Supplemental Notes for Bernoulli Factory Algorithms

This version of the document is dated 2021-02-06.

1 General Factory Functions

The algorithms for **general factory functions** work with two sequences of polynomials: one approaches the function *f*(*λ*) from above, the other from below, where *f* is a continuous function that maps the interval (0, 1) to (0, 1). (These two sequences form a so-called *approximation scheme* for *f*.) One requirement for these algorithms to work correctly is called the *consistency requirement*:

- For each sequence, the difference between one polynomial and the previous one must have non-negative Bernstein coefficients (once the latter polynomial is elevated to the same degree as the other).

The consistency requirement ensures that the polynomials approach the target function without crossing each other. Unfortunately, the reverse is not true in general; even if the upper polynomials "decrease" and the lower polynomials "increase" to *f*, this does not mean that the scheme will ensure consistency.

1.1 Schemes That Don't Work

In the academic literature (papers and books), there are many approximation schemes that involve polynomials that converge from above and below to a function. Unfortunately, most of them cannot be used as is to simulate a function *f* in the Bernoulli Factory setting, because they don't ensure the consistency requirement described earlier.

The following are approximation schemes with counterexamples to consistency.

In this section and the next, **fbelow** and **fabove** mean the k^{th} Bernstein coefficient for the lower or upper degree-*n* polynomial, respectively, where *k* is an integer in the interval [0, *n*].

First scheme. In this scheme (Powell 1981)**[(1)]**, let *f* be a twice differentiable function (that is, a C^2 continuous function, or a function with continuous "slope" and "slope-of-slope" functions). Then the upper polynomial of degree *n* has Bernstein coefficients as follows, for all *n*≥1:

- **fabove**(*n*, *k*) = *f*(*k*/*n*) + *M* / (8**n*),

where *M* is an upper bound of the maximum absolute value of *f*'s slope-of-slope function (second derivative), and where *k* is an integer in the interval [0, *n*].

And the lower polynomial of degree *n* has Bernstein coefficients as follows:

- **fbelow**(n, k) = f(k/n) + M / (8*n).

The counterexample involves the twice differentiable function *g*(*λ*) = sin(*π***λ*)/4 + 1/2.

For g, the Bernstein coefficients for—

- the degree-2 upper polynomial (b(5, k)) are [0.6542..., 0.9042..., 0.6542...], and
- the degree-4 upper polynomial (b(6, k)) are [0.5771..., 0.7538..., 0.8271..., 0.7538..., 0.5771...].

The degree-2 polynomial lies above the degree-4 polynomial everywhere in [0, 1]. However, to ensure consistency, the degree-2 polynomial, once elevated to degree 4, must have Bernstein coefficients that are greater than or equal to those of the degree-4 polynomial.

- Once elevated to degree 4, the degree-2 polynomial's coefficients are [0.6542..., 0.7792..., 0.8208..., 0.7792..., 0.6542...].

As we can see, the elevated polynomial's coefficient 0.8208... is less than the corresponding coefficient 0.8271... for the degree-4 polynomial.

Second scheme. In this scheme, let f be a Lipschitz continuous function in [0, 1] (that is, a continuous function whose slope does not tend to a vertical slope anywhere in [0, 1]). Then the upper polynomial of degree n has Bernstein coefficients as follows, for all n≥2:

- **fabove**(n, k) = f(k/n) + (5/4) / sqrt(n),

where L is the maximum absolute "slope", also known as the Lipschitz constant, and (5/4) is the so-called Popoviciu constant, and where k is an integer in the interval [0, n] (Lorentz 1986)[(2)], (Popoviciu 1935)[(3)].

And the lower polynomial of degree n has Bernstein coefficients as follows, for all n≥1:

- **fbelow**(n, k) = f(k/n) + (5/4) / sqrt(n).

The following counterexamples show that this scheme can fail to ensure consistency, even if the set of functions is restricted to "smooth" functions (not just Lipschitz continuous functions).

For the first counterexample, the function $g(\lambda)$ = min(λ, 1−λ)/2 is Lipschitz continuous with Lipschitz constant 1. (In addition, g has a kink at 1/2, so that it's not differentiable, but this is not essential for the counterexample.)

