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

Mark Hays
`https://github.com/minmus-9`

October 3, 2021

Abstract

In this note I present the Python2, Python3, and `C` `polyfit` package that implements quad-precision [2, 6] least-squares polynomial fitting using orthogonal polynomials [5, 7, 8, 9]. Pure Python and `C` versions are provided; they are slower than the traditional approach (primarily due to being quad-precision), but are numerically more stable and accurate [1, 3, 10] than the traditional approach. A listing of the source code for the 260 SLOC Python reference implementation is included in appendix B. A much faster `C` version also ships with this package; there is a Python ctypes-based interface called `cpolyfit` that integrates this fast version into Python. Both Python interfaces support Python 2.7 and 3.6+.

1 TL;DR Quick Start

This package consists of 4 modules:

- The 260 SLOC pure 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 except that it uses `array.array` objects instead of `list` objects.
- The `polyplus.py` module that implements integration of fit polynomials in quad precision and Gaussian quadrature to reduce weighted sums to much shorter sums.

Unlike `polyfit.py`, the `C` and `ctypes` APIs do not have any inline comments. The `C` version exactly follows the reference implementation and the comments in the reference implementation also apply to the `C` version. The `ctypes` version almost exclusively consists of glue code to `libpolyfit.so` and isn't interesting from an algorithmic viewpoint.

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

The rest of this section contains the listings for the examples `ex1.py` and `ex2.c` to get you going. Both examples do the same thing. There are several other example scripts in the `examples/` directory that include

- Comparisons with `numpy` fitting,
- Integration of the fit polynomial, and
- Gaussian quadrature on a discrete set of points

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

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

"example usage of the polyfit api"

from __future__ import print_function as _

## pylint: disable=invalid-name,bad-whitespace

import math
import sys

sys.path.insert(0, "..")

from polyfit import PolyfitPlan ## pylint: disable=wrong-import-position

def flist(l):
    "format a list to 15 decimal places"
    if not isinstance(l, list):
        l = [l]
    return " ".join("%.15e" % x for x in l)

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

    ## poly coefficients to fit, highest degree first
    cv = [2, 1, -1, math.pi]
    #cv = [1, -2, 1]

    ## evaluate the polynomial above 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
    N  = 10000
    xv = [x for x in range(N)]
    yv = [pv(x) for x in xv]

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

    ##     relative to minimize the relative residual
```

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

    ## perform the fit
    D    = len(cv) - 1
    plan = PolyfitPlan(D, xv, wv)
    fit  = plan.fit(yv)
    ev   = fit.evaluator()

    ## print the fit stats
    deg = plan.maxdeg()
    print("maxdeg", deg)
    print("points", plan.npoints())

    ## print per-degree rms errors
    print("erms  ", flist(fit.rms_errors()))

    ## print a few values
    for i in range(4):
        print("value  %.1f %s" % (xv[i], flist(ev(xv[i], nder=-1))))

    ## print value and all derivatives for all degrees
    for i in range(D + 1):
        print("deg    %d %s" % (i, flist(ev(xv[0], deg=i, nder=-1))))

    ## print coefficients for all degrees about (x - xv[0])
    for i in range(D + 1):
        print("coefs  %d %s" % (i, flist(ev.coefs(xv[0], i))))

    ## coefs halfway through
    print("coefs ", flist(ev.coefs(xv[N >> 1], deg)))

if __name__ == "__main__":
    demo()

## EOF
```

Following is the C example `ex2.c` that corresponds to the Python `ex1.py`

```
/****************************************************************
 * ex2.c - polyfit demo
 */

#include <math.h>
#include <stdio.h>
#include <sys/time.h>

#include "polyfit.h"

#ifndef M_PI
#define M_PI 0
#define USE_ACOS
#endif

#define N 10000
```

```
#define D 3
double xv[N], yv[N], wv[N];

/* poly coefficients to fit, highest degree first */
double cv[D + 1] = { 2, 1, -1, M_PI };

void init() {
    double y;
    int    i, j;

#ifdef USE_ACOS
    cv[D] = acos(-1);
#endif
    for (i = 0; i < N; i++) {

        /* evaluate the poly to fit using horner's method */
        for (y = 0, j = 0; j <= D; j++) {
            y *= i;
            y += cv[j];
        }
        /* define xv[], yv[], and wv[] for the fit */
        xv[i] = i;
        yv[i] = y;
#if 1
        wv[i] = 1;
#else
        /* minimize relative residual */
        wv[i] = 1. / (y * y); /* y != 0 for this example poly */
#endif
    }
}

int main(int argc, char *argv[]) {
    void  *plan, *fit, *ev;
    int    i, j, n;
    double coefs[D + 1], d[D + 1];
    const double *t;

    /* fill in xv, yv, and, wv */
    init();

    /* create the fit plan */
    if ((plan = polyfit_plan(D, xv, wv, N)) == NULL) {
        perror("polyfit_plan");
        return 1;
    }

    /* compute the fit */
    if ((fit = polyfit_fit(plan, yv)) == NULL) {
        perror("polyfit_fit");
        return 1;
    }

    /* make an evaluator */
    if ((ev = polyfit_evaluator(fit)) == NULL) {
        perror("polyfit_evaluator");
```

```
        return 1;
    }

    /* print fit stats */
    if ((n = polyfit_maxdeg(plan)) < 0) {
        perror("polyfit_maxdeg");
        return 1;
    }
    printf("maxdeg %d\n", n);
    if ((n = polyfit_npoints(plan)) < 0) {
        perror("polyfit_npoints");
        return 1;
    }
    printf("points %d\n", n);

    /* print per-degree rms errors */
    if ((t = polyfit_rms_errs(fit, NULL)) == NULL) {
        perror("polyfit_rms_errs");
        return 1;
    }
    printf("erms  ");
    for (i = 0; i <= D; i++) {
        printf(" %.15e", t[i]);
    }
    printf("\n");

    /* print a few values */
    for (i = 0; i < 4; i++) {
        if (polyfit_eval(ev, xv[i], D, d, D) < 0) {
            perror("polyfit_eval");
            return 1;
        }
        printf("value  %.1f", xv[i]);
        for (j = 0; j <= D; j++) {
            printf(" %.15e", d[j]);
        }
        printf("\n");
    }

    /* print value and all derivatives for all degrees */
    for (i = 0; i <= D; i++) {
        if (polyfit_eval(ev, xv[0], i, d, -1) < 0) {
            perror("polyfit_eval");
            return 1;
        }
        printf("deg    %d", i);
        for (j = 0; j <= i; j++) {
            printf(" %.15e", d[j]);
        }
        printf("\n");
    }

    /* print coefficients for all degrees about (x - xv[0]) */
    for (i = 0; i <= D; i++) {
        if (polyfit_coefs(ev, xv[0], i, coefs) < 0) {
            perror("polyfit_coefs");
```

```
            return 1;
        }
        printf("coefs  %d", i);
        for (j = 0; j <= i; j++) {
            printf(" %.15e", coefs[j]);
        }
        printf("\n");
    }

    /* coefs halfway through */
    if (polyfit_coefs(ev, xv[N>>1], D, coefs) < 0) {
        perror("polyfit_coefs");
        return 1;
    }
    printf("coefs ");
    for (i = 0; i <= D; i++) {
        printf(" %.15e", coefs[i]);
    }
    printf("\n");

    /* free the fit objects */
    polyfit_free(ev);
    polyfit_free(fit);
    polyfit_free(plan);
    return 0;
}

/* EOF */
```

