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

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

July 2, 2022

Abstract

In this note I present the Python2, Python3, and `C` `polyfit` package that implements quad-precision mantissa [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 source listing for the 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
```

```
49      ##     note that y is nonzero for this example
50      #wv = [y ** -2. for y in yv]
51
52      ## perform the fit
53      D    = len(cv) - 1
54      plan = PolyfitPlan(D, xv, wv)
55      fit  = plan.fit(yv)
56      ev   = fit.evaluator()
57
58      ## print the fit stats
59      deg = plan.maxdeg()
60      print("maxdeg", deg)
61      print("points", plan.npoints())
62
63      ## print per-degree rms errors
64      print("erms  ", flist(fit.rms_errors()))
65
66      ## print a few values
67      for i in range(4):
68          print("value  %.1f %s" % (xv[i], flist(ev(xv[i], nder=-1))))
69
70      ## print value and all derivatives for all degrees
71      for i in range(D + 1):
72          print("deg    %d %s" % (i, flist(ev(xv[0], deg=i, nder=-1))))
73
74      ## print coefficients for all degrees about (x - xv[0])
75      for i in range(D + 1):
76          print("coefs  %d %s" % (i, flist(ev.coefs(xv[0], i))))
77
78      ## coefs halfway through
79      print("coefs ", flist(ev.coefs(xv[N >> 1], deg)))
80
81  if __name__ == "__main__":
82      demo()
83
84  ## EOF
```

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

```
1   /*****************************************************************
2    * ex2.c - polyfit demo
3    */
4
5   #include <math.h>
6   #include <stdio.h>
7   #include <sys/time.h>
8
9   #include "polyfit.h"
10
11  #ifndef M_PI
12  #define M_PI 0
13  #define USE_ACOS
14  #endif
15
16  #define N 10000
```

```
17  #define D 3
18  double xv[N], yv[N], wv[N];
19
20  /* poly coefficients to fit, highest degree first */
21  double cv[D + 1] = { 2, 1, -1, M_PI };
22
23  void init() {
24      double y;
25      int    i, j;
26
27  #ifdef USE_ACOS
28      cv[D] = acos(-1);
29  #endif
30      for (i = 0; i < N; i++) {
31
32          /* evaluate the poly to fit using horner's method */
33          for (y = 0, j = 0; j <= D; j++) {
34              y *= i;
35              y += cv[j];
36          }
37          /* define xv[], yv[], and wv[] for the fit */
38          xv[i] = i;
39          yv[i] = y;
40  #if 1
41          wv[i] = 1;
42  #else
43          /* minimize relative residual */
44          wv[i] = 1. / (y * y); /* y != 0 for this example poly */
45  #endif
46      }
47  }
48
49  int main(int argc, char *argv[]) {
50      void  *plan, *fit, *ev;
51      int    i, j, n;
52      double coefs[D + 1], d[D + 1];
53      const double *t;
54
55      /* fill in xv, yv, and, wv */
56      init();
57
58      /* create the fit plan */
59      if ((plan = polyfit_plan(D, xv, wv, N)) == NULL) {
60          perror("polyfit_plan");
61          return 1;
62      }
63
64      /* compute the fit */
65      if ((fit = polyfit_fit(plan, yv)) == NULL) {
66          perror("polyfit_fit");
67          return 1;
68      }
69
70      /* make an evaluator */
71      if ((ev = polyfit_evaluator(fit)) == NULL) {
72          perror("polyfit_evaluator");
```

```
 73          return 1;
 74      }
 75
 76      /* print fit stats */
 77      if ((n = polyfit_maxdeg(plan)) < 0) {
 78          perror("polyfit_maxdeg");
 79          return 1;
 80      }
 81      printf("maxdeg %d\n", n);
 82      if ((n = polyfit_npoints(plan)) < 0) {
 83          perror("polyfit_npoints");
 84          return 1;
 85      }
 86      printf("points %d\n", n);
 87
 88      /* print per-degree rms errors */
 89      if ((t = polyfit_rms_errs(fit, NULL)) == NULL) {
 90          perror("polyfit_rms_errs");
 91          return 1;
 92      }
 93      printf("erms  ");
 94      for (i = 0; i <= D; i++) {
 95          printf(" %.15e", t[i]);
 96      }
 97      printf("\n");
 98
 99      /* print a few values */
100      for (i = 0; i < 4; i++) {
101          if (polyfit_eval(ev, xv[i], D, d, D) < 0) {
102              perror("polyfit_eval");
103              return 1;
104          }
105          printf("value  %.1f", xv[i]);
106          for (j = 0; j <= D; j++) {
107              printf(" %.15e", d[j]);
