# The `polyfit` Package for Quad-precision Orthogonal Polynomial Least Squares

Mark Hays `minmus-9`

September 20, 2021

**Abstract**

In this note I present the Python2 and Python3 `polyfit` package that implements quad-precision [2, 5] least-squares polynomial fitting using orthogonal polynomials [4, 6, 7, 8]. The code is written in pure Python and is slow but numerically stable [1, 3, 9] and more accurate than the traditional approach. A copy of the source code for the 250 SLOC Python reference implementation is included in appendix C. A much faster `C` version also ships with this package; there is a Python ctypes-based interface called `cpolyfit` for that version.

## 1 TL;DR Quick Start

This package consists of 3 modules:

- The 250 SLOC Python reference implementation in `polyfit.py`
- The `C` implementation for `libpolyfit.so` contained in `polyfit.c` and `polyfit.h`
- The `cpolyfit.py` Python `ctypes` interface to `libpolyfit.so`. This interface is identical to the one provided by the reference implementation

Unlike `polyfit.py`, the `C` and `ctypes` APIs are not commented. The `C` version exactly follows the reference implementation; the comments in the reference implementation also apply to the `C` version. The `ctypes` version most consists of glue code to `libpolyfit.so` and isn't interesting from an algorithmic viewpoint.

The following `demo()` function is a copy of `examples/ex1.py` and exercises the full `polyfit` API.

```
#!/usr/bin/env python3

import math

from polyfit import Polyfit

def demo():
    "demo of the api"
    ## pylint: disable=unnecessary-comprehension

    ## poly coefficients to fit
    cv = [2, math.sqrt(2), -1, math.pi]

    ## evaluate using horner's method
    def pv(x):
        "evaluate using cv"
        r = 0.
        for c in cv:
            r *= x
            r += c
        return r

    ## define the x and y values for the fit
    xv = [x for x in range(100000)]
    yv = [pv(x) for x in xv]

    ## weights:
    ##     uniform, minimize max residual
    #wv = [1. for _ in xv]

    ##     relative, minimize relative residual
    ##     note that y is nonzero for this example
    wv = [y ** -2. for y in yv]
```

```

    ## perform the fit
    fit = Polyfit(len(cv) - 1, xv, yv, wv)

    ## dump fit stats
    deg = fit.maxdeg()
    print("maxdeg", deg)
    print("points", fit.npoints())
    print("time  ", fit.runtime())

    ## per-degree rms errors; relative resid error across all x
    print("erms  ", [fit.rms_err(d) for d in range(deg + 1)])
    print("relerr", fit.rel_err())

    ## print some values
    for i in range(5):
        print("value  %.1f %s" % (xv[i], fit(i, nderiv=-1)))

    ## coefs about x0=0.; value and all derivs at 0.
    print("coefs0", fit.coefs(deg, x0=0.))
    print("value0", fit(0., deg, deg))

if __name__ == "__main__":
    demo()

## EOF
```

Following is the `pydoc` output for the package's `polyfit` module.

```
Help on module polyfit:

NAME
    polyfit - quad precision orthogonal polynomial least squares fit

DESCRIPTION
    see polyfit.pdf and the code in examples/

CLASSES
```

```
__builtin__.object
    Polyfit

class Polyfit(__builtin__.object)
 |  polynomial fitting class
 |
 |  Methods defined here:
 |
 |  __call__(self, x, degree=None, nderiv=0)
 |      evaluate poly and (optionally) some of its derivatives.
 |      if degree is None, return the values for the maximum
 |      fit degree
 |
 |      if nderiv is negative, return the polynomial value and all
 |      derivatives as a list. this is the default
 |
 |      if nderiv is 0, return the scalar polynomial value.
 |      this is the default
 |
 |  __init__(self, maxdeg, xv, yv, wv)
 |      given x- and y-values in xv[] and yv[],
 |      along with positive fit weights in wv[],
 |      compute all least-squares fits up to
 |      degree maxdeg
 |
 |  close(self)
 |      finalize self (no-op for this version)
 |
 |  coefs(self, degree=None, x0=0)
 |      return the coefficients for fit degree degree about
 |      (x - x0). if degree is None, use the maximum fit
 |      degree
 |
 |  maxdeg(self)
 |      return max fit degree
 |
 |  npoints(self)
 |      return number of data points used to perform the fit
```

```
 |
 |  rel_err(self, degree=None)
 |      return the max relative fit error across all x values
 |
 |  rms_err(self, degree=None)
 |      return the residual rms fit residual for degree degree.
 |      if degree is None, use the maximum fit degree
 |
 |  runtime(self)
 |      return the time it took to perform the fit

DATA
    __all__ = ['Polyfit']
```

The full source code for the reference implementation, `polyfit.py`, accompanies this file and is also included in appendix C.