For g, the Bernstein coefficients for—

- the degree-5 upper polynomial (b(5, k)) are [0.4874..., 0.5874..., 0.6874..., 0.6874..., 0.5874..., 0.4874...], and
- the degree-6 upper polynomial (b(6, k)) are [0.4449..., 0.5283..., 0.6116..., 0.6949..., 0.6116..., 0.5283..., 0.4449...].

The degree-5 polynomial lies above the degree-6 polynomial everywhere in [0, 1]. However, to ensure consistency, the degree-5 polynomial, once elevated to degree 6, must have Bernstein coefficients that are greater than or equal to those of the degree-6 polynomial.

- Once elevated to degree 6, the degree-5 polynomial's coefficients are [0.4874..., 0.5707..., 0.6541..., 0.6874..., 0.6541..., 0.5707..., 0.4874...].

As we can see, the elevated polynomial's coefficient 0.6874... is less than the corresponding coefficient 0.6949... for the degree-6 polynomial.

There is a similar counterexample that can be built:

- When g = sin(4*π*λ)/4 + 1/2, a "smooth" function with Lipschitz constant π, the counterexample is present between the degree-3 and degree-4 lower polynomials.

Thus, we have shown that this approximation scheme is not guaranteed to meet the consistency requirement for all Lipschitz continuous functions.

It is yet to be seen whether a counterexample exists for this scheme when n is restricted to powers of 2.

Third scheme. Same as the second scheme, but replacing (5/4) with the Sikkema constant, S = (4306+837*sqrt(6))/5832 (Lorentz 1986)[(2)], (Sikkema 1961)[(4)]. In fact, the same counterexamples for the second scheme apply to this one, since this scheme merely multiplies the offset to bring the approximating polynomials closer to f.

For example, the first counterexample for this scheme is almost the same as the first one for the second scheme, except the coefficients for—

- the degree-5 upper polynomial are [0.5590..., 0.6590..., 0.7590..., 0.7590..., 0.6590..., 0.5590...], and
- the degree-6 upper polynomial are [0.5103..., 0.5936..., 0.6770..., 0.7603..., 0.6770..., 0.5936..., 0.5103...].

And once elevated to degree 6, the degree-5 polynomial's coefficients are [0.5590..., 0.6423..., 0.7257..., 0.7590..., 0.7257..., 0.6423..., 0.5590...].

As we can see, the elevated polynomial's coefficient 0.7590... is less than the corresponding coefficient 0.7603... for the degree-6 polynomial.

1.2 Other Schemes

I have found how to extend the results of Nacu and Peres (2005)[(5)] to certain functions with a slope that tends to a vertical slope. Moreover, the polynomials satisfy the consistency requirement unlike with the schemes from the previous section.

For example, take a factory function $f(\lambda)$, the function to simulate using flips of a coin with unknown probability of heads of λ. The following scheme to build upper and lower polynomials can be used if $f(\lambda)$—

- is (1/2)-*Hölder continuous*, meaning its vertical slopes, if any, are no "steeper" than m*sqrt(λ), for some number m greater than 0, and
- in the interval [0, 1]— - has a minimum of greater than 0 and a maximum of less than 1, or - is *convex* (the rate of growth of its "slope" never decreases) and has a minimum of greater than 0, or - is *concave* (the rate of growth of its "slope" never increases) and has a maximum of less than 1.

Finding the *Hölder constant* m is non-trivial and it requires knowing whether f has a vertical slope and where, among other things.[(6)] But assuming m is known, then for all n that are powers of 2:

- **fbelow**(n, k) = $f(k/n) - (m*(2^{1/4} + 2^{2/4} + 2^{3/4} + 1))/n^{1/4}$ (or $f(k/n)$ if f is concave).
- **fabove**(n, k) = $f(k/n) + (m*(2^{1/4} + 2^{2/4} + 2^{3/4} + 1))/n^{1/4}$ (or $f(k/n)$ if f is convex).

Proofs are in the appendix.

2 SymPy Code for Building Approximation

Schemes

The following Python code uses the SymPy computer algebra library. It contains a method, named `approxscheme2`, that builds a scheme for approximating a continuous function $f(\lambda)$ whose domain is [0, 1], with the help of polynomials that converge from above and below to that function. Not all functions that admit a Bernoulli factory are supported yet. One example of the output follows.

Note that because numerical methods may be used in some cases, and because only a finite number of polynomials are generated and checked by the code, the approximation scheme is not guaranteed to be correct in all cases.