108          }
109          printf("\n");
110      }
111
112      /* print value and all derivatives for all degrees */
113      for (i = 0; i <= D; i++) {
114          if (polyfit_eval(ev, xv[0], i, d, -1) < 0) {
115              perror("polyfit_eval");
116              return 1;
117          }
118          printf("deg    %d", i);
119          for (j = 0; j <= i; j++) {
120              printf(" %.15e", d[j]);
121          }
122          printf("\n");
123      }
124
125      /* print coefficients for all degrees about (x - xv[0]) */
126      for (i = 0; i <= D; i++) {
127          if (polyfit_coefs(ev, xv[0], i, coefs) < 0) {
128              perror("polyfit_coefs");
```

```
129             return 1;
130         }
131         printf("coefs  %d", i);
132         for (j = 0; j <= i; j++) {
133             printf(" %.15e", coefs[j]);
134         }
135         printf("\n");
136     }
137
138     /* coefs halfway through */
139     if (polyfit_coefs(ev, xv[N>>1], D, coefs) < 0) {
140         perror("polyfit_coefs");
141         return 1;
142     }
143     printf("coefs ");
144     for (i = 0; i <= D; i++) {
145         printf(" %.15e", coefs[i]);
146     }
147     printf("\n");
148
149     /* free the fit objects */
150     polyfit_free(ev);
151     polyfit_free(fit);
152     polyfit_free(plan);
153     return 0;
154 }
155
156 /* EOF */
```

2 API Documentation

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

```
NAME
    polyfit - quad-precision mantissa 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.

```
 1  /*************************************************************************
 2   * polyfit.h - quad-precision mantissa orthogonal polynomial least squares
 3   */
 4
 5  #ifndef polyfit_h__
 6  #define polyfit_h__
 7
 8  #ifdef __cplusplus
```

```
 9  extern "C" {
10  #endif
11
12  /*************************************************************
13   * given x values in xv[] and positive weights in wv[],
14   * make a plan to perform least squares fitting up to
15   * degree maxdeg and return a plan object. returns NULL
16   * and sets errno on error.
17   */
18  extern void *polyfit_plan(
19      const int maxdeg,
20      const double * const xv,
21      const double * const wv,
22      const int npoints
23  );
24
25  /*************************************************************
26   * given a set of y values in yv[], compute all least
27   * squares fits to yv[] up to degree maxdeg and return
28   * a fit object. returns NULL and sets errno on error.
29   */
30  extern void *polyfit_fit(
31      const void * const plan,
32      const double * const yv
33  );
34
35  /*************************************************************
36   * given a fit, return an evaluator that can (a) compute the
37   * fit polynomial and its derivatives and (b) can compute
38   * coefficients of the polynomial about a given point x0.
39   * returns NULL and sets errno on error.
40   */
41  extern void *polyfit_evaluator(
42      const void * const fit
43  );
44
45  /*************************************************************
46   * given a point x, a least squares fit degree degree,
47   * and a desired number of derivatives to compute nderiv,
48   * calculate and return the value of the polynomial and
49   * any requested derivatives.
50   *
51   * if degree is negative, use maxdeg instead. if nderiv is
52   * negative, use the final value of deg; otherwise, compute
53   * nderiv derivatives of the least squares polynomial of
54   * degree deg.
55   *
56   * the derivatives array contains the polynomial value first,
57   * followed by any requested derivatives.
58   *
59   * returns 0 on success. on failure returns -1 and sets errno:
60   *   EINVAL - evaluator is not an evaluator.
61   *          - derivatives is NULL
62   */
63  extern int polyfit_eval(
64      void * const evaluator,
```

```
65      const double x,
66      const int degree,
67      double * const derivatives,
68      const int nderiv
69  );
70
71  /*************************************************************
72   * return the coefficients of the fit polynomial of degree
73   * degree about (x - x0). if degree is negative, use maxdeg
74   * instead.
75   *
76   * returns 0 on success. on failure returns -1 and sets errno:
77   *   EINVAL - evaluator is not an evaluator.
