\frac{1}{1+u} = 1 - u + u^2 - u^3 + \cdots, \qquad |u| < 1,$$

if we set

$$u = \frac{4}{5}h + \frac{1}{5}h^2.$$

After collecting powers of h, we get

$$f(x) = \frac{1}{5} - \frac{4}{25}(x-2) + \frac{11}{125}(x-2)^2 - \frac{24}{625}(x-2)^3 + \frac{41}{3125}(x-2)^4 + \cdots$$

The polynomial coefficients generated by `Sympy.series()` in the code above are reassuringly the same:

```
Using 5 terms:
   Poly(41/3125*_u**4 - 24/625*_u**3 + 11/125*_u**2 -
    4/25*_u + 1/5, _u, domain='EX')
Coefs in ascending power order:
   [1/5, -4/25, 11/125, -24/625, 41/3125]
```

The singularities for $1/(1+x^2)$ are still at $x = \pm i$, but the radius of convergence is now the shortest distance from the expansion point at $x_0 = 2$. Interpreting the numbers as points in the complex plane gives us:

$$2 = (2,0), \qquad i = (0,1), \qquad -i = (0,-1).$$

To which we apply the distance formula,

$$|2-i| = \sqrt{(2-0)^2 + (0-1)^2} = \sqrt{4+1} = \sqrt{5}$$

and

$$|2+i| = \sqrt{(2-0)^2 + (0+1)^2} = \sqrt{4+1} = \sqrt{5}.$$

Since both distances are the same, $R = \sqrt{5}$, so the series converges for $|x-2| < \sqrt{5}$. In other words, $x \in [0.236..., 4.236...]$, which is (roughly) confirmed by the plots in Fig. 20.17.

20.7.5 Removable Singularities. A *removable* singularity means that the Taylor series for the function can be rearranged to remove the element that tends to infinity. Consider $\sin(x)/x$ whose denominator appears to send the function towards a singularity as x approaches 0. However, the Taylor series for $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots$, so the denominator can be divided into each term to give $\sin(x)/x = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \frac{x^8}{9!} - \ldots$.

An interactive Python session confirms these coefficients:

```
>>> import sympy as sp
>>> from faUtils import *
>>> f = lambda x: sp.sin(x)/x
>>> coefs = seriesCoefs(f, verbose=True)
   Poly(1/362880*_u**8 - 1/5040*_u**6 + 1/120*_u**4 -
   1/6*_u**2 + 1, _u, domain='EX')
Coefs in ascending power order:
   [1, 0, -1/6, 0, 1/120, 0, -1/5040, 0, 1/362880, 0]
```

20.8 The Lagrange Polynomial

The main disadvantage of using a Taylor series approximation is that it only approximates the function around a single expansion point. This gives it excellent local accuracy, but this decreases rapidly at points further away from x_0. The other issue is that the presence of singularities limit the series' range even for values which seem to be nowhere near the asymptote.

One solution is to move to function approximation that utilizes interpolation over a series of (x, y) coordinates. Perhaps the simplest approach of this type is the *Lagrange polynomial*.

Let's assume that we have $n+1$ numbers, $x_0, x_1, \dots, x_n$, and a function $f(x)$. The Lagrange polynomial is

$$P(x) = f(x_0)L_{n,0}(x) + \cdots + f(x_n)L_{n,n}(x) = \sum_{k=0}^{n} f(x_k)L_{n,k}(x),$$

where, for each $k = 0, 1, \dots, n$,

$$\begin{aligned} L_{n,k}(x) &= \frac{(x-x_0)(x-x_1)\dots(x-x_{k-1})(x-x_{k+1})\dots(x-x_n)}{(x_k-x_0)(x_k-x_1)\dots(x_k-x_{k-1})(x_k-x_{k+1})\dots(x_k-x_n)} \\ &= \prod_{\substack{i=0 \\ i\neq k}}^{n} \frac{(x-x_i)}{(x_k-x_i)}. \end{aligned}$$

Note that for $k = 0$, $L_{n,0}(x_0) = 1$ while for all the other x values $L_{n,0}(x_i) = 0$. Similarly, when $k = 1$, $L_{n,1}(x_1) = 1$ while $L_{n,1}(x_i) = 0$ for the other x's, and this property applies to all the $L_{n,k}()$ terms. If the coordinates are $(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$, then this implies that

$$\begin{array}{rcccl} P(x_0) &=& 1 \cdot f(x_0) + 0 \cdot f(x_1) + \cdots + 0 \cdot f(x_n) &=& y_0 \\ P(x_1) &=& 0 \cdot f(x_0) + 1 \cdot f(x_1) + \cdots + 0 \cdot f(x_n) &=& y_1 \\ \vdots & & \vdots & & \vdots \\ P(x_n) &=& 0 \cdot f(x_0) + 0 \cdot f(x_1) + \cdots + 1 \cdot f(x_n) &=& y_n \end{array}$$

So $P(x)$ passes through all of the supplied coordinates.

20.8.1 Approximating $1/x$. We have $x_0 = 2$, $x_1 = 2.75$, and $x_2 = 4$, and want to find the Lagrange polynomial for $f(x) = 1/x$. First, determine $L_{2,0}(x)$, $L_{2,1}(x)$, and $L_{2,2}(x)$:

$$\begin{aligned} L_{2,0}(x) &= \frac{(x-2.75)(x-4)}{(2-2.5)(2-4)} &= \frac{2}{3}(x-2.75)(x-4), \\ L_{2,1}(x) &= \frac{(x-2)(x-4)}{(2.75-2)(2.75-4)} &= -\frac{16}{15}(x-2)(x-4), \\ L_{2,2}(x) &= \frac{(x-2)(x-2.75)}{(4-2)(4-2.5)} &= \frac{2}{5}(x-2)(x-2.75). \end{aligned}$$

Also, $f(x_0) = f(2) = 1/2$, $f(x_1) = f(2.75) = 4/11$, and $f(x_2) = f(4) = 1/4$, so that

$$\begin{aligned} P(x) &= \sum_{k=0}^{2} f(x_k) L_{2,k}(x) \\ &= \frac{1}{3}(x-2.75)(x-4) - \frac{64}{165}(x-2)(x-4) + \frac{1}{10}(x-2)(x-2.75) \\ &= \frac{1}{22}x^2 - \frac{35}{88}x + \frac{49}{44}. \end{aligned}$$

It's interesting to compare this polynomial with a Taylor series for $1/x$. Let's create one centered on $x = 1$:

$$\frac{1}{x} = \frac{1}{1 + (x-1)}.$$

Expand the right-hand side using the geometric series for $1/(1+u)$ with $u = x-1$:

$$\frac{1}{x} = \sum_{n=0}^{\infty} (-1)^n (x-1)^n.$$

The first few terms are

$$\frac{1}{x} = 1 - (x-1) + (x-1)^2 - (x-1)^3 + (x-1)^4 - \cdots.$$

The radius of convergence is $R = 1$ since the expansion point is $x = 1$ and a singularity occurs at $x = 0$. Therefore, the series converges for $|x-1| < 1$, or equivalently when $0 < x < 2$. In other words, this Taylor series can not be used to approximate $f(x) = 1/x$ at $x = 3$ unlike the Lagrange.

Implementing a Lagrange Polynomial is easy if we utilize Sympy's `sympy.polys` module for polynomial algebras, and specifically its `interpolating_poly()` function (`https://docs.sympy.org/latest/modules/polys/reference.html`). It generates a Lagrange when supplied with n (x, y) data points, as shown in lagSymHyp.py, which replicates our $1/x$ example:

```
x = sp.symbols('x')
xs = [2, 2.75, 4]   # 0.3, 0.5
ys = [1/v for v in xs]
ipoly = interpolating_poly(len(xs), x, X=xs, Y=ys)
```

```
print("Poly:"); print(ipoly); print()

p = sp.expand(ipoly)   # multiply out all the terms
print("Expanded:", p)
print("Factored:, sp.factor(p))
print("Using Fractions:", sp.nsimplify(p))

# Convert to a Python function
pFunc = sp.lambdify(x, p, "math")
print("P(3) approx:", pFunc(3))
```

The output:

```
Poly:
0.333333333333333*(x - 4)*(x - 2.75) -
0.387878787878788*(x - 4)*(x - 2) +
0.1*(x - 2.75)*(x - 2)
Expanded:
0.0454545454545455*x**2 - 0.397727272727273*x + 1.11363636363636
Factored:
1.11363636363636*(0.0408163265306122*x**2 - 0.357142857142857*x + 1.0)
Using Fractions:
x**2/22 - 35*x/88 + 49/44

P(3) approx: 0.3295454545454505
```

lagSymHyp.py goes on to use the polynomial to generate a series of coordinates between $x = 1/10$ and 5, and plots them against $1/x$, as shown in Fig. 20.18.

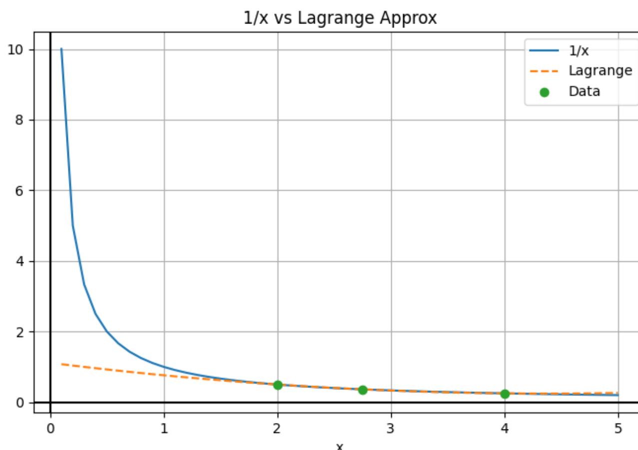


Figure 20.18. Lagrange Polynomial Plot for $1/x$ With 3 Data Points

The polynomial is a good fit within the interpolation range ($x_0 = 2$ to $x_2 = 4$), and passes through those points as expected. However, care must be taken when using the approximation outside the $[2, 4]$ interval, and this becomes clearer if we add $x = 0.3$ and 0.5, which produces the plot in Fig. 20.19.