# 2 The Theory

Following the development in [4], suppose we have $N$ ordered pairs of data

$$\{(x_i, y_i)\}_{i=1}^{N} \tag{1}$$

with $x_i$ distinct. We'd like to find the linear least-squares fit to a set of $D+1$ linearly independent functions $\phi_j$, $0 \le j \le D$

$$\hat{f}(x) = \sum_{k=0}^{D} a_k \, \phi_k(x) \tag{2}$$

$$y_i \approx \hat{f}(x_i), \quad 1 \le i \le N \tag{3}$$

with $D < N - 1$. The functions $\phi_k$ do not need to be linear; it is the dependence on the coefficients $a_k$ that makes the problem linear. In the common polynomial case, you'd likely choose

$$\phi_k(x) = x^k. \tag{4}$$

To perform a least squares fit, we'll minimize the error $E$:

$$E = \sum_{i=1}^{N} w_i \left( y_i - \sum_{k=0}^{D} a_k \phi_k(x_i) \right)^2 \tag{5}$$

for some given positive weights $w_i$, $1 \le i \le N$, and for some unknown coefficients $a_k$, $0 \le k \le D$. The quantity $E$ is clearly a positive-definite quadratic form and so a minimum can be found by setting the gradient of $E$ with respect to the $a_j$ to zero:

$$\frac{\partial E}{\partial a_k} = 0, \quad k \le 0 < M. \tag{6}$$

Computing the partials from (5) and setting them to zero, we get

$$\begin{aligned} \frac{\partial E}{\partial a_k} &= -2 \sum_{i=1}^{N} w_i \left( y_i - \sum_{k=0}^{D} a_k \phi_k(x_i) \right) \phi_k(x_i) \\ &= 0. \end{aligned}$$

After a little rearrangement, this becomes

$$\sum_{i=1}^{N} w_i y_i \phi_k(x_i) = \sum_{k=0}^{D} a_k \sum_{i=1}^{N} w_i \phi_k(x_i) \phi_k(x_i). \tag{7}$$

The functional $(\cdot, \cdot)$ defined by

$$(f, g) = \sum_{i=1}^{N} w_i f(x_i) g(x_i) \tag{8}$$

for arbitrary functions $f$ and $g$ defines an *inner product* on the *vector space* of functions defined on $\{x_i\}$ and spanned by $\{\phi_k\}$ because it is linear, symmetric, and positive definite (since the $w_i$ are positive). In addition, this inner product is associative:

$$(f, g\, h) = (f\, g, h). \tag{9}$$

With this definition, we can rewrite (7) as

$$(y, \phi_k) = \sum_{j=0}^{D} a_j \, (\phi_j, \phi_k) \tag{10}$$

For the common polynomial case in (4), this reads

$$\sum_i w_i y_i x_i^k = \sum_j a_j \sum_i w_i x_i^{j+k}. \tag{11}$$

These two are called the *normal equations* and are the solution to the $M \times M$ linear system

$$\begin{aligned} Aa &= B \qquad (12) \\ A_{j,k} &= (\phi_j, \phi_k) \\ B_k &= (y, \phi_k) \end{aligned}$$

for the coefficient vector $a$. In the common case with $w_i = 1$, the matrix $M$ is the Hilbert matrix, the poster-child for badly behaved linear systems, and the condition number of this matrix is exponential in $D$. You get roundoff error not only in computing the matrix elements, but also during the solution of the linear system. The number of points $N$ and the fit degree $D$ must be small in order to prevent catastrophic roundoff error.

In what follows, we wil make a different choice for the $\phi_k$ that will minimize roundoff errors. Suppose that the functions $\phi_k$ are *orthogonal* with respect to the inner product so that

$$(\phi_j, \phi_k) = \delta_{jk}(\phi_k, \phi_k), \qquad (13)$$

where the *Kronecker delta* is defined as

$$\delta_{jk} = \begin{cases} 1, & j = k \\ 0, & j \neq k \end{cases} . \qquad (14)$$

Equation (7) now takes the simplified form

$$\begin{aligned} (y, \phi_k) &= a_k \, (\phi_k, \phi_k) \qquad (15) \\ & \qquad (16) \end{aligned}$$

so that

$$a_k = \frac{(y, \phi_k)}{(\phi_k, \phi_k)}. \qquad (17)$$

For orthogonal functions, the matrix $A$ for the normal equations is diagonal, making it trivial to obtain the values $a_k$. You still accumulate roundoff error computing the quantities on the right hand side, but only a single roundoff error solving for $a_k$.