78   *          - coefs is NULL
79   */
80  extern int polyfit_coefs(
81      void * const evaluator,
82      const double x0,
83      const int degree,
84      double * const coefs
85  );
86
87  /*************************************************************
88   * return the maximum fit degree
89   *
90   * on failure returns -1 and sets errno:
91   *   EINVAL - plan object is not recognized
92   */
93  extern int polyfit_maxdeg(
94      const void * const plan
95  );
96
97  /*************************************************************
98   * return the number of fit points
99   *
100  * on failure returns -1 and sets errno:
101  *   EINVAL - plan object is not recognized
102  */
103 extern int polyfit_npoints(
104     const void * const plan
105 );
106
107 /*************************************************************
108  * return the list of residuals for the maxdeg fit.
109  *
110  * returns 0 on success. on failure returns -1 and sets errno:
111  *   EINVAL - fit is not a fit.
112  */
113 extern double *polyfit_resids(
114     const void * const fit,
115     double * const resids
116 );
117
118 /*************************************************************
119  * return a list of rms errors, one per fit degree. use them
120  * to detect overfitting.
```

```
121   *
122   * returns 0 on success. on failure returns -1 and sets errno:
123   *   EINVAL - fit is not a fit.
124   */
125  extern double *polyfit_rms_errs(
126      const void * const fit,
127      double * const errs
128  );
129
130  /*************************************************************
131   * free a plan, fit, or evaluator object
132   *
133   * returns 0 on success. on failure returns -1 and sets errno:
134   *   EINVAL - unrecognized object type.
135   *          - polyfit_object is NULL.
136   */
137  extern int polyfit_free(
138      void * const polyfit_object
139  );
140
141  #ifdef __cplusplus
142  };
143  #endif
144
145  #endif
146
147  /** 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, $0 \leq j \leq D$

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

so that

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

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:

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

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

$$(y, \phi_k) = a_k (\phi_k, \phi_k) \tag{15}$$

$$\tag{16}$$

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 with quadruple-precision mantissa (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 mantissa 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
```

```
37
38  def split(a, FACTOR = 1. + 2. ** 27):
39      "4 flops, algorithm 3.2 from ogita"
40      c = FACTOR * a
41      x = c - (c - a)
42      y = a - x
43      return x, y
44
45  def twoproduct(a, b):
46      "23 flops, algorithm 3.3 from ogita"
47      x      = a * b
48      a1, a2 = split(a)
49      b1, b2 = split(b)
50      y      = a2 * b2 - (x - a1 * b1 - a2 * b1 - a1 * b2)
51      return twosum(x, y)
52
53  def sum2s(p):
54      "7n-1 flops, algorithm 4.1 from ogita"
55      pi, sigma = p[0], 0.
56      for i in range(1, len(p)):
57          pi, q  = twosum(pi, p[i])
58          sigma += q
59      return twosum(pi, sigma)
60
61  def vsum(p):
62      "6(n-1) flops, algorithm 4.3 from ogita"
63      im1 = 0
64      for i in range(1, len(p)):
65          p[i], p[im1] = twosum(p[i], p[im1])
66          im1 = i
67      return p
68
69  def sumkcore(p, K):
70      "6(K-1)(n-1) flops, algorithm 4.8 from ogita"
71      for _ in range(K - 1):
72          p = vsum(p)
73      return p
74
75  def sumk(p, K):
76      "(6K+1)(n-1)+6 flops, algorithm 4.8 from ogita"
77      return sum2s(sumkcore(p, K))
78
79  def vectorsum(vec):
80      "accurately sum a vector of floats, 19n-13 flops"
81      return sumk(vec, K=3)
82  ## }}}
83  ## {{{ utility functions
84  def zero():
85      "return quad precision 0"
86      return (0., 0.)
87
88  def one():
89      "return quad precision 1"
90      return (1., 0.)
91
92  def vappend(vec, x):
```

```
 93      """
 94      append quad precision number to vector. this is used
 95      for vectorsum():
 96
 97          v = [ ]
 98          for x in y:
 99              quad = ...
100              vappend(v, quad)
101          s = vectorsum(v)
102
103      vectorsum() is more accurate than using add() in a loop.
104      """
105      vec.extend(x)
106
107  def to_quad(x):
108      "convert float or quad to quad"
109      return x if isinstance(x, tuple) else (float(x), 0.)