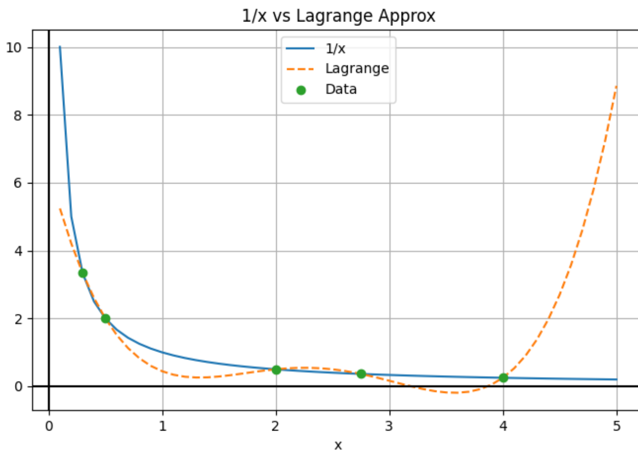


Figure 20.19. Lagrange Polynomial Plot for $1/x$ with 5 Data Points

The polynomial is now of degree 4:

```
10*x**4/33 - 191*x**3/66 + 95909090909091*x**2/10000000000000 -
130037878787879*x/10000000000000 + 64469696969697/10000000000000
```

Aside from the potential for large errors outside the interpolation range, the quality of the results *within* that range have also decreased. This can be remedied by adding more data points, but deciding how many to add and where they should be placed is tricky. Indeed, it may be better to simply return to the first polynomial, and accept a smaller range. This complexity is one reason for preferring Chebyshev polynomial approximation (see the next section), which avoids this problem.

20.8.2 The Lagrange Remainder Term. It's useful to be able to calculate an error term for a Lagrange polynomial.

Let $x_0, x_1, \dots, x_n$ be numbers in the interval $[a, b]$, then for each x, a z exists where

$$f(x) = P(x) + \frac{f^{(n+1)}(z)}{(n+1)!}(x - x_0)(x - x_1)\dots(x - x_n),$$

where $P(x)$ is the Lagrange polynomial, and the remainder is the error (called $R(x)$). It's similar to the error form we saw for the Taylor series:

$$\frac{f^{(n+1)}(z)}{(n+1)!}(x - x_0)^{n+1}.$$

Let's use the Lagrange remainder to find the maximum error in the $[2, 4]$ interval for the $1/x$ approximation. We have

$$f'(x) = -x^{-2}, \quad f''(x) = 2x^{-3}, \text{ and } f'''(x) = -6x^{-4}.$$

Therefore,

$$\begin{aligned} R(x) &= \frac{f'''(z)}{3!}(x - x_0)(x - x_1)(x - x_2) \\ &= -z^{-4}(x-2)(x-2.75)(x-4), \text{ for } z \text{ in } [2, 4]. \end{aligned}$$

The maximum value for z^{-4} in the interval is $2^{-4} = 1/16$, but what about the expression:

$$(x-2)(x-2.75)(x-4) = x^3 - \frac{35}{4}x^2 + \frac{49}{2}x - 22.$$

The cubic's critical points occur at $x = 7/3$ and $7/2$ as 'confirmed' by Fig. 20.20.

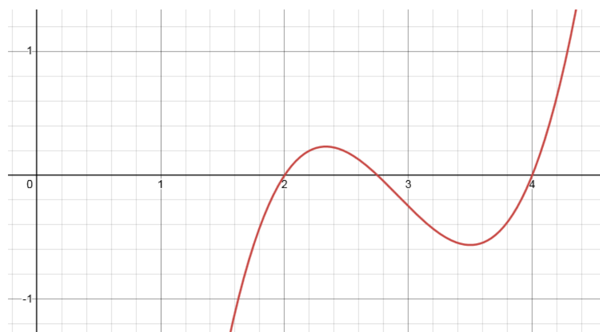


Figure 20.20. Critical Points in $(x-2)(x-2.75)(x-4)$

The corresponding y-values are $25/108$ and $-9/16$, with the latter being bigger in magnitude. Combining these results,

$$\begin{aligned} R(x) &= \left|\frac{f'''(\xi(x))}{3!}(x - x_0)(x - x_1)(x - x_2)\right| \leq \frac{1}{16 \cdot 6}\left|-\frac{9}{16}\right| \\ &= \frac{3}{512} \approx 0.00586. \end{aligned}$$

20.8.3 Approximating $\sin(x)$.

Let $f(x) = \sin(x)$, $n = 3$, and $x_k = k\pi/3$ for $k = 0, 1, 2, 3$. The $L_{n,k}$ values are:

$$L_{3,0}(x) = \frac{x-(\pi/3)}{0-(\pi/3)} \cdot \frac{x-(2\pi/3)}{0-(2\pi/3)} \cdot \frac{x-\pi}{0-\pi}$$
$$= \frac{1}{2\pi^3}(\pi-3x)(2\pi-3x)(\pi-x)$$

$$L_{3,1}(x) = \frac{9}{2\pi^3}x(2\pi-3x)(\pi-x)$$

$$L_{3,2}(x) = -\frac{9}{2\pi^3}x(\pi-3x)(\pi-x)$$

$$L_{3,3}(x) = \frac{1}{2\pi^3}x(\pi-3x)(2\pi-3x)$$

Utilizing these in the Lagrange polynomial:

$$P(x) = \sin(0) \cdot L_{3,0}(x) + \sin(\pi/3) \cdot L_{3,1}(x) + \sin(2\pi/3) \cdot L_{3,2}(x) + \sin(\pi) \cdot L_{3,3}(x)$$
$$= \frac{\sqrt{3}}{2} \cdot \frac{9}{2\pi^3}x(2\pi-3x)(\pi-x) - \frac{\sqrt{3}}{2} \cdot \frac{9}{2\pi^3}x(\pi-3x)(\pi-x)$$
$$= \frac{9\sqrt{3}}{4\pi^2}x(\pi-x)$$

The error bound is:

$$|R(x)| = \left|\frac{\sin z}{4!}(x-0)\left(x-\frac{\pi}{3}\right)\left(x-\frac{2\pi}{3}\right)(x-\pi)\right|$$
$$\le \frac{1}{24}\max_{0\le x\le\pi}\left|x\left(x-\frac{\pi}{3}\right)\left(x-\frac{2\pi}{3}\right)(x-\pi)\right|$$
$$= \frac{1.20258}{24} = 0.050108$$

lagSymSine.py implements this polynomial in code:

```
xs = [k*sp.pi/3 for k in range(4)]
ys = [sp.sin(v) for v in xs]
ipoly = interpolating_poly(len(xs), x, X=xs, Y=ys)
```

and its factored form is reported as:

```
-9*sqrt(3)*x*(x - pi)/(4*pi**2)
```

which agrees with our calculations.

The code also plots the polynomial against $\sin(x)$ in the interval $[0, \pi]$ and draws a second graph showing how the absolute error varies between the two curves (see Fig. 20.21).

The error plot shows that the calculated maximum error is a little too large since the polynomial's error tops out at around 0.045.

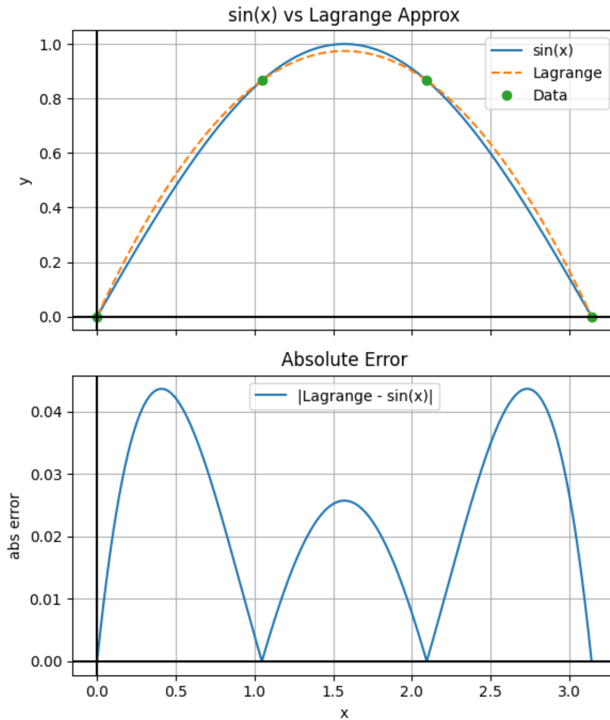


Figure 20.21. Lagrange Polynomial Plot for $sin(x)$

20.8.4 Lagrange Approximation Without Sympy. If there's no need for a polynomial in algebraic form, then it's possible to calculate the polynomial's result using summations over $L_{n,k}()$s expressed as products. This approach is implemented by `lagrangeInterp()` in faUtils.py:

```
def lagrangeInterp(xs, ys, x):
  # compute the Lagrange polynomial at x
  n = len(xs)
  total = 0.0
  for k in range(n):
    term = ys[k]
    for j in range(n):  # 0 to n except for k
      if k != j:
        term *= (x - xs[j]) / (xs[k] - xs[j])
    total += term
  return total
```

This function is used by lagrangeSine.py to replicate the $\sin(x)$ approximation in lagSymSine.py. The crucial lines are:

```
xs = [k*math.pi/3 for k in range(4)]
ys = [math.sin(x) for x in xs]
xPlot = faUtils.linspace(0, math.pi, 200)
yPlot = [faUtils.lagrangeInterp(xs, ys, x) for x in xPlot]
```

20.9 Orthogonality

Least squares (see section 6.8) can fit a function approximation to a collection of data points, but the matrix that needs to be solved can be prone to rounding errors. However, if we assume that the polynomial approximation is *orthogonal* then the calculations become much simpler and less likely to produce errors. Orthogonality is the core element that makes the Chebyshev approximation of the next section so successful, so we'll spend a little time here explaining the concept.