What we will do below is use the $w_i$ and $x_i$ to construct a set of orthogonal polynomials $\phi_k$. Given $y_i$, we can then use (17) to compute the expansion

(2). First we will show that the $\phi_k$ satisfy a three-term recurrence relation. Suppose that $\phi_k(x)$ is monic and has degree exactly $k$ so that its leading term is $x^k$. It is simple to show that [4]

$$x^k = \sum_{j=0}^{k} c_{j,k}\phi_k(x) \tag{18}$$

for some set of $c_{j,k}$. Therefore we can write any polynomial as a weighted sum of the $\phi_k$. With this in mind, write

$$\phi_{k+1} - x\phi_k + b_k\phi_k + c_k\phi_{k-1} = \sum_{j=0}^{k-2} d_{j,k}\phi_j \tag{19}$$

for some $b_k$, $c_k$, and $d_{j,k}$ since $\phi_{k+1} - x\phi_k$ is of degree $k$ at most. Taking inner products with $\phi_{k+1}$, $\phi_k$, $\phi_{k-1}$, and $\phi_j$, $0 \leq j < k-1$ gives

$$\begin{aligned}
(\phi_{k+1}, \phi_{k+1}) - (x\phi_k, \phi_{k+1}) &= 0 \\
-(x\phi_k, \phi_k) + b_k(\phi_k, \phi_k) &= 0 \\
-(x\phi_k, \phi_{k-1}) + c_k(\phi_{k-1}, \phi_{k-1}) &= 0 \\
0 &= d_{j,k}
\end{aligned}$$

Since the inner product (8) is associative (9), we can rewrite these as

$$\begin{aligned}
(\phi_{k+1}, \phi_{k+1}) &= (x\phi_k, \phi_{k+1}) \\
b_k &= \frac{(x\phi_k, \phi_k)}{(\phi_k, \phi_k)} \\
c_k &= \frac{(x\phi_{k-1}, \phi_k)}{(\phi_{k-1}, \phi_{k-1})}
\end{aligned} \tag{20}$$

By setting $k \to k-1$ in the first of these, we obtain the simple relations

$$b_k = \frac{(x\phi_k, \phi_k)}{(\phi_k, \phi_k)} \tag{21}$$

$$c_k = \frac{(\phi_k, \phi_k)}{(\phi_{k-1}, \phi_{k-1})} \tag{22}$$

To summarize,

$$\phi_{k+1} = (x - b_k)\phi_k - c_k\phi_{k-1}, \quad k < N. \tag{23}$$

$$\phi_0 = 1 \tag{24}$$

$$\phi_{i-1} = 0 \tag{25}$$

where $b_k$ is given by (21) and $c_k$ is given by (22). With the initial conditions on $\phi_{-1}$ and $\phi_0$, it is clear that each $\phi_k$ for $k \geq 0$ is monic. Since $d_{j,k} = 0$, the polynomials satisfy the three-term recurrence relation (23) as claimed. Armed with this recurrence, we can compute each $\phi_k(x)$, use (17) to get $a_k$, and build the final solution (2).

There are two things to note about (23). First,

$$\phi_N(x) = \prod_{i=1}^{N}(x - x_i) \tag{26}$$

which vanishes on all of the $x_i$ and would therefore contribute nothing if included in the fit (2). Second, it can be shown [4] that the $k$ zeros of $\phi_k(x)$ are real, simple, and located in the interval spanned by the $x_i$. In particular, this means they are oscillatory over this interval and so care needs to be taken computing and summing them. The Python module is implemented in quadruple precision (using pairs of `float`) [2, 5]. The FORTRAN implementation of this algorithm is given in [6, 7, 8]; a Python2/Python3 implementation is included with this document. The evaluation procedure for (2) uses *Clenshaw's recurrence* [1, 3, 9] because of its numerical stability in computing the fit polynomial and its derivatives. This recurrence is covered in more detail in appendix B.

One advantage of using orthogonal polynomials to fit data is hidden in (17). Having computed a fit up to degree $n$, we can compute the fit of degree $n+1$ by simply computing inner products with $\phi_{k+1}$ in (17) using the recurrence (23) which is $O(N)$ work. Said another way, having computed a fit of order $D$, you immediately know *every* least squares fit of order less than $D$ for *free*.

The `Polyfit` class provides a special method `__call__` to evaluate the fit polynomial and, optionally, its derivatives at a given point, as well as a `coefs()` method to return the Taylor coefficients at a given point. It is important to note that the Taylor coefficients are less accurate than the $a_k$; computing polynomial values using these coefficients will be less accurate than using `polyval()` directly. The `Polyfit` class also provides an `rms_err()` method that returns the RMS residual error for a given fit degree. This information can be used to prevent *overfitting* via statistical tests; in fact, `dpolft` [6] optionally uses this information in a statistical F-test as a possible stopping criterion.

Appendix A compares `polyfit` to a naïve `numpy` implementation using the "standard" normal matrix for $x^k$. The key takeaways from the appendix are:

- For the best results, always scale the $x$ and $y$ values to be $O(1)$. If this is not possible, it is best to use `polyfit`.
- If you have to perform a higher order fit, it is best to use `polyfit`; however, your luck will run out sooner or later.
- Although `polyfit` is slower than `numpy`, it produces far more accurate results.