The `approxscheme2(func,x,kind,lip,double,levels)` method takes these parameters:

- `func`: SymPy expression of the desired function.
- `x`: Variable used by `func`.
- `kind`: a string specifying the approximation scheme, such as "c2" (see code for `buildParam`).
- `lip`: A manually determined parameter that depends on the 'kind', in case the parameter can't be found automatically.
- `double`: Whether to double the degree with each additional level (`True`, the default) or to increase that degree by 1 with each level (`False`).
- `levels`: Number of polynomial levels to generate. The first level will be the polynomial of degree 2 for the kinds "c1", "c0", or "sikkema", and degree 1 otherwise. Default is 9.

The method prints out text describing the approximation scheme, which can then be used in either of the **general factory function algorithms** to simulate $f(\lambda)$ given a black-box way to sample the probability λ. It refers to the functions **fbelow**, **fabove**, and **fbound**, which have the meanings given in those algorithms.

Also in the code below, the `c2params(func,x,n)` method returns symbolic expressions for parameters needed to apply the approximation scheme for C^2 continuous functions. It takes these parameters:

- `func`: SymPy expression of the desired function.
- `x`: Variable used by `func`.
- `n`: A Sympy variable used in the symbolic expressions for the two bounds. It indicates the degree of the polynomial for which the method should find upper and lower bounds.