Suppose $f(x)$ is to be approximated by a polynomial $P_n(x)$ of at most n degree across the x range $[a, b]$. We want to minimize the least squares error

$$\int_a^b [f(x) - P_n(x)]^2 \, dx.$$

Let

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = \sum_{k=0}^{n} a_k x^k,$$

where we want to determine $a_0, a_1, \cdots, a_n$ subject to error minimization

$$E \equiv E_2(a_0, a_1, \ldots, a_n) = \int_a^b \left(f(x) - \sum_{k=0}^{n} a_k x^k \right)^2 dx.$$

A necessary condition is that

$$\frac{\partial E}{\partial a_j} = 0, \quad \text{for each } j = 0, 1, \ldots, n.$$

After expanding the right-hand side of E:

$$E = \int_a^b [f(x)]^2 \, dx - 2 \sum_{k=0}^{n} a_k \int_a^b x^k f(x) \, dx + \int_a^b \left(\sum_{k=0}^{n} a_k x^k \right)^2 dx,$$

we partially differentiate it with respect to each a_j:

$$\frac{\partial E}{\partial a_j} = 0 = -2 \int_a^b x^j f(x) \, dx + 2 \sum_{k=0}^{n} a_k \int_a^b x^{j+k} \, dx.$$

Hence, to find $P_n(x)$, we solve $(n+1)$ a_k equations involving $(n+1)$ j powers:

$$\sum_{k=0}^{n} a_k \int_a^b x^{j+k} \, dx = \int_a^b x^j f(x) \, dx, \quad \text{for each } j = 0, 1, \ldots, n.$$

Let's define the right-hand sides as:

$$b_j = \int_a^b x^j f(x)\,dx, \quad \text{for each } j = 0, 1, \dots, n.$$

and evaluate the integrals on the left-hand sides. The resulting system can be written as a linear system of the form

$$\underbrace{\left[\frac{b^{k+j+1} - a^{k+j+1}}{k+j+1}\right]_{k,j=0}^{n}}_{C} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix}}_{\mathbf{b}}$$

The $n+1$ square matrix C is a *Hilbert matrix*, and a classic example of how round-off errors affect results.

20.9.1 Linearly Independent Functions. Fortunately there's a different technique for obtaining least squares approximations that turns out to be much more computationally efficient. To facilitate the discussion, we need some new concepts.

A set of functions $\{\phi_0, \dots, \phi_n\}$ is said to be *linearly independent* over some range $[a, b]$ if, whenever

$$c_0\phi_0(x) + c_1\phi_1(x) + \cdots + c_n\phi_n(x) = 0, \quad \text{for all } x \in [a, b],$$

then we have $c_0 = c_1 = \cdots = c_n = 0$.

It can be proved that if $\phi_j(x)$ is a polynomial of degree j where $j = 0, 1, \dots, n$, then $\{\phi_0, \dots, \phi_n\}$ **is** linearly independent. Also, any other polynomial can be written as a linear combination of $\phi_0(x), \phi_1(x), \dots, \phi_n(x)$. As a rough analogy, linearly independent functions are like the x, y, and z unit vectors in 3D space, and combinations of those vectors can define any other vector.

As an example, consider the set $\beta = \{1, x+1, x^2+1\}$, which is linearly independent because the only solution to $c_1(1) + c_2(x+1) + c_3(x^2+1) = 0$ is $c_1 = c_2 = c_3 = 0$. Now write some arbitrary polynomial, $p(x) = 2x^2 + 3x + 1$, as a linear combination of the elements of β:

$$\begin{aligned} 2x^2 + 3x + 1 &= a_0(1) + a_1(x+1) + a_2(x^2+1) \\ &= (a_0 + a_1 + a_2) + a_1 x + a_2 x^2, \end{aligned}$$

and so $a_2 = 2$, $a_1 = 3$, and $a_0 = -4$.

20.9.2 Orthogonal Functions. Now we introduce weight functions, and define orthogonality.

The purpose of a weight function w is to assign varying degrees of importance to approximations over portions of the x interval. For example,

$$w(x) = \frac{1}{\sqrt{1 - x^2}}$$

places less emphasis near the center of the interval $(-1, 1)$ and more emphasis when $|x|$ is near 1.

Suppose $\{\phi_0, \phi_1, \dots, \phi_n\}$ is a set of linearly independent functions on $[a, b]$ and w is a weight function for $[a, b]$. We want to linearly combine these functions to define a polynomial $P(x)$

$$P(x) = \sum_{k=0}^{n} a_k \phi_k(x)$$

while also minimizing the error

$$E = E(a_0, \dots, a_n) = \int_a^b w(x) \left[f(x) - \sum_{k=0}^{n} a_k \phi_k(x) \right]^2 dx.$$

This problem reduces to the situation considered at the beginning of this section when $w(x) = 1$ and $\phi_k(x) = x^k$, for each $k = 0, 1, \dots, n$.

This reduces to the series of partial derivatives like the ones from earlier, except that $w(x)$ is now included:

$$\frac{\partial E}{\partial a_j} = 2 \int_a^b w(x) \left[f(x) - \sum_{k=0}^{n} a_k \phi_k(x) \right] \phi_j(x)\, dx.$$

Rearranging the equations gives

$$\int_a^b w(x) f(x) \phi_j(x)\, dx = \sum_{k=0}^{n} a_k \int_a^b w(x) \phi_k(x) \phi_j(x)\, dx, \quad \text{for } j = 0, 1, \dots, n.$$

Now is the time to introduce orthogonality, which will allow us to avoid solving a $n + 1$-square matrix in the next step. Orthogonality is the principle that the functions $\phi_0, \phi_1, \dots, \phi_n$ can be chosen so that

$$\int_a^b w(x) \phi_k(x) \phi_j(x)\, dx = \begin{cases} 0, & \text{when } j \neq k, \\ \alpha_j > 0, & \text{when } j = k, \end{cases}$$

This permits the equations to be greatly simplified, becoming

$$\int_a^b w(x) f(x) \phi_j(x)\, dx = a_j \int_a^b w(x) [\phi_j(x)]^2\, dx = a_j \alpha_j,$$

for each $j = 0, 1, \dots, n$, and are easily solved

$$a_j = \frac{1}{\alpha_j} \int_a^b w(x) f(x) \phi_j(x)\, dx.$$

A rough analogy of orthogonality is the dot product of vectors (see section 10.3). If two vectors are orthogonal (i.e. separated by 90°), then their dot product will be 0. This kind of property is desirable when choosing a set of linearly independent vectors to act as axes in 3D space.

Orthogonality is more restrictive than linear independence. As an example consider the following set of polynomials, which are linearly independent but not orthogonal:

$$p_0(x) = 1, \qquad p_1(x) = x, \qquad p_2(x) = x^2.$$

First, let's show that they're linearly independent. Suppose there are constants a, b, and c such that

$$a\, p_0(x) + b\, p_1(x) + c\, p_2(x) = 0$$

for all x. Substituting the polynomials into this equation gives $a + bx + cx^2 = 0$. The polynomial is zero only if all of its coefficients are zero, so the set $\{1, x, x^2\}$, is linearly independent. However, for a set to be orthogonal, every distinct pair must have an *inner product* $\langle p_i, p_j \rangle = 0$, where

$$\langle p_i, p_j \rangle = \int_{-1}^{1} p_i(x) p_j(x)\, dx.$$

Let's check this for 1 and x:

$$\langle 1, x \rangle = \int_{-1}^{1} x\, dx = 0.$$

What about x and x^2?

$$\langle x, x^2 \rangle = \int_{-1}^{1} x^3\, dx = 0.$$

But for 1 and x^2,

$$\langle 1, x^2 \rangle = \int_{-1}^{1} x^2\, dx = \left[\frac{x^3}{3} \right]_{-1}^{1} = \frac{2}{3},$$

so the set isn't orthogonal because of this pair.

It's not difficult to modify the set's elements, specifically x^2, so that the set becomes orthogonal. Let's define $p_2(x) = x^2 + a$, and once again check the inner products, but with the aim of choosing a value for a that makes the set orthogonal.

Test $\langle 1, x^2 + a \rangle$,

$$0 = \int_{-1}^{1} (x^2 + a)\, dx = \int_{-1}^{1} x^2\, dx + a \int_{-1}^{1} 1\, dx.$$

Evaluating the integrals,

$$0 = \frac{2}{3} + 2a.$$

Hence, $a = -1/3$, making $p_2(x) = x^2 - 1/3$, and the set becomes $\{1,\ x,\ x^2 - 1/3\}$. Its orthogonality should be checked:

$$\begin{aligned}
\langle 1, x\rangle &= \textstyle\int_{-1}^{1} x\,dx = 0.\\
\left\langle 1,\, x^2 - 1/3\right\rangle &= \textstyle\int_{-1}^{1}\left(x^2 - 1/3\right)dx = 2/3 - 2/3 = 0.\\
\left\langle x,\, x^2 - 1/3\right\rangle &= \textstyle\int_{-1}^{1}\left(x^3 - \frac{x}{3}\right)dx = 0.
\end{aligned}$$

Since all of the pairwise inner products vanish, the set is orthogonal.