2 API Documentation

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

```
NAME
    polyfit - quad precision orthogonal polynomial least squares fitting

CLASSES
    PolyfitBase(__builtin__.object)
        PolyfitEvaluator
        PolyfitFit
        PolyfitPlan

    class PolyfitEvaluator(PolyfitBase)
     |  returned by PolyfitFit.evaluator(). this object evaluates
     |  the fit polynomial and its derivatives, and also returns
     |  its coefficients in powers of (x - x0) for given x0.
     |
     |  Method resolution order:
     |      PolyfitEvaluator
     |      PolyfitBase
     |      __builtin__.object
     |
     |  Methods defined here:
     |
     |  __call__(self, x, deg=-1, nder=0)
```

```
 |      given a point x, a least squares fit degree deg,
 |      and a desired number of derivatives to compute nder,
 |      calculate and return the value of the polynomial and
 |      any requested derivatives.
 |
 |      if deg is negative, use maxdeg instead. if nder is
 |      negative, use the final value of deg; otherwise, compute
 |      nder derivatives of the least squares polynomial of
 |      degree deg.
 |
 |      returns a list whose first element is the value of the
 |      least squares polynomial of degree deg at x. subsequent
 |      elements are the requested derivatives. if zero
 |      derivatives are requested, the scalar function value is
 |      returned.
 |
 |  __init__(self, data)
 |
 |  coefs(self, x0, deg=-1)
 |      return the coefficients of the fit polynomial of degree
 |      deg about (x - x0). if deg is negative, use maxdeg
 |      instead.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from PolyfitBase:
 |
 |  close(self)
 |      deallocate resources, a no-op for this impementation
 |
 |  to_data(self)
 |      return low level, serializable, class-specific data
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from PolyfitBase:
 |
 |  from_data(cls, data) from __builtin__.type
 |      return instance from serializable data

class PolyfitFit(PolyfitBase)
 |  orthogonal polynomial fitter returned by PolyfitPlan.fit()
 |
 |  Method resolution order:
 |      PolyfitFit
 |      PolyfitBase
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __init__(self, plan=None, yv=None, data=None)
 |
 |  evaluator(self)
 |      return a PolyfitEvaluator for this fit.
 |
 |  residuals(self)
 |      return the list of residuals for the maxdeg fit.
 |
```

```
 |  rms_errors(self)
 |      return a list of rms errors, one per fit degree. use
 |      them to detect overfitting.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from PolyfitBase:
 |
 |  close(self)
 |      deallocate resources, a no-op for this impementation
 |
 |  to_data(self)
 |      return low level, serializable, class-specific data
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from PolyfitBase:
 |
 |  from_data(cls, data) from __builtin__.type
 |      return instance from serializable data

class PolyfitPlan(PolyfitBase)
 |  orthogonal polynomial least squares planning class. you must
 |  create one of these prior to fitting; it can be reused for
 |  multiple fits of the same xv[] and wv[].
 |
 |  Method resolution order:
 |      PolyfitPlan
 |      PolyfitBase
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __init__(self, maxdeg=None, xv=None, wv=None, data=None)
 |      given x values in xv[] and positive weights in wv[],
 |      make a plan to perform least squares fitting up to
 |      degree maxdeg.
 |
 |      this is code for "compute everything need to calculate
 |      an expansion in xv- and wv-specific orthogonal
 |      polynomials".
 |
 |  fit(self, yv)
 |      given a set of y values in yv[], compute all least
 |      squares fits to yv[] up to degree maxdeg. returns
 |      a PolyfitFit object.
 |
 |  maxdeg(self)
 |      return the maximum fit degree
 |
 |  npoints(self)
 |      return the number of fit points
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from PolyfitBase:
 |
 |  close(self)
 |      deallocate resources, a no-op for this impementation
```

```
     |
     |  to_data(self)
     |      return low level, serializable, class-specific data
     |
     |  ----------------------------------------------------------------------
     |  Class methods inherited from PolyfitBase:
     |
     |  from_data(cls, data) from __builtin__.type
     |      return instance from serializable data

FUNCTIONS
    add(x, y)
        add two quads, 14 flops

    div(x, y)
        divide 2 quads, 36 flops, from dekker

    mul(x, y)
        multiply 2 quads, 33 flops

    one()
        return quad precision 1

    polyfit_coefs(plan, fit, x0=0.0, deg=-1)
        given a plan, a set of expansion coefficients generated
        by polyfit_fit, a center point x0, and a least squares
        fit degree, return the coefficients of powers of (x - x0)
        with the highest powers first. if deg is negative (the
        default), use maxdeg instead. the coefficients are quad
        precision.

    polyfit_eval(plan, fit, x, deg=-1, nder=0, scalar=True)
        given a plan, a fit data object returned by
        polyfit_fit, a point x, a least squares fit degree deg,
        and a desired number of derivatives to compute nder,
        calculate and return the value of the polynomial and
        any requested derivatives.

        if deg is negative, use maxdeg instead. if nder is
        negative, use the final value of deg; otherwise, compute
        ndeg derivatives of the least squares polynomial of
        degree deg.

        returns a list of quads whose first element is the value
        of the least squares polynomial of degree deg at x.
        subsequent elements are the requested derivatives. if zero
        derivatives are requested, the scalar function value
        is returned. if x is a quad, quads are returned.

    polyfit_fit(plan, yv)
        given a previously generated plan and a set of y values
        in yv[], compute all least squares fits to yv[] up to
        degree maxdeg.

        returns a json-serializable fit data object.
```

```
    polyfit_maxdeg(plan)
        return the maximum possible fit degree

    polyfit_npoints(plan)
        return the number of data points being fit

    polyfit_plan(maxdeg, xv, wv)
        given x values in xv[] and positive weights in wv[],
        make a plan to perform least squares fitting up to
        degree maxdeg.

        returns a plan object than can be json-serialized.

    sqrt(x)
        square root of a quad, 35 flops, from dekker

    sub(x, y)
        subtract 2 quads, 14 flops

    to_float(x)
        convert quad to float

    to_quad(x)
        convert float or quad to quad

    vappend(vec, x)
        append quad precision number to vector. this is used
        for vectorsum():

            v = [ ]
            for x in y:
                quad = ...
                vappend(v, quad)
            s = vectorsum(v)

        vectorsum() is more accurate than using add() in a loop.

    vectorsum(vec)
        accurately sum a vector of floats, 19n-13 flops

    zero()
        return quad precision 0

DATA
    __all__ = ['PolyfitPlan', 'PolyfitFit', 'PolyfitEvaluator', 'polyfit_p...
```