# A Appendix: Performance Comparisons

Below is a table of examples comparing a naïve `numpy` polynomial fit with the $x^k$ basis functions to `polyfit()`. In the table, $E_{\text{rms}}$ is the RMS error for the fit and $E_{\text{rel}}$ is the maximum relative error for the fit across all the $x_i$. The fit is against 100,000 points for the cubic polynomial

$$2x^3 + x^2 - x + \pi$$

There are 12 cases via 3 sets of criteria:

1. A cubic versus quartic versus 10th degree fit; all terms above cubic should be zero, of course
2. Whether or not the $x$ values are scaled to the unit interval $[-1, 1]$
3. Whether the weights are uniform or chosen to minimize relative error

The runtime for the two cases is also shown. The orthogonal polynomial case is much slower due to being implemented in pure Python; a C version would have much higher performance.

| Function | Order | X-scaling | Weights | Run time | $E_{\text{rms}}$ | $E_{\text{rel}}$ |
|---|---|---|---|---|---|---|
| `polyfit()` | 3 | unscaled | uniform | 0.043 | 2.2e-2 | 4.8e-3 |
| `numpy` | 3 | unscaled | uniform | 0.022 | 2.4e+2 | 2.1e+2 |
| `polyfit()` | 3 | unscaled | relative | 0.044 | 4.4e-2 | 2.2e-16 |
| `numpy` | 3 | unscaled | relative | 0.020 | 3.5e+0 | 1.1e-11 |
| `polyfit()` | 3 | scaled | uniform | 0.043 | 1.9e-16 | 2.2e-16 |
| `numpy` | 3 | scaled | uniform | 0.015 | 2.6e-13 | 1.4e-13 |
| `polyfit()` | 3 | scaled | relative | 0.043 | 1.9e-16 | 2.2e-16 |
| `numpy` | 3 | scaled | relative | 0.012 | 1.8e-13 | 2.0e-13 |
| `polyfit()` | 4 | unscaled | uniform | 0.053 | 2.2e-2 | 2.1e-3 |
| `numpy` | 4 | unscaled | uniform | 0.021 | 1.5e+3 | 1.6e3 |
| `polyfit()` | 4 | unscaled | relative | 0.053 | 3.1e-2 | 2.2e-16 |
| `numpy` | 4 | unscaled | relative | 0.023 | 6.4e+4 | 1.0e-6 |
| `polyfit()` | 4 | scaled | uniform | 0.056 | 1.9e-16 | 2.2e-16 |
| `numpy` | 4 | scaled | uniform | 0.015 | 1.4e-12 | 1.4e-12 |
| `polyfit()` | 4 | scaled | relative | 0.053 | 1.9e-16 | 2.2e-16 |
| `numpy` | 4 | scaled | relative | 0.014 | 3.9e-13 | 5.1e-13 |
| `polyfit()` | 10 | unscaled | uniform | 0.11 | 1.4e-2 | 3.2e-3 |
| `numpy` | 10 | unscaled | uniform | 0.025 | 1.7e7 | 3.1e7 |
| `polyfit()` | 10 | unscaled | relative | 0.11 | 1.5e-2 | 2.2e-16 |
| `numpy` | 10 | unscaled | relative | 0.026 | 1.9e+9 | 3.3e-1 |
| `polyfit()` | 10 | scaled | uniform | 0.11 | 1.9e-16 | 2.2e-16 |
| `numpy` | 10 | scaled | uniform | 0.018 | 2.3e-8 | 4.2e-8 |
| `polyfit()` | 10 | scaled | relative | 0.11 | 1.9e-16 | 2.2e-16 |
| `numpy` | 10 | scaled | relative | 0.018 | 4.9e-9 | 1.0e-8 |

A number of things are apparent from this table:

- The C `polyfit` version is about 2–4 times slower than the `numpy` version implemented in `C` and `FORTRAN`.
- The RMS and relative errors for `polyfit` are about 1,000 to about 1e+12 smaller than the `numpy` implementation.
- For unscaled $x$ values in the range $[0, 99999]$ the `numpy` fit is *awful*.
- For scaled $x$ values in the range $[0, 1]$ the `numpy` fit is much better, but the error is generally 1,000 times higher than for `polyfit`.

- Using relative weights decreases the relative error $E_{\text{rel}}$ significantly. This should come as no surprise.
- Not shown, but for the 10th degree fit with `numpy`, the coefficients above degree 3 are not small; for `polyfit`, they are tiny in all cases.

