CWP-109P
May 1991

Discrete Approximation of Linear Functionals

by

Jack K. Cohen
Colorado School of Mines
Golden, CO 80401

David R. DeBaun
Unocal
Brea, CA 92621

Center for Wave Phenomena
Colorado School of Mines
Golden, Colorado 80401
(303)273-3557

ABSTRACT

A classical problem in numerical analysis is to determine the weights for finite difference approximations in terms of function values at equally spaced sample points. Equally classical is the criterion that the weights be determined by the constraint that the approximation be exact for the first n powers of x.

The algorithm presented solves this problem for an *arbitrary* linear functional—in contrast to the common textbook approach of treating separately the cases of interpolation, differential and integration operators. The algorithm also has a straightforward generalization to higher dimensions. Packages and example usages are given for the one, two and three dimensional cases—the implementation is short and easy to understand. Finally, a proof of validity is also given.

The case of unequally spaced samples is often important for finding stable approximations and, unfortunately, the algorithm does *not* seem to have a generalization to this case.

INTRODUCTION

An algorithm and its proof are presented that solve the problem:

> Determine the weights for a finite difference approximation to a linear functional using only function values at equally spaced sample points, subject to the criterion of *exact matching* for the first n (zero through $n-1$) powers of x.

The methodology has the flavor of the classical symbolic calculus developed by Boole [Boole, 1880; Milne-Thompson, 1933]. However, the algorithm given here can be applied uniformly to all linear functionals and a self-contained development and proof is given.

Because the algorithm involves only evaluations, finite Taylor Series expansions, and algebraic simplifications, it is easy to implement in a language supporting symbolic manipulations. A procedural implementation would be significantly clumsier; indeed the core of the Mathematica implementation is just 3 lines of straightforward code that "hides" all the computational aspects and presents the user with an interface requiring only the natural inputs: the functional, the base point and the sample spacing.

THE ALGORITHM

Given n equally spaced sample points, $x_0, x_1, \ldots, x_{n-1}$, the algorithm is:

1. Define a certain function $W(X)$ given below.

2. Compute the Taylor Series of $W(X)$ about $X = 1$, discarding terms of order n and higher.

3. Expand the result as a polynomial in X.

4. The desired weights are the coefficients of this polynomial.

For sample points with spacing, h, the generating function, $W(X)$, is given by

$$W(X) = X^{-x_0/h} \sum_{j=0}^{n-1} \frac{m_j}{j! h^j} \ln^j X. \tag{1}$$

Here the "moment", m_j is the result of applying the given linear functional, L, to x^j, that is:

$$m_j = L[x^j] \tag{2}$$

Analogously, for two dimensional linear functionals approximated on a grid with spacings h and k, the generating function is

$$W(X, Y) = X^{-x_0/h} Y^{-y_0/k} \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} \frac{m_{ij}}{i! j! h^i k^j} \ln^i X \ln^j Y, \tag{3}$$

with

$$m_{ij} = L[x^i y^j] \tag{4}$$

and so on for higher dimensions.

The proof of the algorithm is given in Appendix A.

It is worth mentioning that the derivative operators at $x = 0$ form a basis for the linear functionals using a fixed set of sample points. This follows since the functional,

$$D^j[f(x)]|_{x=0}$$

annililates all monomials except x^j. Thus, every moment but the jth vanishes, yielding the single term,

$$W(X) = X^{-x_0/h} \frac{\ln^j X}{h^j}.$$

Tables of the weights for these simple functionals and the moments for a particular linear functional using these sample points would allow computation of the weights for that functional by forming the appropriate linear combination. These considerations can be generalized and used to increase the efficiency of the implementation for specific classes of functionals. However, the direct Mathematica implementation presented in the next section is more suitable for casual use.

THE MATHEMATICA IMPLEMENTATION

Appendix B gives the package containing functions for 1, 2 and 3 dimensional functionals for arbitrary sample spacing. Here, the implementation for 1 dimension is outlined; the extension to the multi-dimensional case is routine.

Given a linear functional L, a basepoint x_0, and sample spacing h, we can form the W in equation (1) by having Mathematica compute the required moments:

$$x^{-x0/h}\text{Sum}[L[x^k]\text{Log}[x]^k/(h^k k!), k, 0, n-1].$$

Steps 2 and 3 of the algorithm can be ipmlemented by the single statement:

$$\text{Collect}[\text{Expand}[\text{Normal}[\text{Series}[w, x, 1, n-1]]], x].$$

Here, the call to Normal accomplishes the discarding of the higher powers required in step 2 and the Expand corresponds to step 3.