The method returns a tuple containing three expressions in the following order: the absolute maximum "slope-of-slope" m (`m`) and the two bounds for **fbound(*n*)** (`bound1` and `bound2`, respectively). See the notes in the section on general factory algorithms for more information.

```
def upperbound(x, boundmult=1000000000000000):
    # Calculates a limited-precision upper bound of x.
    boundmult = S(boundmult)
    return ceiling(x * boundmult) / boundmult

def lowerbound(x, boundmult=1000000000000000):
    # Calculates a limited-precision lower bound of x.
    boundmult = S(boundmult)
    return floor(x * boundmult) / boundmult

def degelev(poly, degs):
```

```
    # Degree elevation of Bernstein-form polynomials.
    # See also Tsai and Farouki 2001.
    n = len(poly) - 1
    ret = []
    nchoose = [math.comb(n, j) for j in range(n // 2 + 1)]
    degschoose = (
        nchoose if degs == n else [math.comb(degs, j) for j in range(degs // 2 + 1)]
    )
    for k in range(0, n + degs + 1):
        ndk = math.comb(n + degs, k)
        c = 0
        for j in range(max(0, k - degs), min(n, k) + 1):
            degs_choose_kj = (
                degschoose[k - j]
                if k - j < len(degschoose)
                else degschoose[degs - (k - j)]
            )
            n_choose_j = nchoose[j] if j < len(nchoose) else nchoose[n - j]
            c += poly[j] * degs_choose_kj * n_choose_j / ndk
        ret.append(c)
    return ret

def c2params(func, x, n):
  dd=buildParam("c2",func,x)
  offset=buildOffset("c2", dd, n)
  nm=nminmax(func, x);
  bound1=nm[0]-offset
  bound2=nm[1]+offset
  return (dd, bound1, bound2)

def buildOffset(kind, dd, n):
    if kind == "c2":
        # Use the theoretical offset for twice
        # differentiable functions. dd=max. abs. second derivative
        return dd / (n * 2)
    elif kind == "lipschitz":
        # Use the theoretical offset for Lipschitz
        # continuous functions. dd=max. abs. "slope"
        return dd * (1 + sqrt(2)) / sqrt(n)
    elif kind == "sikkema":
        # Use the theoretical offset for C0
        # Lipschitz continuous functions involving Sikkema's constant.
        # (If the function is not Lipschitz continuous the formula
        # is sikkema*W(1/sqrt(n)), where W(h) is the function's
        # modulus of continuity.)
        sikkema = S(4306 + 837 * sqrt(6)) / 5832
        return sikkema * dd / sqrt(n)
    elif kind == "c1":
        # Use the theoretical offset for C1
        # functions with a Lipschitz continuous slope.
        # dd=max. abs. "slope-of-slope" (Lipschitz constant
        # of first derivative). (G. G. Lorentz. Bernstein polynomials.
        # Chelsea Publishing Co., New York, second edition, 1986.)
        # (If the slope is not Lipschitz continuous the formula
        # is (3/4)*(1/sqrt(n))*W(1/sqrt(n)), where W(h)
        # is the _modulus of continuity_ of the slope function, that is,
        # the maximum difference between the highest and lowest
        # values of that function in any window of size h inside
        # the interval [0, 1]).
        return (S(3) / 4) * dd / n
    elif kind == "c0":
        # Use the theoretical offset for C0
```

```
        # Lipschitz continuous functions involving a more trivial bound
        # (by Popoviciu)
        return (S(5) / 4) * dd / sqrt(n)
    else:
        raise ValueError

def nminmax(func, x):
    # Find minimum and maximum at [0,1].
    try:
        return [minimum(func, x, Interval(0, 1)), maximum(func, x, Interval(0, 1))]
    except:
        print("WARNING: Resorting to numerical optimization")
    cv = [0, 1]
    df = diff(func)
    for i in range(20):
        try:
            ns = nsolve(df, x, (S(i) / 20, S(i + 1) / 20), solver="bisect")
            cv.append(ns)
        except:
            cv.append(S(i)/20)
            cv.append(S(i+1)/20)
    # Evaluate at critical points, and
    # remove incomparable values
    cv2 = []
    for c in cv:
        c2=func.subs(x, c)
        # Change complex infinity to infinity
        if c2==zoo: c2=oo
        try:
           Min(c2)
           cv2.append(c2)
        except:
           pass
    cv=cv2
    return [Min(*cv).simplify(), Max(*cv).simplify()]

def buildParam(kind, func, x, lip=None):
    if kind == "c2" or kind == "c1":
        try:
            # Maximum of second derivative.
            dd = nminmax(diff(diff(func)), x)
            dd = Max(Abs(dd[0]), Abs(dd[1])).simplify()
        except:
            # Unfortunately, SymPy's maximum and minimum are
            # not powerful enough to handle many common cases
            # of functions (notably piecewise functions), and
            # also has no convenient way to
            # minimize or maximize functions numerically.
            if lip == None:
                raise ValueError
            dd = S(lip)
    elif kind == "lipschitz" or kind == "sikkema" or kind == "c0":
        try:
            # Maximum of first derivative (Lipschitz constant)
            ff = func.rewrite(Piecewise)
            dd = nminmax(diff(diff(func)), x)
            dd = Max(Abs(dd[0]), Abs(dd[1])).simplify()
        except:
            if lip == None:
                raise ValueError
            dd = S(lip)
    else:
```

```
        raise ValueError
    return dd

def consistencyCheckInner(prevcurve, newcurve, ratio=1, diagnose=False, conc=None):
    n, prevcurve, prevoffset = prevcurve
    n2, newcurve, newoffset = newcurve
    degs = n2 - n
    prevoffset *= ratio
    newoffset *= ratio
    # NOTE: For the 'above' and 'below' cases, bounds ensure that in case of doubt,
    # the approximation is judged to be inconsistent