# B Appendix: Clenshaw's Recurrence

Having determined all of the $a_k$, $b_k$, and $c_k$, we would like to evaluate the fit polynomial and its derivatives. Recall that the fit polynomial is given by (2)

$$\hat{f}(x) = \sum_{k=0}^{D} a_k \phi_k(x).$$

Clenshaw's recurrence is a numerically stable method that efficiently yields the values of $\hat{f}$ and its derivatives (optionally) at a given point $x$. The recurrence is given by

$$z_k = a_k + (x - b_k) z_{k+1} - c_{k+1} z_{k+2} \tag{27}$$
$$c_{D+1} = 0 \tag{28}$$
$$z_{D+1} = 0 \tag{29}$$
$$z_{D+2} = 0 \tag{30}$$

and is applied in the downward direction. Solving (27) for $a_k$ and substituting into the above gives Then

$$\hat{f}(x) = \sum_{k=0}^{D} \phi_k(x) \left[ z_k - (x - b_k) z_{k+1} + c_{k+1} z_{k+2} \right] \tag{31}$$
$$= \sum_{k=0}^{D} z_k \left[ \phi_k(x) - (x - b_{k-1}) \phi_{k-1}(x) + c_{k-1} \phi_{k-2}(x) \right] \tag{32}$$

where the second step follows by grouping terms by $z_k$. Since

$$\phi_k(x) = (x - b_{k-1}) \phi_{k-1}(x) - c_{k-1} \phi_{k-2}(x)$$

by (23), all of the terms vanish except for the $k = 0$ term. Since $\phi_0(x) = 1$, we have

$$\hat{f}(x) = z_0. \tag{33}$$

Now on to the derivatives of $\hat{f}$. It is easy to show that

$$\phi_{k+1}^{(j)} = (x - b_k)\phi_k^{(j)} - c_k\phi_{k-1}^{(j)} + j\phi_k^{(j-1)}$$

where the superscript $(j)$ denotes the derivative of order $j$. If we now define

$$\begin{aligned} z_k^{(j)} &= z_k^{(j-1)} + (x - b_{k-j})z_{k+1}^{(j)} + c_{k-j+1}z_{k+2}^{(j)} && (34)\\ z_k^{(0)} &= a_k, && (35) \end{aligned}$$

then

$$\hat{f} = z_0^{(0)}.$$

To compute the $j$-th derivative of $\hat{f}(x)$, we invoke the recurrence (34) for $j = 0$

$$\begin{aligned} \hat{f}^{(j)} &= \sum_{k=j}^{D} a_k\phi_k^{(j)} \\ &= \sum_{k=j}^{D} \phi_k^{(j)}\left[z_k^{(0)} - (x - b_k)z_{k+1}^{(0)} + c_k z_{k+2}^{(0)}\right] \\ &= \sum_{k=j}^{D} z_k^{(0)}\left[\phi_k^{(j)} - (x - b_{k-1})\phi_{k-1}^{(j)} + c_{k-1}\phi_{k-2}^{(j)}\right] \\ &= j\sum_{k=j}^{D} z_k^{(0)}\phi_{k-1}^{(j-1)} \end{aligned}$$

We can use (34) again after solving for $z_k^{(0)}$:

$$\begin{aligned} &= j\sum_{k=j}^{D} \phi_{k-1}^{(j-1)}\left[z_k^{(1)} - (x - b_{k-1})z_{k+1}^{(1)} + c_k z_{k+2}^{(1)}\right] \\ &= j\sum_{k=j}^{D} z_k^{(1)}\left[\phi_{k-1}^{(j-1)} - (x - b_{k-2})\phi_{k-2}^{(j-1)} + c_{k-2}\phi_{k-3}^{(j-1)}\right] \\ &= j(j-1)\sum_{k=j}^{D} z_k^{(1)}\phi_{k-2}^{(j-2)} \end{aligned}$$

Continuing this way, we finally arrive at

$$\hat{f}^{(j)} = j!\sum_{k=j}^{D} z_k^{(j-1)}\phi_{k-j}$$

$$
\begin{aligned}
&= j! \sum_{k=j}^{D} \phi_{k-j} \left[ z_k^{(j)} - (x - b_{k-j}) z_{k+1}^{(j)} + c_{k-j+1} z_{k+2}^{(j)} \right] \\
&= j! \sum_{k=j}^{D} z_k^{(j)} \left[ \phi_{k-j} - (x - b_{k-j-1}) \phi_{k-j-1} + c_{k-j-1} \phi_{k-j-2} \right] \\
&= j!\, z_j^{(j)}
\end{aligned}
$$

since only the $\phi_0$ term remains. Note that while computing the derivative of order $j$, we obtain all of the derivatives of order less than $j$ for *free*.