110
111  def to_float(x):
112      "convert quad to float"
113      return x[0] if isinstance(x, tuple) else float(x)
114  ## }}}
115  ## {{{ quad precision arithmetic
116  def add(x, y):
117      "add two quads, 14 flops"
118      x, xx = x
119      y, yy = y
120      z, zz = twosum(x, y)
121      return twosum(z, zz + xx + yy)
122
123  def sub(x, y):
124      "subtract 2 quads, 14 flops"
125      x, xx = x
126      y, yy = y
127      z, zz = twodiff(x, y)
128      return twosum(z, zz + xx - yy)
129
130  def mul(x, y):
131      "multiply 2 quads, 33 flops"
132      x, xx = x
133      y, yy = y
134      z, zz = twoproduct(x, y)
135      zz   += xx * y + x * yy
136      return twosum(z, zz)
137
138  def div(x, y):
139      "divide 2 quads, 36 flops, from dekker"
140      x, xx = x
141      y, yy = y
142      c     = x / y
143      u, uu = twoproduct(c, y)
144      cc    = (x - u - uu + xx - c * yy) / y
145      return twosum(c, cc)
146
147  def sqrt(x):
148      "square root of a quad, 35 flops, from dekker"
```

```
149      x, xx = x
150      if not (x or xx):
151          return zero()
152      c     = math.sqrt(x)
153      u, uu = twoproduct(c, c)
154      cc    = (x - u - uu + xx) * 0.5 / c
155      return twosum(c, cc)
156  ## }}}
157  ## {{{ polyfit_plan
158  def polyfit_plan(maxdeg, xv, wv):
159      """
160      given x values in xv[] and positive weights in wv[],
161      make a plan to perform least squares fitting up to
162      degree maxdeg.
163
164      returns a plan object than can be json-serialized.
165      """
166      ## pylint: disable=too-many-locals
167
168      ## convert to quad
169      xv = [to_quad(x) for x in xv]
170      wv = [to_quad(w) for w in wv]
171      ## build workspaces and result object
172      N = len(xv)
173      b = [ ]             ## recurrence coefs b_k
174      c = [ ]             ## recurrence coefs c_k
175      g = [one()]         ## \gamma_k^2 \equiv (\phi_k, \phi_k)
176      r = {
177          "D": maxdeg,    ## max fit degree
178          "N": N,         ## number of data points
179          "b": b,         ## coefficients b_k
180          "c": c,         ## coefficients c_k
181          "g": g,         ## normalization constants g_k
182          "x": xv,        ## x values, needed for polyfit_fit
183          "w": wv         ## y values, needed for polyfit_fit
184      }
185      ## \phi_{k-1} and \phi_k
186      phi_km1 = [zero()] * N ## \phi_{-1}
187      phi_k   = [one()]  * N ## \phi_0
188
189      for k in range(maxdeg + 1):
190          bvec, gvec = [ ], [ ]
191          for i in range(N):
192              p   = phi_k[i]
193              ## w_i \phi_k^2(x_i)
194              wp2 = mul(wv[i], mul(p, p))
195              ## w_i x_i \phi_k^2(x_i)
196              vappend(bvec, mul(xv[i], wp2))
197              ## w_i \phi_k^2(x_i)
198              vappend(gvec, wp2)
199          ## compute g_k = (\phi_k, \phi_k), b_k, and c_k
200          gk = vectorsum(gvec)
201          bk = div(vectorsum(bvec), gk)
202          ck = div(gk, g[k])
203          g.append(gk)
204          b.append(bk)
```

```
205         c.append(ck)
206         ## if we aren't done, update pk[] and pkm1[]
207         ## for the next round
208         if k != maxdeg:
209             for i in range(N):
210                 ## \phi_{k+1}(x_i) = (x_i - b_k) \phi_k(x_i) -
211                 ##                     c_k \phi_{k-1}(x_i)
212                 phi_kp1 = sub(
213                     mul(sub(xv[i], bk), phi_k[i]),
214                     mul(ck, phi_km1[i])
215                 )
216                 ## rotate the polys
217                 phi_km1[i] = phi_k[i]
218                 phi_k[i]   = phi_kp1
219     c.append(zero())    ## needed in polyfit_eval
220     return r
221 ## }}}
222 ## {{{ polyfit_fit
223 def polyfit_fit(plan, yv):
224     """
225     given a previously generated plan and a set of y values
226     in yv[], compute all least squares fits to yv[] up to
227     degree maxdeg.
228 
229     returns a json-serializable fit data object.