20.9.3 The Orthogonality of $\cos 2\theta$ and $\cos 5\theta$.

Consider the inner product of these two functions in the interval $[0, 2\pi]$, which we'll show is equal to 0:

$$\int_0^{2\pi} \cos 2\theta\, cos5\theta\, d\theta = 0.$$

Rewrite both functions as polynomials in $\cos\theta$ by the use of multiple-angle identities,

$$\cos 2\theta = 2\cos^2\theta - 1$$

and

$$\cos 5\theta = 16\cos^5\theta - 20\cos^3\theta + 5\cos\theta.$$

Therefore,

$$\int_0^{2\pi} \cos 2\theta \cos 5\theta\, d\theta = \int_0^{2\pi} (2\cos^2\theta - 1)(16\cos^5\theta - 20\cos^3\theta + 5\cos\theta)\, d\theta.$$

Expanding,

$$\begin{aligned}
&= \int_0^{2\pi} \Big(32\cos^7\theta - 40\cos^5\theta + 10\cos^3\theta\\
&\qquad - 16\cos^5\theta + 20\cos^3\theta - 5\cos\theta\Big)\, d\theta\\
&= \int_0^{2\pi} \left(32\cos^7\theta - 56\cos^5\theta + 30\cos^3\theta - 5\cos\theta\right) d\theta.
\end{aligned}$$

Each term contains an odd power of $\cos\theta$:

$$\cos\theta, \quad \cos^3\theta, \quad \cos^5\theta, \quad \cos^7\theta$$

and for any odd power

$$\int_0^{2\pi} \cos^{2k+1}\theta\, d\theta = 0,$$

because the positive and negative halves cancel over a full period. Hence,

$$32\int_0^{2\pi} \cos^7\theta\, d\theta - 56\int_0^{2\pi} \cos^5\theta\, d\theta + 30\int_0^{2\pi} \cos^3\theta\, d\theta - 5\int_0^{2\pi} \cos\theta\, d\theta = 0.$$

Therefore, the integral equals 0 and so $\cos 2\theta$ and $\cos 5\theta$ are orthogonal. This is confirmed by ortho.pdf which calculates the dot product of multiple $\cos(2\theta)$ and $\cos(5\theta)$ values in the interval $[0, 2\pi]$:

```
n = 2; m = 5
nPts = 1000
thetas = [ (2*math.pi*i)/nPts for i in range(nPts)]
cosN = [math.cos(n*t) for t in thetas]
cosM = [math.cos(m*t) for t in thetas]
result = sum([a*b for a, b in zip(cosN, cosM)])
print(f"Dot product = {result:7f}")
```

Our reason for including this example is the importance of this cosine form for proving the orthogonality of Chebyshev polynomials in the next section (specifically subsection 20.10.2).

20.10 Chebyshev Approximation

The Chebyshev polynomial of degree n is defined as $T_n(x) = \cos(n \arccos(x))$, which, perhaps surprisingly, can be reexpressed with the help of trigonometric identities as powers of x,

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\dots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1. \end{aligned}$$

Crucially for their role in function approximation, these polynomials are orthogonal in the interval $[-1, 1]$ and, when combined with the weight function $1/\sqrt{1-x^2}$, simplify to:

$$\int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}}\, dx = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j = 0 \end{cases}$$

20.10.1 Deriving the $T_n(x)$ Polynomials. Note that

$$T_0(x) = \cos(0) = 1 \quad \text{and} \quad T_1(x) = \cos(\arccos(x)) = x.$$

For $n \geq 1$, we introduce the substitution $\theta = \arccos(x)$ so that

$$T_n(x) = \cos(n \arccos(x)) = \cos(n\theta), \quad \text{where } \theta \in [0, \pi].$$

The recurrence relation is derived by noting that

$$T_{n+1}(x) = \cos((n+1)\theta) = \cos(\theta)\cos(n\theta) - \sin(\theta)\sin(n\theta)$$

and

$$T_{n-1}(x) = \cos((n-1)\theta) = \cos(\theta)\cos(n\theta) + \sin(\theta)\sin(n\theta).$$

Adding these gives us

$$T_{n+1}(x) + T_{n-1}(x) = 2\cos(\theta)\cos(n\theta).$$

or

$$T_{n+1}(x) = 2\cos(\theta)\cos(n\theta) - T_{n-1}(x).$$

Replacing $\cos(\theta)$ by x and θ by $\arccos(x)$, we have for $n \geq 1$,

$$T_{n+1}(x) = 2x\cos(n\arccos(x)) - T_{n-1}(x),$$

that is,

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

Note that $\deg(T_n) = n$, and the recursion relation shows that the leading coefficient of the right-hand side of $T_n(x)$ is always 2^{n-1}.

20.10.2 Proving Orthogonality. To show the orthogonality of the Chebyshev polynomials, consider

$$\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}\,dx = \int_{-1}^{1} \frac{\cos(n\arccos(x))\cos(m\arccos(x))}{\sqrt{1-x^2}}\,dx.$$

Let's differentiate $\theta = \arccos(x)$:

$$d\theta = -\frac{1}{\sqrt{1-x^2}}\,dx$$

and so

$$\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}\,dx = -\int_{\pi}^{0} \cos(n\theta)\cos(m\theta)\,d\theta = \int_{0}^{\pi} \cos(n\theta)\cos(m\theta)\,d\theta.$$

This integral is a more general form of our section 20.9.3 example.

Now suppose that $n \neq m$, and since

$$\cos(n\theta)\cos(m\theta) = \frac{1}{2}\left[\cos(n+m)\theta + \cos(n-m)\theta\right],$$

we have

$$\begin{aligned}\int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}\,dx &= \frac{1}{2}\int_{0}^{\pi} \cos(n+m)\theta\,d\theta + \frac{1}{2}\int_{0}^{\pi} \cos(n-m)\theta\,d\theta \\ &= \left[\frac{1}{2(n+m)}\sin(n+m)\theta + \frac{1}{2(n-m)}\sin(n-m)\theta\right]_0^{\pi} = 0.\end{aligned}$$

By a similar technique, we can also show

$$\int_{-1}^{1} \frac{[T_n(x)]^2}{\sqrt{1-x^2}}\,dx = \frac{\pi}{2}, \quad \text{for each } n \geq 1.$$

You may want to check this, and also the $n = 0$ case.

The Chebyshev polynomials also satisfy a discrete form of orthogonality: if x_k ($k = 1, \ldots, n$) are the n zeros of $T_n(x)$, and if $i, j < n$, then

$$\sum_{k=1}^{n} T_i(x_k)T_j(x_k) = \begin{cases} 0 & i \neq j \\ n/2 & i = j \neq 0 \\ n & i = j = 0 \end{cases}$$

20.10.3 Zeros and Extrema. Two important properties of the Chebyshev polynomials are the distribution of their zeros, and their maximums and minimums.

$T_n(x)$ will always have n zeros in the interval $[-1, 1]$, located at the points

$$x_k = \cos(\frac{\pi(k - 1/2)}{n}) \qquad k = 1, 2, \ldots, n$$

This follows from the fact that all the zeros of $T_n(x)$ are solutions for $0 = \cos(n \arccos(x))$, and $\cos(y) = 0$ if and only if $y =$ odd integer $\cdot (\pi/2)$.

There will also always be $n + 1$ maxima and minima at

$$x_j = \cos(\frac{\pi j}{n}) \qquad j = 0, 1, \ldots, n$$

$T_n(x) = 1$ at every maxima , and -1 at the minima, which follows from how $T_n(x) = \cos(y)$ for some y.

These characteristics are visualized in Fig. 20.22 which plots $T_0(x) - T_5(x)$. The graph also shows that the roots are more numerous at the two ends of the $[-1, 1]$ range, which makes Chebyshev polynomials better suited for dealing with the oscillations that appear in some function approximations.