# C Appendix: Source Code for the Python Reference Implementation

```
#!/usr/bin/env pypy3

"""
quad precision orthogonal polynomial least squares fit

see polyfit.pdf and the code in examples/
"""

## {{{ prologue
from __future__ import print_function

## pylint: disable=invalid-name,bad-whitespace,useless-object-inheritanc

import math
import time

__all__ = ["Polyfit"]
## }}}
## {{{ quad precision routines from ogita et al
def twosum(a, b):
    "6 flops, algorithm 3.1 from ogita"
    x = a + b
    z = x - a
    y = (a - (x - z)) + (b - z)
```

```
    return x, y

def twodiff(a, b):
    "6 flops, subtraction version of twosum()"
    x = a - b
    z = x - a
    y = (a - (x - z)) - (b + z)
    return x, y

def split(a, FACTOR = 1. + 2. ** 27):
    "4 flops, algorithm 3.2 from ogita"
    c = FACTOR * a
    x = c - (c - a)
    y = a - x
    return x, y

def twoproduct(a, b):
    "23 flops, algorithm 3.3 from ogita"
    x      = a * b
    a1, a2 = split(a)
    b1, b2 = split(b)
    y      = a2 * b2 - (x - a1 * b1 - a2 * b1 - a1 * b2)
    return twosum(x, y)

def sum2s(p):
    "7n-1 flops, algorithm 4.1 from ogita"
    pi, sigma = p[0], 0.
    for i in range(1, len(p)):
        pi, q  = twosum(pi, p[i])
        sigma += q
    return twosum(pi, sigma)

def vsum(p):
    "6(n-1) flops, algorithm 4.3 from ogita"
    im1 = 0
    for i in range(1, len(p)):
        p[i], p[im1] = twosum(p[i], p[im1])
        im1 = i
```

```
    return p

def sumkcore(p, K):
    "6(K-1)(n-1) flops, algorithm 4.8 from ogita"
    for _ in range(K - 1):
        p = vsum(p)
    return p

def sumk(p, K):
    "(6K+1)(n-1)+6 flops, algorithm 4.8 from ogita"
    p = sumkcore(p, K)
    return sum2s(p)

def vectorsum(vec):
    "19n-13 flops, sumk() with K=3"
    return sumk(vec, K=3)
## }}}
## {{{ utility functions
def vappend(vec, x):
    "append quad to vector"
    vec.extend(x)

def zero():
    "yup"
    return (0., 0.)

def one():
    "yup"
    return (1., 0.)

def to_quad(x):
    "float to quad"
    return x if isinstance(x, tuple) else (float(x), 0.)

def quad_to_float(x):
    "quad to float"
    return x[0] if isinstance(x, tuple) else float(x)
## }}}
```

```
## {{{ quad precision arithmetic
def add(x, y):
    "14 flops"
    x, xx = x
    y, yy = y
    z, zz = twosum(x, y)
    return twosum(z, zz + xx + yy)

def sub(x, y):
    "14 flops"
    x, xx = x
    y, yy = y
    z, zz = twodiff(x, y)
    return twosum(z, zz + xx - yy)

def mul(x, y):
    "33 flops"
    x, xx = x
    y, yy = y
    z, zz = twoproduct(x, y)
    zz   += xx * y + x * yy
    return twosum(z, zz)

def div(x, y):
    "36 flops, from dekker"
    x, xx = x
    y, yy = y
    c     = x / y
    u, uu = twoproduct(c, y)
    cc    = (x - u - uu + xx - c * yy) / y
    return twosum(c, cc)

def sqrt(x):
    "35 flops, from dekker"
    x, xx = x
    if not (x or xx):
        return zero()
    c     = math.sqrt(x)
```

```
    u, uu = twoproduct(c, c)
    cc    = (x - u - uu + xx) * 0.5 / c
    return twosum(c, cc)
## }}}
## {{{ orthogonal polynomial least squares fitting
def polyfit(xv, yv, wv, D):
    ## pylint: disable=too-many-locals
    """
    orthogonal polynomial fit

    given x values xv[], y values yv[], positive weights wv[],
    and a maximum fit degree D, compute the least squares
    fits up to degree D
    """
    ## fit: y_k \approx \sum_{k=0}^D a_k \phi_k(x)

    ## inner product: (f, g) = \sum_{i=0}^{N-1} w_i f(x_i) g(x_i)

    ## recurrence:
    ##      \phi_{k+1}(x) = (x - b_k) \phi_k(x) - c_k \phi_{k-1}(x)

    ## g_k = (\phi_k, \phi_k)
    ## b_k = (x \phi_k, \phi_k) / g_k
    ## c_k = g_k / g_{k-1}
    ## a_k = (y, \phi_k) / g_k
    assert len(xv) == len(yv) == len(wv)
    assert min(wv) > 0
    xv = [to_quad(x) for x in xv]
    yv = [to_quad(y) for y in yv]
    wv = [to_quad(w) for w in wv]
    N  = len(xv)
    a  = [ ]        ## a_k fit coefficients
    b  = [ ]        ## b_k in recurrence
    g  = [one()]    ## g_k poly 2-norm
    c  = [ ]        ## c_k in recurrence
    e  = [ ]        ## rms fit errors

    ret = {         ## fit object
```

```
            "a": a,
            "b": b,
            "c": c,
            "e": e,
            "d": D,
            "n": N
        }

        phi_km1 = [zero()] * N  ## \phi_{-1}
        phi_k   = [one()]  * N  ## \phi_0
        for k in range(D+1):  ## pylint: disable=unused-variable
            ## vectors to hold pieces of inner products
            avec = [ ]
            bvec = [ ]
            gvec = [ ]
            ## compute inner products for a_k, b_k, c_k, g_k
            for i in range(N):
                s = mul(wv[i], phi_k[i])
                t = mul(s, phi_k[i])
                ## a_k += wv[i] * yv[i] * phi_k[i]
                vappend(avec, mul(s, yv[i]))
                ## b_k += wv[i] * xv[i] * phi_k[i] * phi_k[i]
                vappend(bvec, mul(t, xv[i]))
                ## g_k += wv[i] * phi_k[i] * phi_k[i]
                vappend(gvec, t)
            ## turn vectors back to scalars and normalize
            g_k = vectorsum(gvec)
            a_k = div(vectorsum(avec), g_k)
            b_k = div(vectorsum(bvec), g_k)
            c_k = div(g_k, g[-1])
            a.append(a_k)
            b.append(b_k)
            c.append(c_k)
            g.append(g_k)

            ## subtract projection a_k \phi_k from yv, leaving the
            ## residuals in yv. dpolft does this and it does actually
            ## help. plus it enables the rms calculation below.
```

```
        for i in range(N):
            ## yv[i] -= a_k * phi_k[i]
            yv[i] = sub(yv[i], mul(a_k, phi_k[i]))

        ## compute the (unweighted) rms error in the fit
        evec = [ ]
        for i, r in enumerate(yv):
            ## err += res[i] * res[i]
            vappend(evec, mul(r, r))
        erms = quad_to_float(
            sqrt(div(vectorsum(evec), to_quad(N)))
        )
        e.append(erms)

        ## update polys using recurrence
        if k != D:
            for i in range(N):
                ## \phi_{k+1} = (x - b_k) \phi_k - c_k \phi_{k-1}
                phi_kp1    = sub(
                    mul(sub(xv[i], b_k), phi_k[i]),
                    mul(c_k, phi_km1[i])
                )
                phi_km1[i] = phi_k[i]
                phi_k[i]   = phi_kp1

    c.append(zero())    ## for polyfit_val()

    return ret
## }}}
## {{{ least squares polynomial evaluation
def polyfit_val(fit, x, deg=-1, nderiv=0, extended=False):
    ## pylint: disable=too-many-locals
    """
    return the value of the fit for degree deg and nderiv
    derivatives at the point x. if deg is negative, use the
    highest degree of the fit. if nderiv is negative, compute
    all derivatives of the fit