230     """
231     ## pylint: disable=too-many-locals
232     N, D = plan["N"], plan["D"]
233     b, c = plan["b"], plan["c"]
234     g    = plan["g"]
235     wv   = plan["w"]
236     xv   = plan["x"]
237 
238     a, e = [ ], [ ]                 ## fit coefs and rms errors
239     rv   = [to_quad(y) for y in yv] ## residuals
240 
241     ## \phi_{k-1} and \phi_k
242     phi_km1 = [zero()] * N
243     phi_k   = [one()]  * N
244     for k in range(D + 1):
245         ## compute ak as (residual, \phi_k) / (\phi_k, \phi_k)
246         avec = [ ]
247         for i in range(N):
248             vappend(avec, mul(wv[i], mul(rv[i], phi_k[i])))
249         ak = div(vectorsum(avec), g[k + 1])
250         a.append(ak)
251 
252         ## remove the \phi_k component from the residual
253         ## compute rms error for this degree
254         evec = [ ]
255         for i in range(N):
256             rv[i] = r = sub(rv[i], mul(ak, phi_k[i]))
257             vappend(evec, mul(r, r))
258         e.append(sqrt(div(vectorsum(evec), to_quad(N))))
259 
260 
```

```
261         ## if we aren't done, update pk[] and pkm1[]
262         ## for the next round
263         if k != D:
264             for i in range(N):
265                 ## \phi_{k+1}(x_i) = (x_i - b_k) \phi_k(x_i) -
266                 ##                     c_k \phi_{k-1}(x_i)
267                 phi_kp1 = sub(
268                     mul(sub(xv[i], b[k]), phi_k[i]),
269                     mul(c[k], phi_km1[i])
270                 )
271                 ## rotate the polys
272                 phi_km1[i] = phi_k[i]
273                 phi_k[i]   = phi_kp1
274     ## return fit data
275     return {
276         "a": a,     ## orthogonal poly coefs
277         "e": e,     ## per-degree rms errors
278         "r": rv     ## per-point residuals
279     }
280 ## }}}
281 ## {{{ polyfit_eval
282 def polyfit_eval(   ## pylint: disable=too-many-arguments
283         plan, fit, x, deg=-1, nder=0, scalar=True
284     ):
285     """
286     given a plan, a fit data object returned by
287     polyfit_fit, a point x, a least squares fit degree deg,
288     and a desired number of derivatives to compute nder,
289     calculate and return the value of the polynomial and
290     any requested derivatives.
291
292     if deg is negative, use maxdeg instead. if nder is
293     negative, use the final value of deg; otherwise, compute
294     ndeg derivatives of the least squares polynomial of
295     degree deg.
296
297     returns a list of quads whose first element is the value
298     of the least squares polynomial of degree deg at x.
299     subsequent elements are the requested derivatives. if zero
300     derivatives are requested, the scalar function value
301     is returned. if x is a quad, quads are returned.
302     """
303     ## pylint: disable=too-many-locals
304     a, b, c, D = fit["a"], plan["b"], plan["c"], plan["D"]
305
306     if deg < 0:
307         deg = D
308     if nder < 0:
309         nder = deg
310
311     ## z_k^{(j-1)} and z_k^{(j)} for clenshaw's recurrence
312     zjm1 = a[:deg+1] + [zero(), zero()] ## init to a_k
313     zj   = [zero()] * (deg + 3)
314
315     fac   = one()           ## j! factor
316     zeds  = [zero(), zero()]
```

```
317      lim   = min(deg, nder)   ## max degree to compute
318      x, x0 = to_quad(x), x
319      ret   = [ ]              ## return value
320      for j in range(lim + 1):
321          if j > 1:
322              fac = mul(fac, to_quad(j))
323          ## compute z_j^{(j)} using the recurrence
324          for k in range(deg, j - 1, -1):
325              t = k - j
326              ## z_k^{(j)} = z_k^{(j-1)} +
327              ##              (x - b_t) z_{k+1}^{(j)} -
328              ##              c_{t+1} z_{k+2}^{(j)}
329              tmp = sub(
330                  mul(sub(x, b[t]), zj[k + 1]),
331                  mul(c[t + 1], zj[k + 2])
332              )
333              zj[k] = add(zjm1[k], tmp)
334          ## save j! z_j^{(j)}
335          ret.append(mul(fac, zj[j]))
336          ## update z if we aren't done
337          if j != lim:
338              ## update zjm1
339              zjm1[:] = zj
340              ## zj only needs last 2 elements cleared
341              zj[-2:] = zeds
342      if nder > deg:
343          ret += [zero()] * (nder - deg)
344      ## returns quad precision (for polyfit_coefs)
345      ret = ret if isinstance(x0, tuple) else \
346            [to_float(r) for r in ret]
347      return ret[0] if scalar and len(ret) == 1 else ret
348  ## }}}
349  ## {{{ polyfit_coefs
350  def polyfit_coefs(plan, fit, x0=0., deg=-1):
351      """
352      given a plan, a set of expansion coefficients generated
353      by polyfit_fit, a center point x0, and a least squares
354      fit degree, return the coefficients of powers of (x - x0)
355      with the highest powers first. if deg is negative (the
356      default), use maxdeg instead. the coefficients are quad
357      precision.