Fig. 20.22 was generated by chebPolys.py as a byproduct of it generating the equations for the first six Chebyshev polynomials:

```
T0(x) = 1
T1(x) = x
T2(x) = 2*x**2 - 1
T3(x) = 4*x**3 - 3*x
T4(x) = 8*x**4 - 8*x**2 + 1
T5(x) = 16*x**5 - 20*x**3 + 5*x
```

The code contains two ways to generate these polynomials, both of which use Sympy for algebraic manipulation. `chebyshevIters()` employs the Chebyshev recurrence relation:

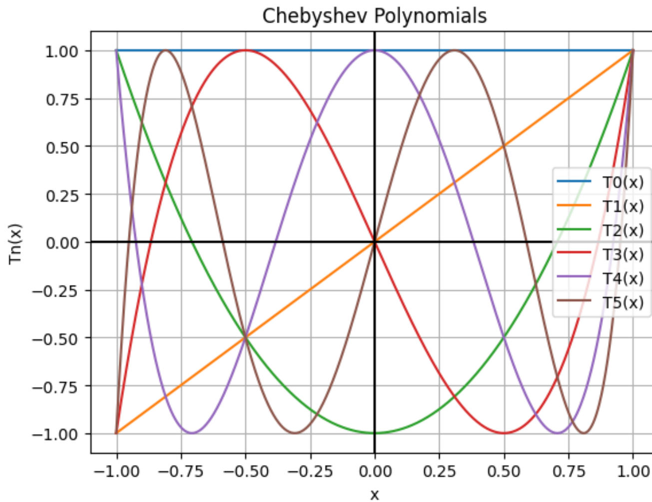


Figure 20.22. Plots of the Chebyshev Polynomials $T_0(x)$ – $T_5(x)$

```
def chebyshevIters(n):
  # the nth Chebyshev polynomial in x, T_n(x)
  if n == 0:
    return 1
  if n == 1:
    return x

  # implement T_n(x) = 2*x*T_{n-1}(x) - T_{n-2}(x)
  t0 = 1
  t1 = x
  for _ in range(2, n+1):
    t2 = sp.expand(2*x*t1 - t0)
      # expand the polynomial expression
    t0 = t1
    t1 = t2
  return t1
```

The second approach is to simply call Sympy's `chebyshevt()`:

```
for n in range(6):
  # the nth Chebyshev polynomial in x, T_n(x)
  # cPoly = chebyshevIters(n)
  cPoly = sp.chebyshevt(n, x)
  print(f"T{n}(x) = {cPoly}")
```

20.10.4 Using the Chebyshev Polynomials. Our aim is to approximate a function $f(x)$ in the interval $[-1, 1]$ in terms of a summation of n $T_j(x)$ Chebyshev polynomials ($j = 0, 1, ..., n-1$) multiplied by coefficients c_j that we have to determine. The resulting $(n-1)$-degree approximation for $f(x)$ has the form:

$$f(x) \approx \frac{1}{2}c_0 + \sum_{j=1}^{n-1} c_j T_j(x)$$

The coefficients c_j are defined by

$$\begin{aligned} c_j &= \tfrac{2}{n} \textstyle\sum_{k=1}^{n} f(x_k) T_j(x_k) \\ &= \tfrac{2}{n} \textstyle\sum_{k=1}^{n} f\Big[\cos\Big(\tfrac{\pi(k-1/2)}{n}\Big)\Big] \cos\Big(\tfrac{\pi j(k-1/2)}{n}\Big) \end{aligned}$$

Note that we're using the n zeros, x_k, of the polynomial, which means that we're only able to solve for n c_j coefficients.

The c_j equation is derived using $T_n(x) = \cos(n \arccos(x))$ and the n zeros located at

$$x_k = \cos\left(\frac{\pi\,(k - 1/2)}{n}\right), \qquad k = 1, 2, \ldots, n$$

Consider the kth zero in the $T_j(x)$ polynomial:

$$T_j(x_k) = \cos\left(j \arccos\left[\cos\left(\frac{\pi\,(k - 1/2)}{n}\right)\right]\right)$$

so that

$$T_j(x_k) = \cos\left(\frac{\pi j\,(k - 1/2)}{n}\right).$$

Let's assume that $f(x)$ can be approximated by some combination of Chebyshev polynomials such that:

$$f(x) \approx \sum_{j=0}^{n-1} a_j T_j(x).$$

Focus on the kth zero:

$$f(x_k) = \sum_{j=0}^{n-1} a_j T_j(x_k).$$

Multiply both sides by $T_i(x_k)$, where $i = 0, 1, ..., n-1$:

$$f(x_k) T_i(x_k) = \sum_{j=0}^{n-1} a_j T_j(x_k) T_i(x_k).$$

Now sum over the zero's range $k = 1, ..., n$:

$$\sum_{k=1}^{n} f(x_k) T_i(x_k) = \sum_{j=0}^{n-1} a_j \sum_{k=1}^{n} T_j(x_k) T_i(x_k).$$

The payoff of having the Chebyshev polynomials be orthogonal has arrived, since all the $T_j(x) \cdot T_i(x)$ multiples vanish except when $i = j$. There are two cases to consider:

Case 1: $i = j \neq 0$.

$$\sum_{k=1}^{n} f(x_k)T_j(x_k) = a_j \frac{n}{2}.$$

Hence

$$a_j = \frac{2}{n} \sum_{k=1}^{n} f(x_k)T_j(x_k) \qquad (j \neq 0).$$

Case 2: $i = j = 0$. Since $T_0(x) = 1$ we have

$$\sum_{k=1}^{n} f(x_k) = a_0 n.$$

Therefore

$$a_0 = \frac{1}{n} \sum_{k=1}^{n} f(x_k).$$

We use these a_j's to calculate the Chebyshev c_j coefficients. For $j = 0$,

$$a_0 = \frac{1}{n} \sum_{k=1}^{n} f(x_k) \qquad \text{and} \qquad c_0 = \frac{2}{n} \sum_{k=1}^{n} f(x_k).$$

Therefore, $a_0 = \frac{1}{2} c_0$, while for $j \geq 1$, $a_j = c_j$.

Starting from

$$f(x) \approx \sum_{j=0}^{n-1} a_j T_j(x),$$

replace a_0 by $\frac{1}{2} c_0$ and a_j by c_j for $j \geq 1$ to produce:

$$f(x) \approx \frac{1}{2} c_0 + \sum_{j=1}^{n-1} c_j T_j(x).$$

20.10.5 Reducing the Degree of the Approximation. Consider what happens if we truncate the approximation to the mth term:

$$f(x) \approx \frac{1}{2} c_0 + \sum_{j=1}^{m-1} c_j T_j(x).$$

Since the $T_j(x)$'s are bounded between ± 1, the loss can be no larger than the sum of the discarded c_j's. In fact, if the c_j's are rapidly decreasing (which is the typical case), then the error is dominated by $c_m T_m(x)$, an oscillatory function with $m+1$ extrema distributed smoothly over the interval $[-1, 1]$.

20.10.6 Implementing Chebyshev Fitting and Evaluation. We need code to obtain the c_j coefficients, and also to evaluate the approximation using those coefficients and $T_j(x)$.

For the first task, we've modified a function presented in Section 5.8 of Press et. al's *Numerical Recipes in C* [**PTVF92**]. `chebFit()` in faUtils.py allows the approximation range to be between arbitrary limits a and b, instead of just -1 to 1. This is effected by a change of variable

$$y = \frac{x - 1/2(b+a)}{1/2(b-a)}$$

The rest of the code implements the summation expressions for calculating c_j:

```
def chebFit(a, b, nTerms, f):
  coefs = [0]*nTerms
  scale = (b-a)/2
  offset = (a+b)/2

  # sample f() at Chebyshev zeros
  fzs = [ f(offset + math.cos(math.pi*(k+0.5)/nTerms) * scale)
                       for k in range(nTerms) ]
        # 0 to nTerms-1 == index 1 to n in the math
  fac = 2/nTerms

  # calculate nTerms Chebyshev coefs c_0 to c_n-1
  for j in range(nTerms):
    tot = 0
    for k in range(nTerms):
      # 0 to nTerms-1 == index 1 to n in the math
      tot += fzs[k] * math.cos(math.pi*j*(k+0.5)/nTerms)
    coefs[j] = fac * tot
  coefs[0] *= 0.5
  return coefs
```

`chebEval()` in faUtils.py calculates the $f(x)$ approximation. It generates each Chebyshev polynomial using the recurrence relation $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$ inside a loop:

```
def chebEval(a, b, coefs, nTerms, x):
  if (x-a)*(x-b) > 0:
    print("x not in range in chebEval()")
    sys.exit(1)
  xv = (2*x - (b+a))/(b-a)     # map x onto [-1,1]

  t0 = 1  # T_0(x) = 1
  approx = coefs[0] * t0
  if nTerms == 0:
    return approx
```

```
  t1 = xv   # T_1(x) = x
  approx += coefs[1] * t1
  for k in range(2, nTerms):
    # T_n(x) = 2*x*T_{n-1}(x) - T_{n-2}(x)
    t2 = 2*xv*t1 - t0
    approx += coefs[k] * t2
    t0 = t1      # shift vars
    t1 = t2
  return approx
```

This is not quite the same approach as used by Press *et. al* [**PTVF92**] who employ Clenshaw's recurrence.

The two functions are tested in chebTest.py by generating an 8-term approximation for $f(x) = x(x-2)\sin(x)$ in the range $[-\pi/2, \pi/2]$:

```
nTerms = 8   # try 6, 8, 10
a = -math.pi/2; b = math.pi/2
print(f"Evaluating f(x) using {nTerms} terms in [{a},{b}]:")
coefs = faUtils.chebFit(a, b, nTerms, f)
faUtils.printCoefs(coefs)

print(f"    x           actual       chebyshev     abs error")
for i in range(-4, 5):
  x = i * math.pi/2 / 10
  chebx = faUtils.chebEval(a, b, coefs, nTerms, x)
  print(f"{x:12.6f} {f(x):12.6f} {chebx:12.6f}  {abs(f(x) - chebx):12.6f}")
```

The approximation's results are correct to two or more decimal places, and improve with a larger `nTerms` setting.

chebRunge.py generates three Chebyshev polynomials that fits $f(x) = 1/(1+25x^2)$ in the range $[-1, 1]$ using 6, 11, and 16 terms, as seen in Fig. 20.23. Recall that a variant of this function ($1/(1+x^2)$) posed problems for the Taylor series approximation back in section 20.7.4, and this version is a well-known difficult case for approximations such as Lagrange that have a tendency to oscillate.

Oscillation also occur with the Chebyshev polynomial, but the fact that the approximation's zeros are concentrated at the two ends of the interval helps to reduce the size of the problem.

20.11 Padé Approximation

Polynomial approximations can model most functions, are easily evaluated, and give good results. However, a polynomial can't easily represent poles or steep local behavior, and also has a tendency to oscillate.