All that remains is to print the result in a meaningful way and supply an interface giving the user reasonable default values for the parameters.

The function call for computing the weights for one dimensional functionals has the form:

```
Weights[L, n, (h), (x0), (x)]
```

The first two arguments are mandatory: the linear functional name and the number of sample points. The third argument is the sample spacing and defaults to 1. The fourth argument is the basepoint with default being the choice that provides a centered set of sample points. The last argument is the variable of the functions acted on by the linear functional and defaults to literal x.

The function call for computing the weights for 2 dimensional functionals has the form:

```
Weights2d[L, nx, (ny), (hx), (hy), (x0), (y0), (x), (y)]
```

The new features of the usage are that ny, the number of points in the y-direction, defaults to nx, the number of points used in the x-direction. Also, the sample spacings in the x and y directions *both* default to unity. Likewise the defaults for *both* the x and y basepoints provide centered formulas and the default function variables are respectively x and y.

The function call, Weights3d, for 3 dimensional functionals is entirely analogous to that for Weights2d.

EXAMPLES

For a first example define the derivative functional as the first derivative evaluated at $x = 0$ as

```
deriv1[f_] := D[f, {x, 1}] /. x -> 0
```

Then use the Weights function to compute the weights for an asymmetric approximation using the quasi-symmetric sample points, $(-1, 0, 1, 2)$:

```
Weights[deriv1, 4, 1, -1]
```

```
-2 f   - 3 f  + 6 f  - f
    -1      0      1    2
-------------------------
            6
```

To obtain a completely symmetric approximation, use the default spacing and base point:

```
Weights[deriv1, 4]
```

```
f       - 27 f       + 27 f    - f
 -(3/2)       -(1/2)       1/2    3/2
-------------------------------------
                 24
```

Next find the familiar 3 point approximation to the second derivative at 0 by defaulting the spacing and base point:

```
deriv2[f_] := D[f, {x, 2}] /. x -> 0
Weights[deriv2, 3]
```

```
f   - 2 f  + f
 -1      0    1
```

Now shift the base point to $-h$ and use sample spacing h:

```
Weights[deriv2, 3, h, -h]
```

```
-2 f  + f   + f
    0    -h    h
---------------
        2
       h
```

However, this *is* a symmetric approximation, so we can achieve the same result by defaulting the last argument:

```
Weights[deriv2, 3, h]

-2 f  + f   + f
    0    -h    h
---------------
        2
       h
```

Next consider a 3 point approximation to the integral from -1 to 1 ("integ1") defined by

```
integ1[f_] := Integrate[f, {x, -1, 1}].
```

The Weights function gives Simpson's rule:

```
Weights[integ1, 3]

f   + 4 f  + f
 -1      0    1
--------------
       3
```

Next approximate the integral from -2 to 2, again symmetrically, but not using the endpoints:

```
integ2[f_] := Integrate[f, {x, -2, 2}]
Weights[integ2, 3]

8 f   - 4 f  + 8 f
   -1      0      1
------------------
         3
```

to obtain Milne's 3 point "open" integration formula.

Now obtain the weights for the 11 point closed Newton-Cotes formula. In the absence of a symbolic manipulation language, problems like this are fraught with numerical accuracy difficulties.

```
integ5[f_] := Integrate[f, {x, -5, 5}]
Weights[integ5, 11]
```

```
(80335 f   + 531500 f   - 242625 f   + 1362000 f   -
        -5           -4           -3            -2

>   1302750 f   + 2136840 f  - 1302750 f  + 1362000 f  -
             -1            0            1            2

>   242625 f  + 531500 f  + 80335 f ) / 299376
             3           4          5
```

Exercise the multi-dimensional functions in the package with two simple examples:

```
xy0 = {x -> 0, y -> 0}
laplacian2d[f_] := D[f, {x, 2}] + D[f, {y, 2}] /. xy0
Weights2d[laplacian2d, 3]

f     + f     - 4 f    + f    + f
 -1,0    0,-1      0,0    0,1    1,0

xyz0 = {x -> 0, y -> 0, z -> 0}
laplacian[f_] := D[f, {x, 2}] + D[f, {y, 2}] +
 D[f, {z, 2}] /. xyz0
Weights3d[laplacian, 3]

f       + f       + f       - 6 f      + f      +
 -1,0,0    0,-1,0    0,0,-1      0,0,0    0,0,1

> f      + f
   0,1,0    1,0,0
```