Following is the `polyfit.h` C API header.

```
/**********************************************************************
 * polyfit.h - quad-precision orthogonal polynomial least squares
 */

#ifndef polyfit_h__
#define polyfit_h__

#ifdef __cplusplus
```

```
extern "C" {
#endif

/***************************************************************
 * given x values in xv[] and positive weights in wv[],
 * make a plan to perform least squares fitting up to
 * degree maxdeg and return a plan object. returns NULL
 * and sets errno on error.
 */
extern void *polyfit_plan(
    const int maxdeg,
    const double * const xv,
    const double * const wv,
    const int npoints
);

/***************************************************************
 * given a set of y values in yv[], compute all least
 * squares fits to yv[] up to degree maxdeg and return
 * a fit object. returns NULL and sets errno on error.
 */
extern void *polyfit_fit(
    const void * const plan,
    const double * const yv
);

/***************************************************************
 * given a fit, return an evaluator that can (a) compute the
 * fit polynomial and its derivatives and (b) can compute
 * coefficients of the polynomial about a given point x0.
 * returns NULL and sets errno on error.
 */
extern void *polyfit_evaluator(
    const void * const fit
);

/***************************************************************
 * given a point x, a least squares fit degree degree,
 * and a desired number of derivatives to compute nderiv,
 * calculate and return the value of the polynomial and
 * any requested derivatives.
 *
 * if degree is negative, use maxdeg instead. if nderiv is
 * negative, use the final value of deg; otherwise, compute
 * nderiv derivatives of the least squares polynomial of
 * degree deg.
 *
 * the derivatives array contains the polynomial value first,
 * followed by any requested derivatives.
 *
 * returns 0 on success. on failure returns -1 and sets errno:
 *   EINVAL - evaluator is not an evaluator.
 *          - derivatives is NULL
 */
extern int polyfit_eval(
    void * const evaluator,
```

```
    const double x,
    const int degree,
    double * const derivatives,
    const int nderiv
);

/****************************************************************
 * return the coefficients of the fit polynomial of degree
 * degree about (x - x0). if degree is negative, use maxdeg
 * instead.
 *
 * returns 0 on success. on failure returns -1 and sets errno:
 *   EINVAL - evaluator is not an evaluator.
 *          - coefs is NULL
 */
extern int polyfit_coefs(
    void * const evaluator,
    const double x0,
    const int degree,
    double * const coefs
);

/****************************************************************
 * return the maximum fit degree
 *
 * on failure returns -1 and sets errno:
 *   EINVAL - plan object is not recognized
 */
extern int polyfit_maxdeg(
    const void * const plan
);

/****************************************************************
 * return the number of fit points
 *
 * on failure returns -1 and sets errno:
 *   EINVAL - plan object is not recognized
 */
extern int polyfit_npoints(
    const void * const plan
);

/****************************************************************
 * return the list of residuals for the maxdeg fit.
 *
 * returns 0 on success. on failure returns -1 and sets errno:
 *   EINVAL - fit is not a fit.
 */
extern double *polyfit_resids(
    const void * const fit,
    double * const resids
);

/****************************************************************
 * return a list of rms errors, one per fit degree. use them
 * to detect overfitting.
```

```
 *
 * returns 0 on success. on failure returns -1 and sets errno:
 *   EINVAL - fit is not a fit.
 */
extern double *polyfit_rms_errs(
    const void * const fit,
    double * const errs
);

/*************************************************************
 * free a plan, fit, or evaluator object
 *
 * returns 0 on success. on failure returns -1 and sets errno:
 *   EINVAL - unrecognized object type.
 *          - polyfit_object is NULL.
 */
extern int polyfit_free(
    void * const polyfit_object
);

#ifdef __cplusplus
};
#endif

#endif

/** EOF */
```

3 The Theory

Following the development in [5], suppose that we have N ordered pairs of data points

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

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

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

so that

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

with $D < N-1$. The functions ϕ_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 weighted error E:

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

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 positives semidefinite quadratic form and so a minimum can be found by setting the gradient of E with respect to the a_j to zero:

$$\frac{\partial E}{\partial a_j} = 0, \quad j \le 0 \le D \tag{6}$$

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

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

After a little rearrangement, this becomes

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

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

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

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

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

These two are called the *normal equations* and are the solution to the $(D+1) \times (D+1)$ linear system

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

for the coefficient vector a. In the common case with $w_i = 1$, the matrix A 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. Using special linear solvers such as Cholesky decomposition, SVD, and friends [4] are strongly recommended for this approach.

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

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

where the *Kronecker delta* is defined as

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

Equation (7) now takes the simplified form

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

so that

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

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 ϕ_k. Given y_i, we can then use (17) to compute the expansion (2).

First we will show that the ϕ_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 [5]

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

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

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

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

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

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,

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

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

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

$$
\begin{aligned}
b_k &= \frac{(x\phi_k, \phi_k)}{(\phi_k, \phi_k)} \\
c_k &= \frac{(\phi_k, \phi_k)}{(\phi_{k-1}, \phi_{k-1})}
\end{aligned} \tag{26}
$$

With the initial conditions on ϕ_{-1} and ϕ_0, it is clear that each ϕ_k for $k \geq 0$ is monic. Since $d_{jk} = 0$, 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,

$$
\phi_N(x) = \prod_{i=1}^{N} (x - x_i) \tag{27}
$$

vanishes on all of the x_i and would therefore contribute nothing if included in the fit (2).

Second, it can be shown [5] that the k zeros of $\phi_k(x)$ are real, simple, and located in the interval spanned by the x_i. This fact is proven in appendix F. 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 `double`) [2, 6]. The FORTRAN implementation of this algorithm is given in [7, 8, 9]; a Python2/Python3 implementation is included with this document. The evaluation procedure for (2) uses *Clenshaw's recurrence* [1, 3, 10] because of its numerical stability in computing the fit polynomial and its derivatives. This recurrence is covered in more detail in appendix E.