    # Below
    belowbernconew = [lowerbound(a - newoffset) for a in newcurve]
    maxbernconew = max(belowbernconew)
    if maxbernconew < 0:
        # Fully below 0
        pass  # return "offcurve"
    belowberncoold = degelev([upperbound(b - prevoffset) for b in prevcurve], degs)
    for oldv, newv in zip(belowberncoold, belowbernconew):
        if newv < oldv:
            # Inconsistent approximation from below
            if diagnose:
                print("Inconsistent from below")
                print(["n, n2, prevoffset, newoffset", n, n2, prevoffset, newoffset])
                print([S(c).n() for c in belowberncoold])
                print([S(c).n() for c in belowbernconew])
            return "incons"
    # Above
    bernconew = [upperbound(a + newoffset) for a in newcurve]
    minbernconew = min(bernconew)
    if minbernconew > 1:
        # Fully above 1
        pass  # return "offcurve"
    bernco old = degelev([lowerbound(b + prevoffset) for b in prevcurve], degs)
    for oldv, newv in zip(berncoold, bernconew):
        if oldv < newv:
            # Inconsistent approximation from above
            if diagnose:
                print("Inconsistent from above")
                print(["n, n2, prevoffset, newoffset", n, n2, prevoffset, newoffset])
                print([S(c).n() for c in berncoold])
                print([S(c).n() for c in bernconew])
            return "incons"
    return True

def isinrange(curve, ratio):
    n, curve, offset = curve
    offset *= ratio
    for c in curve:
       lb=lowerbound(c-offset)
       ub=upperbound(c+offset)
       if lb<0 or ub>1: return False
    return True

def concavity(func,x):
   nm=nminmax(diff(diff(func)),x)
   if nm[0]>=0: return "convex"
   if nm[1]<=0: return "concave"
   return None

def consistencyCheckCore(curvedata, ratio, diagnose=False):
    for i in range(len(curvedata) - 1):
```

```
        cons = consistencyCheckInner(
            curvedata[i], curvedata[i + 1], ratio=ratio, diagnose=diagnose
        )
        if cons == "incons":
            return False
    return True

def approxscheme2(func, x, kind="c2", lip=None, double=True, levels=9):
    print(func)
    curvedata = []
    if kind=="c1" or kind=="c0" or kind=="sikkema":
       deg = 2
    else:
       deg = 1
    nmm=nminmax(func, x)
    if nmm[0] < 0 or nmm[1] > 1:
       print("Function does not admit a Bernoulli factory")
       return
    conc = concavity(func, x)
    if conc != "concave" and nmm[0]<=0:
       print("Non-concave functions with minimum of 0 not yet supported")
       return
    if conc != "convex" and nmm[1]>=1:
       print("Non-convex functions with maximum of 1 not yet supported")
       return
    dd = buildParam(kind, func, x, lip)
    if dd==oo or dd==zoo:
       print("Function not supported for the scheme %s" % (kind))
       return
    if dd==0:
       if lip!=None:
           dd=S(lip)
       else:
           print("Erroneous parameter calculated for the scheme %s" % (kind))
           return
    for i in range(1, levels + 1):
        offset = buildOffset(kind, dd, deg)
        curvedata.append(
            (deg, [func.subs(x, S(j) / deg) for j in range(deg + 1)], offset)
        )
        if double:
            deg *= 2
        else:
            deg += 1
    offset = buildOffset(kind, dd, 1)
    if not consistencyCheckCore(curvedata, Rational(1)):
        print(
            "INCONSISTENT --> offset=%s [dd=%s, kind=%s]"
            % (S(offset).n(), upperbound(dd.n()).n(), kind)
        )
        consistencyCheckCore(curvedata, Rational(1), diagnose=True)
        return
    for cdlen in range(3, len(curvedata) + 1):
        left = Rational(0, 1)
        right = Rational(1, 1)
        for i in range(0, 6):
            mid = (left + right) / 2
            if consistencyCheckCore(curvedata[0:cdlen], mid):
                right = mid
            else:
                left = mid
        # NOTE: If 'ratio' appears to stabilize to much less than 0, then the
```

```
        # approximation scheme is highly likely to be correct.
        print(
            "consistent(len=%d) --> offset_deg1=%s [ratio=%s, dd=%s, kind=%s]"
            % (cdlen, S(offset * right).n(), right.n(), upperbound(dd.n()).n(), kind)
        )
    inrangedeg = -1
    for cd in curvedata:
        if isinrange(cd, right):
            inrangedeg=cd[0]
            break
    offsetn = buildOffset(kind, dd, symbols('n'))
    offsetn *= right
    conc = concavity(func, x)
    data = "* Let _f_(_&lambda;_) = %s.  " % (
        str(func.subs(x,symbols('lambda'))).replace("*","\*"))
    if double:
        data += "Then, for all _n_ that are powers of 2, starting from 1:\n"
    else:
        data += "Then:\n"
    data += "    * **fbelow**(_n_, _k_) = "
    if conc == "convex" and inrangedeg>=0:
        data += "1 if _n_&lt;%d; otherwise, " % (inrangedeg)
    data += "_f_(_k_/_n_)"
    if conc != "concave":
       data += " &minus; `%s`" % (str(offsetn.subs(x,symbols('lambda'))).replace("*","\*"))
    data += ".\n"
    data += "    * **fabove**(_n_, _k_) = "
    if conc == "concave" and inrangedeg>=0:
        data += "1 if _n_&lt;%d; otherwise, " % (inrangedeg)
    data += "_f_(_k_/_n_)"
    if conc != "convex":
       data += " + `%s`" % (str(offsetn.subs(x,symbols('lambda'))).replace("*","\*"))
    data += ".\n"
    if inrangedeg>=0:
       if conc == "concave" or conc == "convex":
         data += (
           "    * **fbound**(_n_) = [0, 1].\n"
         )
       else:
         data += (
           "    * **fbound**(_n_) = [0, 1] if _n_&ge;%d, or [&minus;1, 2] otherwise.\n" %
(inrangedeg)
         )
    print(data)
    return
```