358      """
359      ## get value and derivs, divide by j!
360      vals = polyfit_eval(
361          plan, fit, to_quad(x0), deg, deg,
362          scalar=False
363      )
364      fac  = one()
365      for j in range(2, len(vals)):
366          fac     = div(fac, to_quad(j))
367          vals[j] = mul(vals[j], fac)
368      ## get highest power first
369      vals.reverse()
370      return vals if isinstance(x0, tuple) else \
371          [to_float(v) for v in vals]
372  ## }}}
```

```
373 ## {{{ polyfit_maxdeg
374 def polyfit_maxdeg(plan):
375     "return the maximum possible fit degree"
376     return plan["D"]
377 ## }}}
378 ## {{{ polyfit_npoints
379 def polyfit_npoints(plan):
380     "return the number of data points being fit"
381     return plan["N"]
382 ## }}}
383 ## {{{ Polyfit classes
384 class PolyfitBase(object):
385     "base class for polyfit classes"
386     ## pylint: disable=too-few-public-methods
387 
388     data = None
389 
390     def close(self):
391         "deallocate resources, a no-op for this impementation"
392 
393     def to_data(self):
394         "return low level, serializable, class-specific data"
395         return self.data
396 
397     @classmethod
398     def from_data(cls, data):
399         "return instance from serializable data"
400         return cls(data=data)
401 
402 class PolyfitEvaluator(PolyfitBase):
403     """
404     returned by PolyfitFit.evaluator(). this object evaluates
405     the fit polynomial and its derivatives, and also returns
406     its coefficients in powers of (x - x0) for given x0.
407     """
408 
409     def __init__(self, data):
410         self.data = data
411 
412     def __call__(self, x, deg=-1, nder=0):
413         """
414         given a point x, a least squares fit degree deg,
415         and a desired number of derivatives to compute nder,
416         calculate and return the value of the polynomial and
417         any requested derivatives.
418 
419         if deg is negative, use maxdeg instead. if nder is
420         negative, use the final value of deg; otherwise, compute
421         nder derivatives of the least squares polynomial of
422         degree deg.
423 
424         returns a list whose first element is the value of the
425         least squares polynomial of degree deg at x. subsequent
426         elements are the requested derivatives. if zero
427         derivatives are requested, the scalar function value is
428         returned.
```

```
429         """
430         plan, fit = self.data["plan"], self.data["fit"]
431         return polyfit_eval(plan, fit, x, deg, nder)
432 
433     def coefs(self, x0, deg=-1):
434         """
435         return the coefficients of the fit polynomial of degree
436         deg about (x - x0). if deg is negative, use maxdeg
437         instead.
438         """
439         plan, fit = self.data["plan"], self.data["fit"]
440         return polyfit_coefs(plan, fit, x0, deg)
441 
442 class PolyfitFit(PolyfitBase):
443     """
444     orthogonal polynomial fitter returned by PolyfitPlan.fit()
445     """
446 
447     def __init__(self, plan=None, yv=None, data=None):
448         self.data = data if data else \
449             { "plan": plan.copy(), "fit": polyfit_fit(plan, yv) }
450         ## no need for these now that we have a fit, saves a lot
451         ## of serialization space
452         self.data["plan"].pop("x", None)
453         self.data["plan"].pop("w", None)
454 
455     def evaluator(self):
456         """
457         return a PolyfitEvaluator for this fit.
458         """
459         return PolyfitEvaluator(self.data)
460 
461     def residuals(self):
462         """
463         return the list of residuals for the maxdeg fit.
464         """
465         return [to_float(e) for e in self.data["fit"]["r"]]
466 
467     def rms_errors(self):
468         """
469         return a list of rms errors, one per fit degree. use
470         them to detect overfitting.
471         """
472         return [to_float(e) for e in self.data["fit"]["e"]]
473 
474 class PolyfitPlan(PolyfitBase):
475     """
476     orthogonal polynomial least squares planning class. you must
477     create one of these prior to fitting; it can be reused for
478     multiple fits of the same xv[] and wv[].
479     """
480 
481     def __init__(
482             self, maxdeg=None, xv=None, wv=None, data=None
483         ):
484         """
```

```
485         given x values in xv[] and positive weights in wv[],
486         make a plan to perform least squares fitting up to
487         degree maxdeg.