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

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 (potentially far less accurate) than using `__call__()` 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 *over-fitting* via statistical tests; in fact, `dpolft` [7] optionally uses this information in a statistical F-test as an optional stopping criterion.

Appendix A compares `polyfit` to a naïve `numpy` implementation using the "standard" normal matrix for x^k using Cholesky decomposition [4] for stability. This listing for the `numpy` code is provided in appendix C.

A Appendix: Performance Comparisons

Below is a table of examples comparing a naïve `numpy` polynomial fit with the x^k basis functions to `polyfit()`. This code for this test is available in `examples/ex3.py` and the `numpy` test modules is in `examples/np.py`. In the table, E_{rms} is the RMS residual for the fit and E_{rel} is the maximum relative error for the fit across all the x_i. The fit is for 100,000 points with 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 slower primarily due to being implemented in quadruple precision (orthogonal polynomial fitting is acutally inherently slower than directly computing moments).

Function	Order	X-scaling	Weights	E_{rms}	E_{rel}
`polyfit()`	3	unscaled	uniform	2.2e-2	4.8e-3
`numpy`	3	unscaled	uniform	4.5e+0	4.2e+0
`polyfit()`	3	unscaled	relative	4.4e-2	2.2e-16
`numpy`	3	unscaled	relative	1.6e-1	2.4e-15
`polyfit()`	3	scaled	uniform	1.9e-16	2.2e-16
`numpy`	3	scaled	uniform	3.4e-14	4.1e-14
`polyfit()`	3	scaled	relative	1.9e-16	2.2e-16
`numpy`	3	scaled	relative	2.4e-15	2.0e-15
`polyfit()`	4	unscaled	uniform	2.2e-2	3.1e-3
`numpy`	4	unscaled	uniform	1.4e+1	1.6e+1
`polyfit()`	4	unscaled	relative	3.1e-2	2.2e-16
`numpy`	4	unscaled	relative	1.6e-14	2.4e-15
`polyfit()`	4	scaled	uniform	1.9e-16	2.2e-16
`numpy`	4	scaled	uniform	3.4e-14	4.1e-14
`polyfit()`	4	scaled	relative	1.9e-16	2.2e-16
`numpy`	4	scaled	relative	2.4e-15	2.0e-15
`polyfit()`	10	unscaled	uniform	1.4e-2	3.2e-3
`numpy`	10	unscaled	uniform	1.2e+5	2.3e+5
`polyfit()`	10	unscaled	relative	1.5e-2	2.2e-16
`numpy`	10	unscaled	relative	3.9e+2	1.3e-11
`polyfit()`	10	scaled	uniform	1.9e-16	2.2e-16
`numpy`	10	scaled	uniform	4.7e-10	8.2e-10
`polyfit()`	10	scaled	relative	1.9e-16	2.2e-16
`numpy`	10	scaled	relative	1.1e-9	1.7e-9

A number of things are apparent from this table:

- The RMS and relative errors for `polyfit` are much smaller.
- 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 much higher than for `polyfit`.
- Using relative weights decreases the relative error E_{rel} significantly. This should come as no surprise.
- Not shown, but for the 10th degree fit with `numpy`, the coefficients above degree 3 are up to $O(10^{-2})$; for `polyfit`, they are $O(10^{-12})$.

The next test gives the highest degree fit D obtainable for the model polynomial

$$p(x) = \left(x - \frac{N}{2}\right)^D - \pi$$

such that $p(N/2)$ is correct to 6 figures (3.14159) for various values of N.

N	polyfit	numpy
10^1	10	7
10^2	16	4
10^3	10	3
10^4	7	2
10^5	6	2
10^6	5	1

Clearly `polyfit` outperforms `numpy` in this comparison.

To be fair, `numpy` does not use quad precision in `cho_factor()` or `cho_solve()` and using quad precision will make things better for it. However, comparisons with one such (closed source) routine show that `polyfit` is still the winner by a good margin.

B Appendix: Source Code for the Python Reference Implementation

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

"quad precision orthogonal polynomial least squares fitting"

## {{{ prologue
from __future__ import print_function as _

## pylint: disable=invalid-name,bad-whitespace
## pylint: disable=useless-object-inheritance
## pylint: disable=unnecessary-comprehension

import math

__all__ = [
    "PolyfitPlan", "PolyfitFit", "PolyfitEvaluator",
    "polyfit_plan", "polyfit_fit", "polyfit_eval",
    "polyfit_coefs", "polyfit_maxdeg", "polyfit_npoints",

    "zero", "one", "vappend", "vectorsum", "to_quad",
    "to_float", "add", "sub", "mul", "div", "sqrt",
]
## }}}
## {{{ 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"
    return sum2s(sumkcore(p, K))

def vectorsum(vec):
    "accurately sum a vector of floats, 19n-13 flops"
    return sumk(vec, K=3)
## }}}
## {{{ utility functions
def zero():
    "return quad precision 0"
    return (0., 0.)

def one():
    "return quad precision 1"
    return (1., 0.)

def vappend(vec, x):
```

```
    """
    append quad precision number to vector. this is used
    for vectorsum():

        v = [ ]
        for x in y:
            quad = ...
            vappend(v, quad)
        s = vectorsum(v)

    vectorsum() is more accurate than using add() in a loop.
    """
    vec.extend(x)

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

def to_float(x):
    "convert quad to float"
    return x[0] if isinstance(x, tuple) else float(x)
## }}}
## {{{ quad precision arithmetic
def add(x, y):
    "add two quads, 14 flops"
    x, xx = x
    y, yy = y
    z, zz = twosum(x, y)
    return twosum(z, zz + xx + yy)

def sub(x, y):
    "subtract 2 quads, 14 flops"
    x, xx = x
    y, yy = y
    z, zz = twodiff(x, y)
    return twosum(z, zz + xx - yy)

def mul(x, y):
    "multiply 2 quads, 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):
    "divide 2 quads, 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):
    "square root of a quad, 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)
## }}}
## {{{ polyfit_plan
def polyfit_plan(maxdeg, xv, wv):
    """
    given x values in xv[] and positive weights in wv[],
    make a plan to perform least squares fitting up to
    degree maxdeg.

    returns a plan object than can be json-serialized.
    """
    ## pylint: disable=too-many-locals