```

```
    returns a list of the function values and its derivatives
    """
    x = to_quad(x)
    a, b, c = fit["a"], fit["b"], fit["c"]
    if nderiv < 0:
        nderiv = len(a) - 1
    if deg < 0:
        deg = len(a) - 1

    ret  = [ ]
    ## init z^{(j-1)} and z^{(j)}
    zjm1 = a[:deg+2] + [zero(), zero()]
    zj   = [zero()]  * (deg + 3)
    fac  = one()
    for j in range(min(deg, nderiv) + 1):
        if j > 1:
            fac = mul(fac, to_quad(j))
        ## compute the next lowest z_k^{(j)}
        for k in range(deg, j - 1, -1):
            t = k - j
            ## (x-b[t]) * zj[k+1] - c[t+1] * zj[k+2]j
            tmp = sub(
                mul(sub(x, b[t]), zj[k+1]),
                mul(c[t+1], zj[k+2])
            )
            zj[k] = add(zjm1[k], tmp)
        ## save off the function value or derivative
        val = mul(fac, zj[j])
        ret.append(val if extended else quad_to_float(val))
        ## update z vectors
        zjm1 = zj
        zj   = [zero()] * (deg + 3)
    if nderiv > deg:
        ret.extend([zero() if extended else 0.] * (nderiv - deg))
    return ret
## }}}
## {{{ least squares polynomial coefficients
def polyfit_cofs(fit, deg=-1, x0=0., extended=False):
```

```
    """
    return taylor coefficients of fit for the given degree deg
    about x0.
    """
    derivs = polyfit_val(fit, x0, deg=deg, nderiv=-1, extended=True)
    fac    = one()
    for i in range(1, len(derivs)):
        fac = div(fac, to_quad(i))
        derivs[i] = mul(derivs[i], fac)
    return \
        derivs if extended else [quad_to_float(d) for d in derivs]
## }}}
## {{{ least squares polynomial per-degree errors
def polyfit_err(fit, degree):
    "return the rms errors for each fit degree"
    return fit["e"][degree]
## }}}
## {{{ fetch fit params
def polyfit_npoints(fit):
    "return number of data points in fit"
    return fit["n"]

def polyfit_maxdeg(fit):
    "return max degree of fit"
    return fit["d"]
## }}}
## {{{ class-based interface
class Polyfit(object):
    "polynomial fitting class"

    def __init__(self, maxdeg, xv, yv, wv):
        """
        given x- and y-values in xv[] and yv[], along with
        positive fit weights in wv[], compute all least-squares
        fits up to degree maxdeg
        """
        t0         = time.time()
        self.xv    = xv
```

```
        self.yv    = yv
        self.wv    = wv
        self._fit  = polyfit(xv, yv, wv, D=maxdeg)
        self._time = time.time() - t0

    def __call__(self, x, degree=None, nderiv=0):
        """
        evaluate poly and (optionally) some of its derivatives.
        if degree is None, return the values for the maximum
        fit degree

        if nderiv is negative, return the polynomial value and
        all derivatives as a list. this is the default. if
        nderiv is 0, return the scalar polynomial value. this
        is the default
        """
        if degree is None:
            degree = self.maxdeg()
        nd  = self.maxdeg() if nderiv < 0 else nderiv
        ret = polyfit_val(self._fit, x, degree, nd)
        return ret if nderiv else ret[0]

    def close(self):
        "finalize self (no-op for this version)"

    def coefs(self, degree=None, x0=0):
        """
        return the coefficients for fit degree degree about
        (x-x0). if degree is None, use the maximum fit degree
        """
        if degree is None:
            degree = self.maxdeg()
        return polyfit_cofs(self._fit, degree, x0)

    def maxdeg(self):
        "return max fit degree"
        return polyfit_maxdeg(self._fit)