488
489         this is code for "compute everything need to calculate
490         an expansion in xv- and wv-specific orthogonal
491         polynomials".
492         """
493         self.data = data if data else \
494             { "plan": polyfit_plan(maxdeg, xv, wv) }
495
496     def fit(self, yv):
497         """
498         given a set of y values in yv[], compute all least
499         squares fits to yv[] up to degree maxdeg. returns
500         a PolyfitFit object.
501         """
502         return PolyfitFit(self.data["plan"], yv)
503
504     def maxdeg(self):
505         "return the maximum fit degree"
506         return polyfit_maxdeg(self.data["plan"])
507
508     def npoints(self):
509         "return the number of fit points"
510         return polyfit_npoints(self.data["plan"])
511 ## }}}
512
513 ## EOF
```

C Appendix: Source Code for the numpy Fit Code

```
 1 "numpy fit using quad-precision setup"
 2
 3 from __future__ import print_function as _
 4
 5 ## pylint: disable=invalid-name,bad-whitespace
 6
 7 import math
 8 import sys
 9
10 import numpy as np
11 import scipy.linalg as la
12
13 sys.path.insert(0, "..")
14 from polyfit import (   ## pylint: disable=wrong-import-position
15     sub, mul, vectorsum, vappend, to_quad, to_float
16 )
17
18 def npfit(xv, yv, wv, D):
19     "numpy fit, quad-precision setup"
20     ## pylint: disable=too-many-locals
21     xv = [to_quad(x) for x in xv]
22     yv = [to_quad(y) for y in yv]
23     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"]

```

```
18  ## {{{ integration
19  class PolyplusIntegrator(PolyfitBase):
20      """
21      this class computes the definite integral of a fitted
22      polynomial. be careful: we have to compute coefficients
23      in powers of x to make this work, and those coefficients
24      might be less accurate than desired.
25      """
26
27      def __init__(self, data, deg=-1):
28          """
29          create an integrator from a fit polynomial of given
30          degree.
31          """
32          if isinstance(data, dict):
33              self.data   = data
34              self._coefs = data["coefs"]
35          else:
36              self.data   = data.to_data()["fit"].copy()
37              self._coefs = coefs = data.evaluator().coefs(zero(), deg)
38              self.data["coefs"] = coefs
39
40              deg = len(coefs) - 1
41              uno = one()
42              for j in range(deg, -1, -1):
43                  i = deg - j
44                  fac = div(uno, to_quad(j + 1))
45                  coefs[i] = mul(coefs[i], fac)
46              ## there is an implied-zero constant term
47
48      def qcoefs(self):
49          """
50          return the coefficients for the integrated polynomial
51          in quad precision.
52          """
53          return self._coefs + [zero()]
54
55      def coefs(self):
56          "same as qcoefs, but in double precision"
57          return [to_float(c) for c in self.qcoefs()]
58
59      def __call__(self, x):
60          """
61          return the quad-precision definite integral from 0 to x.
62          the return value is quad if x is quad.
63          """
64          q   = to_quad(x)
65          ret = zero()
66          for c in self._coefs:
67              ret = add(mul(ret, q), c)
68          ## handle the implied zero constant term
69          ret = mul(ret, q)
70          return ret if isinstance(x, tuple) else to_float(ret)
71  ## }}}
72  ## {{{ quad precision root finding using bisection
73  def bis(    ## pylint: disable=too-many-arguments
```

```
74          func, a, fa, b, fb,
75          maxiter=108,    ## rel err >= 2**-107
76      ):
77      """
78      quad precision root finding using bisection
79      """
80      ## this is over the top but works for its use
81      ## in this module; don't use this for general
82      ## root finding!
83      assert to_float(mul(fa, fb)) < 0
84      half = to_quad(0.5)
85      for _ in range(maxiter):
86          c = mul(half, add(a, b))
87          if c in (a, b):
88              break
89          fc = func(c)
90          if fc in (fa, fb):
91              break
92          if to_float(mul(fa, fc)) < 0:
93              b, fb = c, fc
94          elif fc == (0, 0):
95              break
96          else:
97              a, fa = c, fc
98      return c
99  ## }}}
100 ## {{{ quadrature over wv[] and xv[]
101 class PolyplusQuadrature(PolyfitBase):
102     """
103     this class implements gaussian quadrature on a
104     discrete set of points.