    ## convert to quad
    xv = [to_quad(x) for x in xv]
    wv = [to_quad(w) for w in wv]
    ## build workspaces and result object
    N = len(xv)
    b = [ ]             ## recurrence coefs b_k
    c = [ ]             ## recurrence coefs c_k
    g = [one()]         ## \gamma_k^2 \equiv (\phi_k, \phi_k)
    r = {
        "D": maxdeg,    ## max fit degree
        "N": N,         ## number of data points
        "b": b,         ## coefficients b_k
        "c": c,         ## coefficients c_k
        "g": g,         ## normalization constants g_k
        "x": xv,        ## x values, needed for polyfit_fit
        "w": wv         ## y values, needed for polyfit_fit
    }
    ## \phi_{k-1} and \phi_k
    phi_km1 = [zero()] * N ## \phi_{-1}
    phi_k   = [one()]  * N ## \phi_0

    for k in range(maxdeg + 1):
        bvec, gvec = [ ], [ ]
        for i in range(N):
            p   = phi_k[i]
            ## w_i \phi_k^2(x_i)
            wp2 = mul(wv[i], mul(p, p))
            ## w_i x_i \phi_k^2(x_i)
            vappend(bvec, mul(xv[i], wp2))
            ## w_i \phi_k^2(x_i)
            vappend(gvec, wp2)
        ## compute g_k = (\phi_k, \phi_k), b_k, and c_k
        gk = vectorsum(gvec)
        bk = div(vectorsum(bvec), gk)
        ck = div(gk, g[k])
        g.append(gk)
        b.append(bk)
```

```
        c.append(ck)
        ## if we aren't done, update pk[] and pkm1[]
        ## for the next round
        if k != maxdeg:
            for i in range(N):
                ## \phi_{k+1}(x_i) = (x_i - b_k) \phi_k(x_i) -
                ##                     c_k \phi_{k-1}(x_i)
                phi_kp1 = sub(
                    mul(sub(xv[i], bk), phi_k[i]),
                    mul(ck, phi_km1[i])
                )
                ## rotate the polys
                phi_km1[i] = phi_k[i]
                phi_k[i]   = phi_kp1
    c.append(zero())    ## needed in polyfit_eval
    return r
## }}}
## {{{ polyfit_fit
def polyfit_fit(plan, yv):
    """
    given a previously generated plan and a set of y values
    in yv[], compute all least squares fits to yv[] up to
    degree maxdeg.

    returns a json-serializable fit data object.
    """
    ## pylint: disable=too-many-locals
    N, D = plan["N"], plan["D"]
    b, c = plan["b"], plan["c"]
    g    = plan["g"]
    wv   = plan["w"]
    xv   = plan["x"]

    a, e = [ ], [ ]                 ## fit coefs and rms errors
    rv   = [to_quad(y) for y in yv] ## residuals

    ## \phi_{k-1} and \phi_k
    phi_km1 = [zero()] * N
    phi_k   = [one()]  * N
    for k in range(D + 1):
        ## compute ak as (residual, \phi_k) / (\phi_k, \phi_k)
        avec = [ ]
        for i in range(N):
            vappend(avec, mul(wv[i], mul(rv[i], phi_k[i])))
        ak = div(vectorsum(avec), g[k + 1])
        a.append(ak)

        ## remove the \phi_k component from the residual
        ## compute rms error for this degree
        evec = [ ]
        for i in range(N):
            rv[i] = r = sub(rv[i], mul(ak, phi_k[i]))
            vappend(evec, mul(r, r))
        e.append(sqrt(div(vectorsum(evec), to_quad(N))))
```

```
        ## if we aren't done, update pk[] and pkm1[]
        ## for the next round
        if k != D:
            for i in range(N):
                ## \phi_{k+1}(x_i) = (x_i - b_k) \phi_k(x_i) -
                ##                      c_k \phi_{k-1}(x_i)
                phi_kp1 = sub(
                    mul(sub(xv[i], b[k]), phi_k[i]),
                    mul(c[k], phi_km1[i])
                )
                ## rotate the polys
                phi_km1[i] = phi_k[i]
                phi_k[i]   = phi_kp1
    ## return fit data
    return {
        "a": a,     ## orthogonal poly coefs
        "e": e,     ## per-degree rms errors
        "r": rv     ## per-point residuals
    }
## }}}
## {{{ polyfit_eval
def polyfit_eval(   ## pylint: disable=too-many-arguments
        plan, fit, x, deg=-1, nder=0, scalar=True
    ):
    """
    given a plan, a fit data object returned by
    polyfit_fit, a point x, a least squares fit degree deg,
    and a desired number of derivatives to compute nder,
    calculate and return the value of the polynomial and
    any requested derivatives.

    if deg is negative, use maxdeg instead. if nder is
    negative, use the final value of deg; otherwise, compute
    ndeg derivatives of the least squares polynomial of
    degree deg.

    returns a list of quads whose first element is the value
    of the least squares polynomial of degree deg at x.
    subsequent elements are the requested derivatives. if zero
    derivatives are requested, the scalar function value
    is returned. if x is a quad, quads are returned.
    """
    ## pylint: disable=too-many-locals
    a, b, c, D = fit["a"], plan["b"], plan["c"], plan["D"]

    if deg < 0:
        deg = D
    if nder < 0:
        nder = deg

    ## z_k^{(j-1)} and z_k^{(j)} for clenshaw's recurrence
    zjm1 = a[:deg+1] + [zero(), zero()] ## init to a_k
    zj   = [zero()] * (deg + 3)

    fac   = one()            ## j! factor
    zeds  = [zero(), zero()]
```

```
    lim   = min(deg, nder)   ## max degree to compute
    x, x0 = to_quad(x), x
    ret   = [ ]              ## return value
    for j in range(lim + 1):
        if j > 1:
            fac = mul(fac, to_quad(j))
        ## compute z_j^{(j)} using the recurrence
        for k in range(deg, j - 1, -1):
            t = k - j
            ## z_k^{(j)} = z_k^{(j-1)} +
            ##              (x - b_t) z_{k+1}^{(j)} -
            ##              c_{t+1} z_{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 j! z_j^{(j)}
        ret.append(mul(fac, zj[j]))
        ## update z if we aren't done
        if j != lim:
            ## update zjm1
            zjm1[:] = zj
            ## zj only needs last 2 elements cleared
            zj[-2:] = zeds
    if nder > deg:
        ret += [zero()] * (nder - deg)
    ## returns quad precision (for polyfit_coefs)
    ret = ret if isinstance(x0, tuple) else \
          [to_float(r) for r in ret]
    return ret[0] if scalar and len(ret) == 1 else ret
## }}}
## {{{ polyfit_coefs
def polyfit_coefs(plan, fit, x0=0., deg=-1):
    """
    given a plan, a set of expansion coefficients generated
    by polyfit_fit, a center point x0, and a least squares
    fit degree, return the coefficients of powers of (x - x0)
    with the highest powers first. if deg is negative (the
    default), use maxdeg instead. the coefficients are quad
    precision.
    """
    ## get value and derivs, divide by j!
    vals = polyfit_eval(
        plan, fit, to_quad(x0), deg, deg,
        scalar=False
    )
    fac  = one()
    for j in range(2, len(vals)):
        fac     = div(fac, to_quad(j))
        vals[j] = mul(vals[j], fac)
    ## get highest power first
    vals.reverse()
    return vals if isinstance(x0, tuple) else \
        [to_float(v) for v in vals]
## }}}
```