```

```
    def npoints(self):
        "return number of data points used to perform the fit"
        return polyfit_npoints(self._fit)

    def rel_err(self, degree=None):
        "return the max relative fit error across all x values"
        if degree is None:
            degree = self.maxdeg()
        err = -1.
        for x, exp in zip(self.xv, self.yv):
            obs = self(x, degree=degree)
            if exp:
                rel = abs(obs / exp - 1.)
                if rel > err:
                    err = rel
        return err

    def rms_err(self, degree=None):
        """
        return the residual rms fit residual for degree degree.
        if degree is None, use the maximum fit degree
        """
        if degree is None:
            degree = self.maxdeg()
        return polyfit_err(self._fit, degree)

    def runtime(self):
        "return the time it took to perform the fit"
        return self._time
## }}}

## vim: ft=python
```

# References


[1] CW Clenshaw, “Curve Fitting with a Digital Computer,” The Computer Journal, Volume 2, Issue 4, Pages 170–173, 1960. Online at

https://academic.oup.com/comjnl/article/2/4/170/470550

[2] TJ Dekker, “A Floating-Point Technique for Extending the Available Precision,” Numer. Math. 18, 224–242, 1971.

[3] GE Forsythe, “Generation and Use of Orthogonal Polynomials for Data-Fitting with a Digital Computer,” Journal of the Society for Industrial and Applied Mathematics, vol 5, no 2 74-88, June 1957.

[4] FB Hildebrand, Introduction to Numerical Analysis, 2nd ed. Dover, 1974.

[5] T Ogita, SM Rump, and S Oishi, “Accurate Sum and Dot Product,” Siam J Sci Comp, 26(6), 2005. Online at https://www.tuhh.de/ti3/paper/rump/OgRuOi05.pdf

[6] LF Shampine, SM Davenport, and RE Huddleston, https://netlib.org/slatec/src/dpolft.f to compute a linear least-squares orthogonal polynomial fit.

[7] LF Shampine, SM Davenport, and RE Huddleston, https://netlib.org/slatec/src/dp1vlu.f to evaluate the fit polynomial and its derivatives $p^{(j)}(x)$ at a given point $x$.

[8] LF Shampine, SM Davenport, and RE Huddleston, https://netlib.org/slatec/src/dpcoef.f To return the coefficients $c_k$ of the polynomial as $\sum_k c_k(x - x_0)^k$ at a given point $x_0$.

[9] FJ Smith, “An Algorithm for Summing Orthogonal Polynomial Series and their Derivatives with Applications to Curve-Fitting and Interpolation,” Mathematics of Computation, vol 19, no 89 33-36, April 1965. Online at https://www.ams.org/journals/mcom/1965-19-089/S0025-5718-1965-0172445-6/S0025-5718-1965-0172445-6.pdf