Rational approximations, such as the Padé, perform well near singularities, can often approximate functions accurately with fewer terms than polynomial

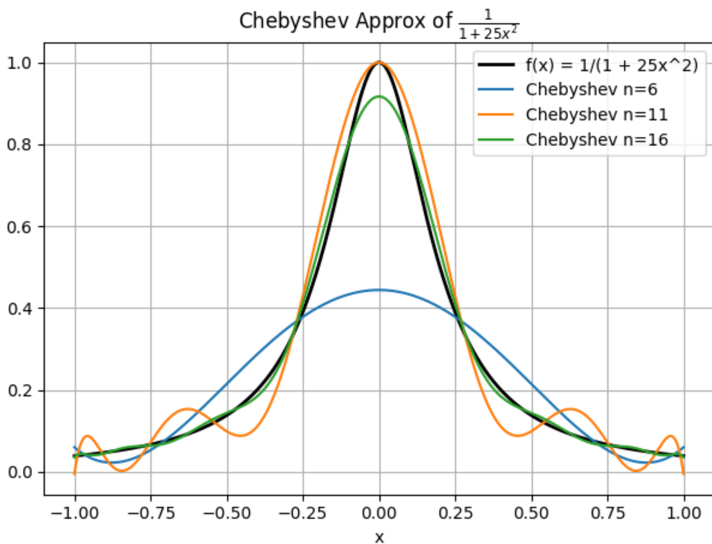


Figure 20.23. Three Approximations to $1/(1 + 25x^2)$

versions, and avoid oscillations by spreading errors more evenly over the expansion interval. On the downside, their generation requires the solving of a linear system, and they may be less stable outside the interval.

Suppose we are given a Taylor power series $\sum_{k=0}^{\infty} a_k x^k$ representing a function $f(x)$. A Padé approximation is a rational

$$r(x) = \frac{p(x)}{q(x)} = \frac{p_0 + p_1 x + \cdots + p_n x^n}{q_0 + q_1 x + \cdots + q_m x^m}$$

which has a expansion that agrees with the Taylor series based on the chosen (n, m) order. Although there are $n + 1$ numerator and $m + 1$ denominator coefficients in the rational, we can divide through by q_0, leaving just $n + m + 1$ unknowns. This suggests that the rational ought to fit the power series through the orders $1, x, x^2, \ldots, x^{n+m}$, or in other words:

$$\sum_{k=0}^{\infty} a_k x^k = \frac{p_0 + p_1 x + \cdots + p_n x^n}{q_0 + q_1 x + \cdots + q_m x^m} + O(x^{n+m+1})$$

or

$$(q_0 + q_1 x + \cdots + q_m x^m)(a_0 + a_1 x + \ldots) = p_0 + p_1 x + \cdots + p_n x^n + O(x^{n+m+1}). \quad (1)$$

We'll assume that the term coefficients for x^{n+1} and higher on the right-hand side of equ. (1) equal 0. This lets us write the expressions on the left-hand side which

utilize the same powers of x as:

$$\begin{array}{rcl} q_m a_{n-m+1} + q_{m-1} a_{n-m+2} + \cdots + q_0 a_{n+1} & = & 0 \\ q_m a_{n-m+2} + q_{m-1} a_{n-m+3} + \cdots + q_0 a_{n+2} & = & 0 \\ & \vdots & \\ q_m a_n + q_{m-1} a_{n+1} + \cdots + q_0 a_{n+m} & = & 0 \end{array}$$

The first equation is for terms involving x^{n+1}, the second for x^{n+2}, and so on up to x^{n+m}. Since $q_0 = 1$, the last term of these equations simplifies to an a coefficient, leaving m linear equations involving m q's. These can be expressed in matrix form as:

$$\begin{bmatrix} a_{n-m+1} & a_{n-m+2} & \cdots & a_n \\ a_{n-m+2} & a_{n-m+3} & \cdots & a_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ a_n & a_{n+1} & \cdots & a_{n+m-1} \end{bmatrix} \begin{bmatrix} q_m \\ q_{m-1} \\ \vdots \\ q_1 \end{bmatrix} = - \begin{bmatrix} a_{n+1} \\ a_{n+2} \\ \vdots \\ a_{n+m} \end{bmatrix}$$

This linear system can be solved to find the q coefficients.

The rational's numerator values, $p_0, p_1, \ldots, p_n$ are obtained by equating the coefficients of $1, x, x^2, \ldots, x^n$ on the left- and right-sides of equ. (1):

$$\begin{array}{rcl} a_0 & = & p_0 \\ a_1 + q_1 a_0 & = & p_1 \\ a_2 + q_1 a_1 + q_2 a_0 & = & p_2 \\ & \vdots & \\ a_n + \sum_{k=1}^{\min(n,m)} q_k a_{n-k} & = & p_n \end{array}$$

This is a linear system of $n+1$ unknowns in $n+1$ equations, so is solvable.

Often these two linear systems are combined into a single matrix:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & -a_0 & 0 & \cdots & 0 \\ & & \vdots & & & & \vdots & \\ 0 & 0 & \cdots & 1 & -a_n & -a_{n-1} & \cdots & -a_{n-m+1} \\ 0 & 0 & \cdots & 0 & -a_{n+1} & -a_n & \cdots & -a_{n-m+2} \\ 0 & 0 & \cdots & 0 & -a_{n+2} & -a_{n+1} & \cdots & -a_{n-m+3} \\ & & \vdots & & & & \vdots & \\ 0 & 0 & \cdots & 0 & -a_{n+m-1} & -a_{n+m-2} & \cdots & -a_n \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \\ q_1 \\ q_2 \\ \vdots \\ q_m \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \\ a_{n+1} \\ a_{n+2} \\ \vdots \\ a_{n+m} \end{bmatrix}$$

20.11.1 A Padé Approximation for e^{-x}. Consider the Maclaurin series:

$$e^{-x} = \sum_{i=0}^{\infty} \frac{(-1)^i}{i!} x^i.$$

We'll use this series to formulate a Padé approximation for e^{-x} of degree 5 with $n = 3$ and $m = 2$. We need values for p_0, p_1, p_2, p_3, q_1, and q_2 so that the coefficients of x^k for $k = 0, 1, \ldots, 5$ match in the expression:

$$\left(1 - x + \frac{x^2}{2} - \frac{x^3}{6} + \cdots\right)(1 + q_1 x + q_2 x^2) = (p_0 + p_1 x + p_2 x^2 + p_3 x^3).$$

Expanding and collecting terms:

$$\begin{array}{llll}
x^5: & -\frac{1}{120} + \frac{1}{24}q_1 - \frac{1}{6}q_2 = 0 & x^2: & \frac{1}{2} - q_1 + q_2 = p_2 \\
x^4: & \frac{1}{24} - \frac{1}{6}q_1 + \frac{1}{2}q_2 = 0 & x^1: & -1 + q_1 \qquad = p_1 \\
x^3: & -\frac{1}{6} + \frac{1}{2}q_1 - q_2 = p_3 & x^0: & 1 \qquad\quad = p_0.
\end{array}$$

$p_0 = 1$ and there are five more variables to solve using the equations for x^1 to x^5. The solution is:

$$\left\{p_1 = -\frac{3}{5},\ p_2 = \frac{3}{20},\ p_3 = -\frac{1}{60},\ q_1 = \frac{2}{5},\ q_2 = \frac{1}{20}\right\}$$

so the approximation is

$$r(x) = \frac{1 - \frac{3}{5}x + \frac{3}{20}x^2 - \frac{1}{60}x^3}{1 + \frac{2}{5}x + \frac{1}{20}x^2}.$$

20.11.2 Implementing Padé. faUtils.py contains two Padé support functions – `padeApprox()` and `padeSolve()`. `padeApprox()` generates a power series for the given function, and passes it to `padeSolve()` to create the approximation:

```
def padeApprox(synfn, n, m, x0=0):
  print(f"\nUsing the series:")
  coefs = seriesCoefs(synfn, x0=x0, nTerms=n+m+1, verbose=True)
  p, q = padeSolve(n, m, coefs)
  # print the coefficients in p and q;
  # generate a Latex expression for the p/q rational
  #   :
  return p, q, latexFnm
```

The power series is obtained using `seriesCoefs()` which was described back in section 20.7.

`padeSolve()` is an implementation of the two-stage linear solving outlined above. Gaussian elimination (see section 6.8.4) solves the linear equations for the q coefficients, and the p coefficients are obtained by simple substitutions:

```
def padeSolve(n, m, coefs):
  # Build linear system for q1..qm (m equations)
  # since we know q0 == 1
  # Sum(q[j] * coefs[n+k-j]) = -coefs[n+k] for k = 1..m
  arr = []
  for k in range(1, m+1):
    # k == 1 corresponds to x^{n+1}
    row = []
    for j in range(1, m+1):
      row.append(coefs[n+k-j])
    row.append(-coefs[n+k])
    arr.append(row)

  xs = gaussian(arr)  # Solve for q1..qm
  # extract q coefficients
  q = [0] * (m+1)
  q[0] = 1
  for j in range(1, m+1):
    q[j] = xs[j-1]

  # Compute p coefficients (p0 to pn) using the q's
  # p[k] = Sum(q[j] * coefs[k-j])
  p = [0] * (n+1)
  for k in range(n+1):
    tot = 0
    for j in range(min(k, m) + 1):
      tot += q[j] * coefs[k-j]
    p[k] = tot
  return p, q
```

20.11.3 Using Padé. padeTests.py presents a list of functions to the user, and the one selected is given a Padé approximation based on a predefined (n, m) order. The approximation is evaluated against multiple values, and its results are compared with those produced by the actual function. The approximation is also plotted against the function.