```
## {{{ polyfit_maxdeg
def polyfit_maxdeg(plan):
    "return the maximum possible fit degree"
    return plan["D"]
## }}}
## {{{ polyfit_npoints
def polyfit_npoints(plan):
    "return the number of data points being fit"
    return plan["N"]
## }}}
## {{{ Polyfit classes
class PolyfitBase(object):
    "base class for polyfit classes"
    ## pylint: disable=too-few-public-methods

    data = None

    def close(self):
        "deallocate resources, a no-op for this impementation"

    def to_data(self):
        "return low level, serializable, class-specific data"
        return self.data

    @classmethod
    def from_data(cls, data):
        "return instance from serializable data"
        return cls(data=data)

class PolyfitEvaluator(PolyfitBase):
    """
    returned by PolyfitFit.evaluator(). this object evaluates
    the fit polynomial and its derivatives, and also returns
    its coefficients in powers of (x - x0) for given x0.
    """

    def __init__(self, data):
        self.data = data

    def __call__(self, x, deg=-1, nder=0):
        """
        given a point x, a least squares fit degree deg,
        and a desired number of derivatives to compute nder,
        calculate and return the value of the polynomial and
        any requested derivatives.

        if deg is negative, use maxdeg instead. if nder is
        negative, use the final value of deg; otherwise, compute
        nder derivatives of the least squares polynomial of
        degree deg.

        returns a list whose first element is the value of the
        least squares polynomial of degree deg at x. subsequent
        elements are the requested derivatives. if zero
        derivatives are requested, the scalar function value is
        returned.
```

```
        """
        plan, fit = self.data["plan"], self.data["fit"]
        return polyfit_eval(plan, fit, x, deg, nder)

    def coefs(self, x0, deg=-1):
        """
        return the coefficients of the fit polynomial of degree
        deg about (x - x0). if deg is negative, use maxdeg
        instead.
        """
        plan, fit = self.data["plan"], self.data["fit"]
        return polyfit_coefs(plan, fit, x0, deg)

class PolyfitFit(PolyfitBase):
    """
    orthogonal polynomial fitter returned by PolyfitPlan.fit()
    """

    def __init__(self, plan=None, yv=None, data=None):
        self.data = data if data else \
            { "plan": plan.copy(), "fit": polyfit_fit(plan, yv) }
        ## no need for these now that we have a fit, saves a lot
        ## of serialization space
        self.data["plan"].pop("x", None)
        self.data["plan"].pop("w", None)

    def evaluator(self):
        """
        return a PolyfitEvaluator for this fit.
        """
        return PolyfitEvaluator(self.data)

    def residuals(self):
        """
        return the list of residuals for the maxdeg fit.
        """
        return [to_float(e) for e in self.data["fit"]["r"]]

    def rms_errors(self):
        """
        return a list of rms errors, one per fit degree. use
        them to detect overfitting.
        """
        return [to_float(e) for e in self.data["fit"]["e"]]

class PolyfitPlan(PolyfitBase):
    """
    orthogonal polynomial least squares planning class. you must
    create one of these prior to fitting; it can be reused for
    multiple fits of the same xv[] and wv[].
    """

    def __init__(
            self, maxdeg=None, xv=None, wv=None, data=None
        ):
        """
```

```
        given x values in xv[] and positive weights in wv[],
        make a plan to perform least squares fitting up to
        degree maxdeg.

        this is code for "compute everything need to calculate
        an expansion in xv- and wv-specific orthogonal
        polynomials".
        """
        self.data = data if data else \
            { "plan": polyfit_plan(maxdeg, xv, wv) }

    def fit(self, yv):
        """
        given a set of y values in yv[], compute all least
        squares fits to yv[] up to degree maxdeg. returns
        a PolyfitFit object.
        """
        return PolyfitFit(self.data["plan"], yv)

    def maxdeg(self):
        "return the maximum fit degree"
        return polyfit_maxdeg(self.data["plan"])

    def npoints(self):
        "return the number of fit points"
        return polyfit_npoints(self.data["plan"])
## }}}

## EOF
```

C Appendix: Source Code for the numpy Fit Code

```
"numpy fit using quad-precision setup"

from __future__ import print_function as _

## pylint: disable=invalid-name,bad-whitespace

import math
import sys

import numpy as np
import scipy.linalg as la

sys.path.insert(0, "..")
from polyfit import (   ## pylint: disable=wrong-import-position
    sub, mul, vectorsum, vappend, to_quad, to_float
)

def npfit(xv, yv, wv, D):
    "numpy fit, quad-precision setup"
    ## pylint: disable=too-many-locals
    xv = [to_quad(x) for x in xv]
    yv = [to_quad(y) for y in yv]
    wv = [to_quad(w) for w in wv]
```

```
    xa = wv[:]          ## accumulator
    mx = [ ]            ## quad-prec moments
    r  = [ ]            ## quad-prec rhs in Ac=r
    for i in range((D + 1) * 2):
        if i <= D:
            ## compute rhs
            v = [ ]
            for x, y in zip(xa, yv):
                vappend(v, mul(x, y))
            r.append(vectorsum(v))
        ## compute moments up to 2D+1
        v = [ ]
        for x in xa:
            vappend(v, x)
        mx.append(vectorsum(v))
        for j, x in enumerate(xa):
            xa[j] = mul(x, xv[j])
    ## build the normal matrix from qprec moments
    A = [ ]
    for i in range(D + 1):
        A.append([to_float(m) for m in mx[i:i+D+1]])
    A = np.array(A)
    ## build the numpy rhs from the qprec one
    b = np.array([to_float(x) for x in r])
    ## solve the normal equations
    info = la.cho_factor(A)
    cofs = la.cho_solve(info, b)
    ## get 'em into std order for horner's method
    cofs = list(cofs)
    cofs.reverse()
    return cofs

## EOF
```

D Appendix: Source Code for Integration and Quadrature Routines

```
"""
integrals and gaussian quadrature add-ons for polyfit.py
"""

from __future__ import print_function as _

## pylint: disable=invalid-name,bad-whitespace

from polyfit import (
    zero, one, to_quad, to_float,
    add, sub, div, mul,
    vappend, vectorsum,
    PolyfitBase
)