105
106     if you want to compute
107
108         sum(f(x_i) * w_i for x_i, w_i in zip(xv, wv))
109
110     you can replace this with gaussian quadrature as
111
112         sum(f(z_i) * H_i for z_i, H_i in zip(Z, H))
113
114     the difference is that D = len(Z) is much smaller
115     than len(xv). Z and H are generated from a fit
116     plan for xv and wv. this module provides a function
117     to accurately sum f().
118
119     the quadrature formula is exact for polynomial
120     functions f() up to degree 2D-1.
121
122     for non-polynomial functions, the error is
123     proportional to the 2D-th derivative of f()
124     divided by (2D)!
125     """
126
127     def __init__(self, data):
128         "init self from a plan"
129         if isinstance(data, dict):
```

```
130             self.data = data
131         else:
132             self.data = data = data.to_data()["plan"].copy()
133 
134             data["x0"], data["x1"] = min(data["x"]), max(data["x"])
135 
136             data["roots"]   = { 0: [ ] }
137             data["schemes"] = { }
138 
139         self.b, self.c, self.g = data["b"], data["c"], data["g"]
140         self.x0, self.x1 = to_quad(data["x0"]), to_quad(data["x1"])
141         self.the_roots = data["roots"]
142         self.the_schemes = data["schemes"]
143 
144     def __call__(self, func, deg):
145         r"""
146         compute \sum_{i=1}^N w_i func(x_i) using the quadrature
147         scheme \sum_{i=0}^{deg} H_i func(z_i). this is exact if
148         func() is a poly of degree < 2D. func is assumed to take
149         and return a float.
150         """
151         f = lambda x: func(to_float(x))
152         return to_float(self.qquad(f, deg))
153 
154     def qquad(self, func, deg):
155         r"""
156         compute \sum_{i=1}^N w_i func(x_i) using the quadrature
157         scheme \sum_{i=0}^{deg} H_i func(z_i). this is exact if
158         func() is a poly of degree < 2D. func is assumed to take
159         and return a quad.
160         """
161         v = [ ]
162         for z, H in self.scheme(deg):
163             vappend(v, mul(H, to_quad(func(z))))
164         return vectorsum(v)
165 
166     def roots(self, k):
167         "return the roots of the orthogonal poly of degree k"
168         if k not in self.the_roots:
169             self.the_roots[k] = self._roots(k)
170         return self.the_roots[k]
171 
172     def scheme(self, k):
173         """
174         return the ordinates and christoffel numbers for
175         the quadrature scheme of order k
176         """
177         if k not in self.the_schemes:
178             self.the_schemes[k] = self._scheme(k)
179         return self.the_schemes[k]
180 
181     def _phi_k(self, x, k):
182         "internal: compute the k-th orthogonal poly at x"
183         x    = to_quad(x)
184         b    = self.b
185         c    = self.c
```

```
186         pjm1 = zero()
187         pj   = one()
188         for j in range(k):
189             pjp1 = sub(
190                 mul(sub(x, b[j]), pj),
191                 mul(c[j], pjm1)
192             )
193             pjm1 = pj
194             pj   = pjp1
195         return pj
196 
197     def _roots(self, k):
198         r"""
199         internal:
200 
201         compute the roots of \phi_k using the separation
202         property
203         """
204         ranges = [self.x0] + self.roots(k - 1) + [self.x1]
205         ret    = [ ]
206         func   = lambda x: self._phi_k(x, k)
207         for i in range(len(ranges) - 1):
208             a  = ranges[i]
209             fa = func(a)
210             b  = ranges[i + 1]
211             fb = func(b)
212             ret.append(bis(func, a, fa, b, fb))
213         return ret
214 
215     def _scheme(self, k):
216         """
217         internal:
218 
219         compute the quadrature scheme of order k using the
220         christoffel-darboux identity
221         """
222         b   = self.b
223         c   = self.c
224         g   = self.g
225         uno = one()
226         ret = [ ]
227         for r in self.roots(k):
228             v    = [ ]
229             pjm1 = zero()
230             pj   = one()
231             ## loop over polys
232             for j in range(k):
233                 ## add in the next term
234                 u = div(mul(pj, pj), g[j + 1])
235                 vappend(v, u)
236                 ## compute the next poly
237                 pjp1 = sub(
238                     mul(sub(r, b[j]), pj),
239                     mul(c[j], pjm1)
240                 )
241                 pjm1 = pj
```

```
242                 pj   = pjp1
243             ## save the root and christoffel number
244             ret.append(
245                 (r, div(uno, vectorsum(v)))
246             )
247         return ret
248 ## }}}
249
250 ## 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

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

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 $k = 0 \ldots D$ 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`