Let's compare the `padeApprox()` e^{-x} with the version calculated by hand earlier. The call to `padeTests.py` is:

```
> python padeTests.py
Functions: ['sin(x)', 'exp(x)', 'exp(-x)',
            '1-0.5*ln(1+x^2)', 'tan(x)']
Enter index of function name: 2
Approximating exp(-x) for order (3, 2)
```

Fig. 20.24 shows the plot of the approximation versus the actual function.

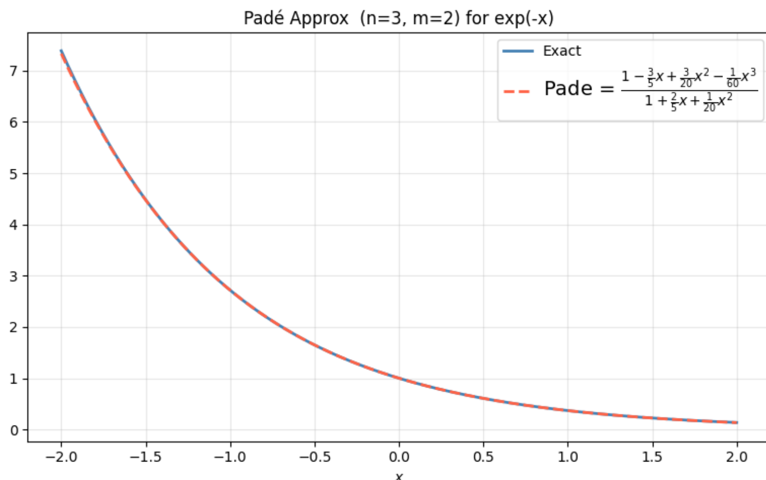


Figure 20.24. A Padé Approximation for e^{-x}

Note that the equation for the Padé approximation printed in the plot's legend matches the one we calculated above. The program's table of comparisons:

```
exp(-x) Comparisons:
      x         Actual          Pade          Abs Err        Rel Err
----------------------------------------------------------------------
  -2.000     7.38905610     7.33333333     0.05572277     0.00754126
  -1.000     2.71828183     2.71794872     0.00033311     0.00012254
  -0.500     1.64872127     1.64871795     0.00000332     0.00000201
   0.000     1.00000000     1.00000000     0.00000000     0.00000000
   0.500     0.60653066     0.60652921     0.00000145     0.00000239
   1.000     0.36787944     0.36781609     0.00006335     0.00017220
   2.000     0.13533528     0.13333333     0.00200195     0.01479252
Average Relative Error = 3.23327537e-03
```

The approximation is excellent around the Taylor series' expansion point ($x_0 = 0$), but becomes less accurate further away. One way to improve matters is to increase the number of n and m coefficients, but the better approach, as typified by Allison's code at the start of this chapter, is to scale and translate the x input down to a narrower range.

One use of a Padé approximation is to determine the location of function singularities when only the first few terms in its expansion are known. Let's demonstrate this in a simple way by looking at $\tan(x)$ which is known to have singularities at $x = \pm\pi/2, \pm 3\pi/2$, etc.

A Padé approximation for tan(x) generated by `padeTests.py`:

$$r(x) = \frac{x - \frac{2}{21}x^3}{1 - \frac{3}{7}x^2 + \frac{1}{105}x^4}$$

and its plot between -2 and 2 is shown in Fig. 20.25.

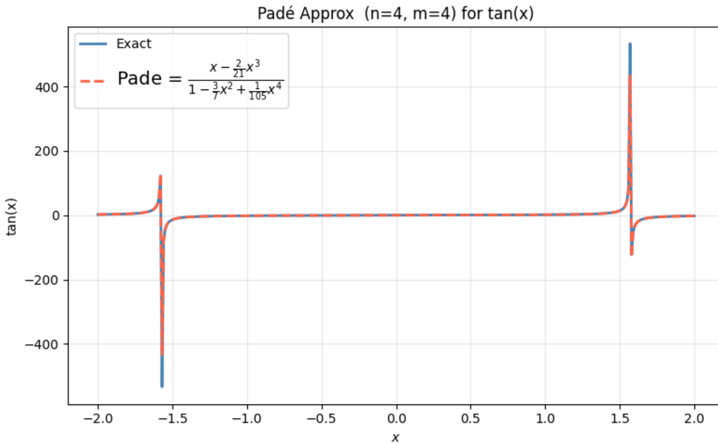


Figure 20.25. A Padé Approximation for tan(x)

It's remarkable that the curve is able to deal fairly well with the singularities at $\pm\pi/2$.

Aside from an examination of the plot, the curve's singularities can also be identified by setting the rational's denominator to 0 and solving it:

$$1 - \frac{3}{7}x^2 + \frac{1}{105}x^4 = 0$$

Let $y = x^2$ so that $y^2 - 45y + 105 = 0$, and then use the quadratic formula:

$$y = \frac{45 \pm \sqrt{45^2 - 4(105)}}{2}$$

to get

$$y_1 \approx 42.531 \quad \text{and} \quad y_2 \approx 2.469.$$

Taking square roots:

$$x \approx \pm 1.571 \quad \text{and} \quad x \approx \pm 6.522$$

and note that $\frac{\pi}{2} \approx 1.570796$.

The $r(x)$ rational for $\tan(x)$ appears to be very different from its standard Maclaurin series expansion:

$$\tan(x) = x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \frac{62x^9}{2835} + \frac{1382x^{11}}{155925} + \cdots$$

This may seem strange since the Padé approximation is derived from the Maclaurin. But if the $r(x)$ numerator is algebraically divided by its denominator, the Maclaurin can be recovered (subject to truncation errors caused by the choice of (n, m)). This can be confirmed by using Sympy:

```
>>> import sympy as sp
>>> x = sp.Symbol('x')
>>> numer = x - (2/21)*x**3
>>> denom = 1 - (3/7)*x**2  + (1/105)*x**4
>>> expr = numer/denom
>>> series = expr.series(x, 0, 8)
>>> seriesFrac = sp.nsimplify(series)
>>> seriesFrac
x + x**3/3 + 2*x**5/15 + 17*x**7/315 + O(x**8)
```

20.11.4 A Return to 'Quick and Dirty'.

Most of Allison's functions from section 20.1 use rational approximations obtained from Hart's *Computer Approximations* [**Har68**]. Hart utilized Remez rationals, but it's interesting to see how well Padé can perform under similar constraints.

dpade.py is similar to `padeTests.py`, but utilizes Taylor expansions around points other than $x_0 = 0$. It presents a list of functions to the user, and the selected one is given a Padé approximation based on an (n, m) order. The rational is evaluated against a series of values, and the results are compared to those produced by the actual function.

For example, $\arctan(x)$ uses a Taylor series expanded around $x_0 = 0.5$, and an order $(2, 3)$ which is the same as `atanD(x)` in `dirty.py` (see section 20.4).

`dpade.py`'s Padé approximation is:

$$\frac{\frac{4126}{8899} + \frac{8957}{8844}x + \frac{275}{1591}x^2}{1 + \frac{2413}{5258}x + \frac{635}{2342}x^2 - \frac{141}{2387}x^3}$$

or

$$\begin{aligned} p &= [0.463648, 1.012777, 0.172847] \\ q &= [1, 0.458920, 0.271136, -0.059070] \end{aligned}$$

while the coefficients employed in `dirty.py` are:

$$\begin{aligned} p &= [80.78998, 72.58144, 11.11774] \\ q &= [80.78999, 99.51128, 28.13286, 1]. \end{aligned}$$

Even after scaling, the two sets are disappointingly quite different. In addition, while `atanD(x)` boasts 6 or 7 decimal places of accuracy, the `dpade.py` approximation only manages 3-4:

```
atan(x) Comparisons:
     x          Actual           Pade           Abs Err        Rel Err
-----------------------------------------------------------------------
   0.100     0.09966865     0.09980653     0.00013788     0.00138338
   0.500     0.46364761     0.46364761     0.00000000     0.00000000
   0.900     0.73281510     0.73286184     0.00004673     0.00006377
```

20.11.5 Chebyshev Rational Function Approximation. More accurate rational approximations are possible if we use a Chebyshev polynomial as the starting point rather than a Taylor series. The rest of the procedure is pretty much unchanged except that each x^k term in the Padé approximation is replaced by a kth-degree Chebyshev polynomial, $T_k(x)$. $r(x)$ has the form

$$r(x) = \frac{\sum_{k=0}^{n} p_k T_k(x)}{\sum_{k=0}^{m} q_k T_k(x)},$$

and the original $f(x)$ is expressed as a sum of Chebyshev polynomials

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x).$$

These are made equivalent as before

$$\sum_{k=0}^{\infty} a_k T_k(x) = \frac{\sum_{k=0}^{n} p_k T_k(x)}{\sum_{k=0}^{m} q_k T_k(x)}$$

or

$$(T_0(x) + q_1 T_1(x) + \cdots + q_m T_m(x))(a_0 T_0(x) + a_1 T_1(x) + \dots) = (p_0 T_0(x) + p_1 T_1(x) + \cdots + p_n T_n(x)).$$

Two problems make this Chebyshev approach more difficult to implement. One is that the multiplication of the polynomial $q(x)$ and the series for $f(x)$ involves products of Chebyshev polynomials. This is addressed by making use of the relationship

$$T_i(x)T_j(x) = \frac{1}{2}\left[T_{i+j}(x) + T_{|i-j|}(x)\right].$$

The other issue involves the computation of the Chebyshev series for $f(x)$; usually numerical integration is required.