__all__ = ["PolyplusIntegrator", "PolyplusQuadrature"]

```

```
## {{{ integration
class PolyplusIntegrator(PolyfitBase):
    """
    this class computes the definite integral of a fitted
    polynomial. be careful: we have to compute coefficients
    in powers of x to make this work, and those coefficients
    might be less accurate than desired.
    """

    def __init__(self, data, deg=-1):
        """
        create an integrator from a fit polynomial of given
        degree.
        """
        if isinstance(data, dict):
            self.data   = data
            self._coefs = data["coefs"]
        else:
            self.data   = data.to_data()["fit"].copy()
            self._coefs = coefs = data.evaluator().coefs(zero(), deg)
            self.data["coefs"] = coefs

            deg = len(coefs) - 1
            uno = one()
            for j in range(deg, -1, -1):
                i = deg - j
                fac = div(uno, to_quad(j + 1))
                coefs[i] = mul(coefs[i], fac)
            ## there is an implied-zero constant term

    def qcoefs(self):
        """
        return the coefficients for the integrated polynomial
        in quad precision.
        """
        return self._coefs + [zero()]

    def coefs(self):
        "same as qcoefs, but in double precision"
        return [to_float(c) for c in self.qcoefs()]

    def __call__(self, x):
        """
        return the quad-precision definite integral from 0 to x.
        the return value is quad if x is quad.
        """
        q   = to_quad(x)
        ret = zero()
        for c in self._coefs:
            ret = add(mul(ret, q), c)
        ## handle the implied zero constant term
        ret = mul(ret, q)
        return ret if isinstance(x, tuple) else to_float(ret)
## }}}
## {{{ quad precision root finding using bisection
def bis(    ## pylint: disable=too-many-arguments
```

```
        func, a, fa, b, fb,
        maxiter=108,    ## rel err >= 2**-107
    ):
    """
    quad precision root finding using bisection
    """
    ## this is over the top but works for its use
    ## in this module; don't use this for general
    ## root finding!
    assert to_float(mul(fa, fb)) < 0
    half = to_quad(0.5)
    for _ in range(maxiter):
        c = mul(half, add(a, b))
        if c in (a, b):
            break
        fc = func(c)
        if fc in (fa, fb):
            break
        if to_float(mul(fa, fc)) < 0:
            b, fb = c, fc
        elif fc == (0, 0):
            break
        else:
            a, fa = c, fc
    return c
## }}}
## {{{ quadrature over wv[] and xv[]
class PolyplusQuadrature(PolyfitBase):
    """
    this class implements gaussian quadrature on a
    discrete set of points.

    if you want to compute

        sum(f(x_i) * w_i for x_i, w_i in zip(xv, wv))

    you can replace this with gaussian quadrature as

        sum(f(z_i) * H_i for z_i, H_i in zip(Z, H))

    the difference is that D = len(Z) is much smaller
    than len(xv). Z and H are generated from a fit
    plan for xv and wv. this module provides a function
    to accurately sum f().

    the quadrature formula is exact for polynomial
    functions f() up to degree 2D-1.

    for non-polynomial functions, the error is
    proportional to the 2D-th derivative of f()
    divided by (2D)!
    """

    def __init__(self, data):
        "init self from a plan"
        if isinstance(data, dict):
```

```
            self.data = data
        else:
            self.data = data = data.to_data()["plan"].copy()

            data["x0"], data["x1"] = min(data["x"]), max(data["x"])

            data["roots"]   = { 0: [ ] }
            data["schemes"] = { }

        self.b, self.c, self.g = data["b"], data["c"], data["g"]
        self.x0, self.x1 = to_quad(data["x0"]), to_quad(data["x1"])
        self.the_roots = data["roots"]
        self.the_schemes = data["schemes"]

    def __call__(self, func, deg):
        r"""
        compute \sum_{i=1}^N w_i func(x_i) using the quadrature
        scheme \sum_{i=0}^{deg} H_i func(z_i). this is exact if
        func() is a poly of degree < 2D. func is assumed to take
        and return a float.
        """
        f = lambda x: func(to_float(x))
        return to_float(self.qquad(f, deg))

    def qquad(self, func, deg):
        r"""
        compute \sum_{i=1}^N w_i func(x_i) using the quadrature
        scheme \sum_{i=0}^{deg} H_i func(z_i). this is exact if
        func() is a poly of degree < 2D. func is assumed to take
        and return a quad.
        """
        v = [ ]
        for z, H in self.scheme(deg):
            vappend(v, mul(H, to_quad(func(z))))
        return vectorsum(v)

    def roots(self, k):
        "return the roots of the orthogonal poly of degree k"
        if k not in self.the_roots:
            self.the_roots[k] = self._roots(k)
        return self.the_roots[k]

    def scheme(self, k):
        """
        return the ordinates and christoffel numbers for
        the quadrature scheme of order k
        """
        if k not in self.the_schemes:
            self.the_schemes[k] = self._scheme(k)
        return self.the_schemes[k]

    def _phi_k(self, x, k):
        "internal: compute the k-th orthogonal poly at x"
        x    = to_quad(x)
        b    = self.b
        c    = self.c
```

```
        pjm1 = zero()
        pj   = one()
        for j in range(k):
            pjp1 = sub(
                mul(sub(x, b[j]), pj),
                mul(c[j], pjm1)
            )
            pjm1 = pj
            pj   = pjp1
        return pj

    def _roots(self, k):
        r"""
        internal:

        compute the roots of \phi_k using the separation
        property
        """
        ranges = [self.x0] + self.roots(k - 1) + [self.x1]
        ret    = [ ]
        func   = lambda x: self._phi_k(x, k)
        for i in range(len(ranges) - 1):
            a  = ranges[i]
            fa = func(a)
            b  = ranges[i + 1]
            fb = func(b)
            ret.append(bis(func, a, fa, b, fb))
        return ret

    def _scheme(self, k):
        """
        internal:

        compute the quadrature scheme of order k using the
        christoffel-darboux identity
        """
        b   = self.b
        c   = self.c
        g   = self.g
        uno = one()
        ret = [ ]
        for r in self.roots(k):
            v    = [ ]
            pjm1 = zero()
            pj   = one()
            ## loop over polys
            for j in range(k):
                ## add in the next term
                u = div(mul(pj, pj), g[j + 1])
                vappend(v, u)
                ## compute the next poly
                pjp1 = sub(
                    mul(sub(r, b[j]), pj),
                    mul(c[j], pjm1)
                )
                pjm1 = pj
```

```
                pj   = pjp1
            ## save the root and christoffel number
            ret.append(
                (r, div(uno, vectorsum(v)))
            )
        return ret
## }}}