Finally, return to the first example of the section to demonstrate the use of an alternate function variable:

```
deriv[f_] :=D[f, {u, 1}] /. u -> 0
Weights[deriv, 4, 1, -1, u]


-2 f   - 3 f  + 6 f  - f
    -1      0      1    2
------------------------
            6
```

It is remarkable that execution of the entire set of examples discussed in this section took only seconds on a NeXT Computer running version 1.2 of Mathematica.

ACKNOWLEDGEMENTS

David H. Carlson was a co-discoverer of a special case of the algorithm discussed above. We also thank TEXACO, USA for granting permission to publish this work.

REFERENCES

Boole, G., Treatise on the Calculus of Finite Differences, 3rd ed., Macmillan and Co., London, 1880

Milne-Thompson, L. M., 1933, The Calculus of Finite Differences: Macmillan and Co., London

APPENDIX A: PROOF OF THE ALGORITHM

Start from equation (1) defining the generating function, W, and replace the moments, m_j, by the exact matching condition,

$$\sum_{k=0}^{N-1} w_k x_k^j = m_j, \tag{5}$$

to obtain:

$$W(X) = \sum_{k=0}^{n-1} w_k X^{-x_0/h} \sum_{j=0}^{n-1} \frac{(x_k \ln X)^j}{j! h^j}. \tag{6}$$

Recognize the beginning of the exponential series to replace the inner sum by:

$$\exp(\frac{x_k}{h} \ln X) + O[\ln^n X].$$

Thus, since $\ln X \to 0$ *linearly* as $X \to 1$,

$$W(X) = \sum_{k=0}^{n-1} w_k X^{-x_0/h} (X^{x_k/h} + O[(X-1)^n]). \tag{7}$$

Finally, since $x_k = x_0 + kh$, this last result can be rewritten as

$$W(X) = \sum_{k=0}^{n-1} w_k X^k + O[(X-1)^n] \tag{8}$$

verifying the algorithm in the text.

Observe that for *unequally* spaced sample points, one still obtains an analog to the last equation—unfortunately, it doesn't imply the values of all the weights in this case.