Exercises

(1) dirty.py contains Python versions of Dennis Allison's five short but efficient subroutines for $\sqrt{x}$, $\log_{10} x$, 10^x, $\arctan x$, and $\cos x$ [**All78**]. The functions utilize various range reducing formulae, and most employ polynomial or rational function approximations borrowed from Hart's *Computer Approximations* [**Har68**].

Fig. 20.2 shows that Allison approximates $\log_{10}(x)$ using a rational with coefficients that offer a precision of 8.66 (8 or 9 correct decimal digits). He approximates $\arctan(x)$ with a similar level of precision, as shown in Fig. 20.7. However, it would be more useful nowadays to offer 16 correct decimal digits when working with Python floats. This level of precision for $\log_{10}(x)$ and $\arctan(x)$ leads us to Hart's 2328 and 5100 tables, shown in Fig. 20.26.

LOG10 2328

Coeff.	Exp.	Value
P00	(+ 3)	−.42616 43208 95968 90386 37477
P01	(+ 3)	+.91144 54651 77967 80082 9589
P02	(+ 3)	−.64969 14305 03776 14358 6509
P03	(+ 3)	+.17197 78945 75261 36426 622
P04	(+ 2)	−.12409 45842 40688 46300 36
Q00	(+ 3)	−.49063 98062 30494 57788 77669
Q01	(+ 4)	+.12128 86972 67476 95006 65389
Q02	(+ 4)	−.10541 52597 77636 39322 90112
Q03	(+ 3)	+.37689 50738 71445 48066 757
Q04	(+ 2)	−.47842 12345 10828 24153 49
Q05	(+ 1)	+.1

ARCTN 5100

Coeff.	Exp.	Value
P00	(+ 4)	+.12097 47001 75809 07217 24071 5
P01	(+ 4)	+.30310 74595 61150 83044 21280 7
P02	(+ 4)	+.27617 19824 61388 34959 05378 4
P03	(+ 4)	+.11141 29072 84551 83546 17294 2
P04	(+ 3)	+.19257 92014 48155 96134 74286
P05	(+ 2)	+.11322 15941 16764 65523 6245
P06	(− 1)	+.97627 21591 71763 30369 83
Q00	(+ 4)	+.12097 47001 75809 07287 51419 7
Q01	(+ 4)	+.34343 23596 19753 51716 54706 9
Q02	(+ 4)	+.36645 44956 32837 49893 50479 6
Q03	(+ 4)	+.18216 00339 29184 64941 50922 5
Q04	(+ 3)	+.42307 16464 80904 78045 24206
Q05	(+ 2)	+.39917 88424 86537 98150 1999
Q06	(+ 1)	+.1

Figure 20.26. Hart's Tables for $\log_{10}(x)$ and $\arctan(x)$ with 16 d.p. of Accuracy

Modify `log10D(x)` and `atanD(x)` in dirty.py to use the coefficients in Fig. 20.26. Incidentally, there's no need to for you to type in these lengthy values, as they're available in HartTables.txt.

Compare the quality of the results with the old versions of `log10D(x)` and `atanD(x)`.

(2) Investigate the power series methods for computing e^{-x} when x is close to -3. The Maclaurin series is:

$$e^{-x} = \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n!}$$

and the Taylor series centered at $x = -3$ is

$$e^{-x} = e^3 \sum_{n=0}^{\infty} \frac{(-(x+3))^n}{n!}$$

Note that the Taylor series is equivalent to the Maclaurin with variable renaming. Let $\delta = x + 3$, and then

$$e^{-x} = e^3 e^{-\delta} = e^3 \sum_{n=0}^{\infty} \frac{(-\delta)^n}{n!}.$$

Explain why the second series might be expected to converge more rapidly near $x = -3$, and why one form is more susceptible to round-off errors.

See how the errors vary with polynomial degree and across a range of x values. You can measure the absolute error using $|f(x) - P_N(x)|$, and the relative error with $\frac{|f(x)-P_N(x)|}{|f(x)|}$, where $P_N(x)$ is the power series expanded to N terms.

Several techniques for rewriting a Maclaurin series are listed below. Investigate how they affect the quality of the e^{-x} series.

- **Reciprocals**. Since $e^{-x} = 1/e^x$, you can compute $e^x = e^{-3}e^{\delta}$ and take the reciprocal of the result.
- **Repeated Halving**. Use

  $$e^{-x} = \left(e^{-x/2^m}\right)^{2m}$$

  where m is chosen so that $|x|/2^m$ is small. This approach (and the next two) relies on the fact that multiplication by powers of two is both fast and accurate in floating-point arithmetic, and often implemented by bit-shifting.
- **Centering**. Since $e^{-\ln 2} = 1/2$, you can write $x = k \ln 2 + r$, giving $e^{-x} = 2^{-k}e^{-r}$.
- **Rational Approximation**. Define

  $$e^{-\delta} = \frac{e^{-\delta/2}}{e^{\delta/2}},$$

  and approximate its numerator and denominator separately.
- **Hyperbolics**. Utilize the identity $e^{-x} = \cosh(x) - \sinh(x)$, along with the Maclaurin series for those functions.

Another type of series optimization involves the way that calculations are performed. For example:

- **Horner's Method**. Rather than evaluating

$$1-\delta+\frac{\delta^2}{2!}-\frac{\delta^3}{3!}+\cdots,$$

rewrite the expression as

$$1+\delta\left(-1+\delta\left(\frac{1}{2}+\delta\left(-\frac{1}{6}+\cdots\right)\right)\right).$$

- **Summing Terms Recursively**. Instead of computing $\frac{\delta^n}{n!}$ from scratch each time (e.g. by calling `math.factorial()`), use

$$t_0=1, \qquad t_{n+1}=t_n\frac{-\delta}{n+1},$$

- **Sum Small Terms First**. Sum the series terms in reverse order to reduce the chance of rounding errors.

3. Table 20.3 lists the population of the United States from 1950 to 2020 along with estimations for 2021-2024. Use Lagrange interpolation to approximate the population in the years 1940, 2000, 2024, and 2050. The population in 1940 was approximately 132,165,000. How accurate do you think your 2050 figure is?

Year	Population
1950	151,325,798
1960	179,323,175
1970	203,211,926
1980	226,545,805
1990	248,709,873
2000	281,421,906
2010	308,745,538
2020	331,449,281
2021	332,031,554 (est)
2022	333,287,557 (est)
2023	336,806,231 (est)
2024	340,110,988 (est)

Table 20.3. Population of the United States, 1950–2024.

There's no need to type in this population data, as it's available in `usPop.txt`.

4. Using a Maclaurin polynomial for xe^x, obtain a small degree Chebyshev approximation while keeping the error less than 0.01 across the $[-1,1]$ interval.

5. Using a Maclaurin polynomial for $\sin(x)$, obtain a small degree Chebyshev approximation while keeping the error less than 0.01 across the $[-1, 1]$ interval.

6. Show that for any positive integers i and j with $i > j$, we have

$$T_i(x)T_j(x) = \frac{1}{2}[T_{i+j}(x) + T_{i-j}(x)].$$

7. Show that the Chebyshev polynomial $T_n(x)$ has n distinct zeros in $[-1, 1]$.

8. The table below lists results for the Padé approximation of $f(x) = e^{-x}$ using degree 5 with $n = 3$ and $m = 2$, the sixth Maclaurin polynomial $(n + m + 1)$, and function values when $x = 0.2, 0.4, 0.6, 0.8$, and 1. The Padé approximation is clearly better than the Maclaurin.

```
    x       Actual      Pade        Mac          Pade Err     Mac Err
------------------------------------------------------------------
  0.200    0.818731    0.818731    0.818733     0.000000     0.000003
  0.400    0.670320    0.670320    0.670400     0.000000     0.000080
  0.600    0.548812    0.548808    0.549400     0.000004     0.000588
  0.800    0.449329    0.449310    0.451733     0.000019     0.002404
  1.000    0.367879    0.367816    0.375000     0.000063     0.007121
```

But does the Padé approximation still win when it utilizes different orders of degree 5:

a. $n = 0, m = 5$

b. $n = 1, m = 4$

c. $n = 2, m = 3$

d. $n = 4, m = 1$

e. $n = 5, m = 0$

Answers

3. See usPop.py.

4. See chebExp.py.

6. If $i > j$, then

$$\begin{aligned} \frac{1}{2}(T_{i+j}(x) + T_{i-j}(x)) &= \frac{1}{2}(\cos(i+j)\theta + \cos(i-j)\theta) \\ &= \cos(i\theta)\cos(j\theta) \\ &= T_i(x)\,T_j(x). \end{aligned}$$

(n,m)	Error $\times 10^{-6}$
(0,5)	133
(1,4)	80
(2,3)	13
(3,2)	17
(4,1)	47
(5,0)	321

Table 20.4. Padé Errors for Different (n, m).

7. The zeros of $T_n(x)$ are $x_k = \cos\left(\frac{2k-1}{2n}\pi\right)$ for $k = 1, 2, \ldots, n$.

The cosine function is strictly decreasing from 0 to π, with values decreasing from $\cos(0) = 1$ to $\cos(\pi) = -1$. So, in reverse order:

$$\begin{aligned}(-1 = \cos(\pi)) &< (\cos\left(\tfrac{2n-1}{2n}\pi\right) = x_n) \\ &< x_{n-1} < \cdots < (x_1 = \cos\left(\tfrac{\pi}{2n}\right)) \\ &< (\cos(0) = 1)\end{aligned}$$

Hence the zeros are distinct and lie in $[-1, 1]$.

8. See `padeExp.py`.

The average error for different versions of (n, m) are given in Table 20.4