## EOF
```

E 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 yields the values of $\hat{f}$ and its derivatives (optionally) at a given point x. The recurrence is given by

$$\begin{aligned} z_k &= a_k + (x - b_k) z_{k+1} - c_{k+1} z_{k+2} && (28) \\ c_{D+1} &= 0 && (29) \\ z_{D+1} &= 0 && (30) \\ z_{D+2} &= 0 && (31) \end{aligned}$$

and is applied in the downward direction. Solving (28) for a_k and substituting into (23) gives

$$\begin{aligned} \hat{f}(x) &= \sum_{k=0}^{D} \phi_k(x) \left[z_k - (x - b_k) z_{k+1} + c_{k+1} z_{k+2} \right] && (32) \\ &= \sum_{k=0}^{D} z_k \left[\phi_k(x) - (x - b_{k-1}) \phi_{k-1}(x) + c_{k-1} \phi_{k-2}(x) \right] && (33) \end{aligned}$$

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

$$\hat{f}(x) = z_0. \qquad (34)$$

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

$$\begin{aligned} z_k^{(j)} &= z_k^{(j-1)} + (x - b_{k-j}) z_{k+1}^{(j)} + c_{k-j+1} z_{k+2}^{(j)} && (35) \\ z_k^{(0)} &= a_k, && (36) \end{aligned}$$

then

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

To compute the j-th derivative of $\hat{f}(x)$, we invoke the recurrence (35) 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 (35) 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

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

since only the ϕ_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*.

F Appendix: The Zeros of $\phi_k(x)$

Earlier it was claimed that

1. The zeros of $\phi_k(x)$ are real
2. The zeros of $\phi_k(x)$ are simple
3. The zeros of $\phi_k(x)$ are lie in the interval spanned by the $\{x_i\}$
4. The zeros of $\phi_k(x)$ separate the zeros of $\phi_{k+1}(x)$

We will prove these facts in this appendix; the development follows Hildebrand [5].

Define

$$a = \min_i x_i \tag{37}$$
$$b = \max_i x_i \tag{38}$$

and consider the sum

$$\sum_{i=1}^{N} w_i \phi_0(x_i)\phi_k(x_i) = 0, \quad k > 0. \tag{39}$$

Since $k > 0$, this sum always equals zero; however, since $w_i\phi_0(x_i)$ does not change sign in $[a, b]$, it must be the case that $\phi_k(x)$ has at least one root in $[a, b]$. Now let the roots of $\phi_k(x)$ that lie in $[a, b]$ and have odd multiplicity be denoted $r_1, r_2, \ldots, r_q$. By definition the roots are r_i are distinct. Define $p(x)$ by

$$p(x) = (x - r_1)(x - r_2)\cdots(x - r_q). \tag{40}$$

Clearly q cannot exceed k since $\phi_k(x)$ is of degree k and therefore can only have k (possibly complex) roots. Since the roots of $p(x)$ are simple and distinct, the quantity $p(x)\phi_k(x)$ cannot change sign in $[a, b]$ and

$$\sum_{i=1}^{N} w_i p(x_i)\phi_k(x_i) > 0, \quad k > 0. \tag{41}$$

If we assume that $q < k$, we reach a contriction because this sum must be zero since the degree of $p(x)$ is less than k but $\phi_k(x)$ is orthogonal to all polynomials of degree less than k. Therefore we must have $q = k$ so that the roots of $\phi_k(x)$ are real, simple, and lie in $[a, b]$.

To prove the zero-separation property, define γ_k by

$$\gamma_k^2 = (\phi_k, \phi_k) > 0 \tag{42}$$

since $w_i > 0$ and rewrite (23) as

$$x\frac{\phi_k(x)}{\gamma_k^2} = \frac{\phi_{k+1}}{\gamma_k^2} + \frac{\phi_{k-1}}{\gamma_{k-1}^2} + b_k\frac{\phi_k(x)}{\gamma_k^2} \tag{43}$$

Multiplying this by $\phi_k(y)$, exchanging x and y, and subtracting the two eliminates the b_k term to give

$$(x-y)\frac{\phi_k(x)\phi_k(y)}{\gamma_k^2} = \frac{\phi_{k+1}(x)\phi_k(y) - \phi_k(x)\phi_{k+1}(y)}{\gamma_k^2} \tag{44}$$
$$- \frac{\phi_k(x)\phi_{k-1}(y) - \phi_{k-1}(x)\phi_k(y)}{\gamma_{k-1}^2} \tag{45}$$

Summing this from $k = 0 \dots D$ telescopes to yield the *Christoffel-Darboux* identity

$$\sum_{k=0}^{m} \frac{\phi_k(x)\phi_k(y)}{\gamma_k^2} = \frac{\phi_{m+1}(x)\phi_m(y) - \phi_m(x)\phi_{m+1}(y)}{\gamma_m^2(x-y)}. \tag{46}$$

The confluent form (which will be important below) is obtained by letting $y \to x$:

$$\sum_{k=0}^{m} \frac{[\phi_k(x)]^2}{\gamma_k^2} = \frac{\phi'_{m+1}(x)\phi_m(x) - \phi'_m(x)\phi_{m+1}(x)}{\gamma_m^2}. \tag{47}$$

The Christoffel-Darboux identity may be used to prove the fact that the roots of ϕ_k separate the roots of ϕ_{k+1}. To see this, suppose that x_i and x_{i+1} are consecutive roots of ϕ_{m+1}. Substituting these roots into (47), we see that the second term on the right hand side vanishes. The left hand side is strictly positive since $\phi_0 = 1$; therefore, $\phi'_{m+1}(x_i)\phi_m(x_i)$ and $\phi'_{m+1}(x_{i+1})\phi_m(x_{i+1})$ are positive. Since the zeros of all the ϕ_k are simple, it must be the case that $\phi'_{m+1}(x_i)$ and $\phi'_{m+1}(x_{i+1})$ have opposite sign. This means that $\phi_m(x_i)$ and $\phi_{m+1}(x_{i+1})$ have opposite sign; therefore, ϕ_m must have a root between x_i and x_{i+1} as was to be shown.

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] GH Golub and CF van Loan, Matrix Computations 2nd ed, Baltimore: Johns Hopkins University Press, 1989

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

[6] 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`

[7] LF Shampine, SM Davenport, and RE Huddleston, `https://netlib.org/slatec/src/dpolft.f` to compute a linear least-squares orthogonal polynomial fit. Written 6/1/1974, updated 5/27/1992.

[8] 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. Written 6/1/1974, updated 5/1/1992.

[9] 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. Written 6/1/1974, updated 5/1/1992.

[10] 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`