APPENDIX B: MATHEMATICA IMPLEMENTATION

```
BeginPackage["Weights`"];

(* Declaration of public function names in Weights package *)

Weights::usage = "Weights[L, n, (h), (x0), (x)]                \n
                                                               \n
Compute weights for a discrete approximation of a linear       \n
functional, L, with unit step-size.                            \n
                                                               \n
n is the number of points to use in the approximation.         \n
                                                               \n
h is the step size and defaults to unity.                      \n
                                                               \n
x0 is the first point to be used in the approximation          \n
and defaults to the value giving a symmetric approximation.    \n
                                                               \n
x is the variable of the functions acted on by L and           \n
defaults to literal x.
"

Weights2d::usage = "Weights2d[L, nx, (ny), (hx), (hy),
                              (x0), (y0), (x), (y)]
                                                               \n
                                                               \n
Compute weights for a discrete approximation of a 2-D linear   \n
functional, L, with unit step-size.                            \n
                                                               \n
nx is the number of points to use in the x-direction.          \n
ny is the number of points in the y-direction and              \n
defaults to nx.                                                \n
                                                               \n
hx is the step size in the x-directions and defaults to unity. \n
Similarly for hy.                                              \n
                                                               \n
x0 is the first point to be used in the approximation          \n
in the x-direction and defaults to the value giving a          \n
symmetric approximation.  Similarly for y0.                    \n
                                                               \n
x and y are the variables of the functions acted on by         \n
L and default respectively to literal x and y.
"
```

```
Weights3d::usage = "Weights3d[L, nx, (ny), (nz), (hx), (hy), (hz),
                                (x0), (y0), (z0), (x), (y), (z)]
                                                                  \n
                                                                  \n
Compute weights for a discrete approximation of a 3-D linear      \n
functional, L, with unit step-size.                               \n
                                                                  \n
nx is the number of points to use in the x-direction.             \n
Similarly for ny and nz with ny defaulting to nx and nz to ny.    \n
                                                                  \n
hx is the step size in the x-directions and defaults to unity.    \n
Similarly for hy and hz.                                          \n
                                                                  \n
x0 is the first point to be used in the approximation             \n
in the x-direction and defaults to the value giving a             \n
symmetric approximation.  Similarly for y0 and z0.                \n
                                                                  \n
x, y, z are the variables of the functions acted on by            \n
L and default respectively to literal x, y ,z.
"


Format[f[arg__]] := Subscripted[f[arg]];


Begin["`private`"];

Weights[L_, n_Integer?Positive, h_:1, x0_:Automatic, x_:Weights`x] :=

Block [
        {w, poly, coefs, k, nm1 = n-1, shift = -x0/h},

        If [x0 === Automatic, shift = nm1/2];

        w     = x^(shift) Sum[L[x^k] Log[x]^k/(h^k k!), {k, 0, nm1}];

        poly  = Collect[Expand[Normal[Series[w, {x, 1, nm1}]]], x];

        coefs = Sum[Coefficient[poly, x, k] f[(k - shift) h],
        {k, 0, nm1}];

        Print[];
```

```
        Return[Print[Together[coefs]]]
]


Weights2d[L_, nx_Integer?Positive, ny_:Automatic, hx_:1, hy_:1,
             x0_:Automatic, y0_:Automatic,
     x_:Weights‘x, y_:Weights‘y] :=

Block [
        {w, poly, coefs, kx, ky, nyy = ny,
         nxm1 = nx-1, nym1 = ny-1, shiftx = -x0/hx, shifty = -y0/hy},

        If [ny === Automatic, nyy = nx; nym1 = nyy-1];
        If [x0 === Automatic, shiftx = nxm1/2];
        If [y0 === Automatic, shifty = nym1/2];

        w = x^shiftx y^shifty *
            Sum[
                    L[x^kx y^ky] *
                    Log[x]^kx/(hx^kx kx!) Log[y]^ky/(hy^ky ky!),
            {kx, 0, nxm1}, {ky, 0, nym1}];

        poly = Collect[Expand[Normal[Series[
           w, {x, 1, nxm1}, {y, 1, nym1}]]],
               {x, y}];

        coefs = Sum[
                    Coefficient[Coefficient[poly, x, kx], y, ky] *
                    f[(kx - shiftx) hx, (ky - shifty) hy],
                {kx, 0, nxm1}, {ky, 0, nym1}];

        Print[];
        Return[Print[Together[coefs]]]
]


Weights3d[L_, nx_Integer?Positive, ny_:Automatic, nz_:Automatic,
                hx_:1, hy_:1, hz_:1,
                x0_:Automatic, y0_:Automatic, z0_:Automatic,
                x_:Weights‘x, y_:Weights‘y, z_:Weights‘z] :=

Block [
        {w, poly, coefs, kx, ky, kz, nyy = ny, nzz = nz,
```

```
 nxm1 = nx-1, nym1 = ny-1, nzm1 = nz-1,
 shiftx = -x0/hx, shifty = -y0/hy, shiftz = -z0/hz},

If [ny === Automatic, nyy = nx; nym1 = nyy-1];
If [nz === Automatic, nzz = nyy; nzm1 = nzz-1];
If [x0 === Automatic, shiftx = nxm1/2];
If [y0 === Automatic, shifty = nym1/2];
If [z0 === Automatic, shiftz = nzm1/2];

w = x^shiftx y^shifty z^shiftz *
        Sum[
            L[x^kx y^ky z^kz] Log[x]^kx/(hx^kx kx!) *
            Log[y]^ky/(hy^ky ky!) Log[z]^kz/(hz^kz kz!),
        {kx, 0, nxm1}, {ky, 0, nym1}, {kz, 0, nzm1}];

poly = Collect[Expand[Normal[Series[
           w, {x, 1, nxm1}, {y, 1, nym1}, {z, 1, nzm1}]]],
       {x, y, z}];

coefs = Sum[
            Coefficient[Coefficient[Coefficient[
                poly, x, kx], y, ky], z, kz] *
                f[(kx - shiftx) hx, (ky - shifty) hy,
                    (kz - shiftz) hz],
        {kx, 0, nxm1}, {ky, 0, nym1}, {kz, 0, nzm1}];

        Print[];
        Return[Print[Together[coefs]]]
]

End[];
Endpackage[];
```