3 Notes

- (1) Powell, M.J.D., *Approximation Theory and Methods*, 1981
- (2) G. G. Lorentz. Bernstein polynomials. 1986.
- (3) Popoviciu, T., "Sur l'approximation des fonctions convexes d'ordre supérieur", Mathematica (Cluj), 1935.
- (4) Sikkema, P.C., "Der Wert einiger Konstanten in der Theorie der Approximation mit Bernstein-Polynomen", Numer. Math. 3 (1961).
- (5) Nacu, Şerban, and Yuval Peres. "**Fast simulation of new coins from old**", The Annals of Applied Probability 15, no. 1A (2005): 93-115.
- (6) Specifically, the constant *m* is an upper bound of abs(*f*(*x*)−*f*(*y*))/sqrt(abs(*x*−*y*)) for all *x*, *y* pairs, where *x* and *y* are each in [0, 1] and *x* != *y*. However, this bound can't directly be calculated as it would involve

checking an infinite number of x, y pairs.

4 Appendix

4.1 Proofs for Hölder Function Approximation Scheme

There is an easy extension to lemma 6(i) of Nacu and Peres (2005)[(5)] to certain functions with a slope that tends to a vertical slope. Specifically, it applies to any *Hölder continuous* function, which means a continuous function whose slope doesn't go exponentially fast to a vertical slope.

The parameters α and M, in the lemma below, mean that the function is no "steeper" than $M*\lambda^{\alpha}$; α is in the interval (0, 1] and M is greater than 0.

Lemma 1. *Let f(λ) be a continuous function that maps [0, 1] to [−1, 1], and let X be a hypergeometric(2*n, k, n) random variable. If f is α-Hölder continuous with Hölder constant M, then abs(**E**[f(X/n)] − f(k/(2*n))) is bounded from above by* $M*(1/(2*n))^{\alpha/2}$.

Proof. abs($\mathbf{E}[f(X/n)] - f(k/(2*n))$) ≤ $\mathbf{E}$[abs($f(X/n) - f(k/(2*n))$)] ≤ $M*\mathbf{E}[\text{abs}(X/n - k/(2*n))]^{\alpha}$ (by the definition of Hölder continuous functions) ≤ $M*(\mathbf{E}[\text{abs}(X/n - k/(2*n))]^2)^{\alpha/2}$ = $M*\mathbf{Var}[X/n]^{\alpha/2} \le M*(1/(2*n))^{\alpha/2}$. □

Notes:

1. **E**[.] means expected or average value, and **Var**[.] means variance.
2. A *Lipschitz-continuous* function has no slope that tends to a vertical slope, making it a 1-Hölder continuous function with M equal to its Lipschitz constant.
3. An α-Hölder continuous function in [0, 1] is also β-Hölder continuous for any β less than α.

Theorem 1. *Let f(λ) be an α-Hölder continuous function with Hölder constant M that maps [0, 1] to (0, 1). The following Bernstein coefficients (**fabove**(n, k) for the upper polynomials, and **fbelow**(n, k) for the lower polynomials) form an approximation scheme that meets conditions (i), (iii), and (iv) of Proposition 3 of Nacu and Peres (2005)[(5)], for all n≥1, and thus can be used to simulate f via the algorithms for general factory functions described at the top of this page:*

- ***fbelow**(n, k) = f(k/n) − δ(n) (kth Bernstein coefficient of lower nth degree polynomial).*
- ***fabove**(n, k) = f(k/n) + δ(n) (kth Bernstein coefficient of upper nth degree polynomial).*

Where δ(n) is a solution to the functional equation $\delta(n) = \delta(2*n) + M*(1/(2*n))^{\alpha/2}$.

Proof. Follows from Lemma 1 above as well as the proof of Proposition 10 of Nacu and Peres (2005)[(5)]. □

Proposition 1.

1. *Let f be as given in Theorem 1, except f is concave and may have a minimum of 0.*

*The approximation scheme remains valid if **fbelow**(n, k) = f(k/n), rather than as given in Theorem 1.*

2. *Let f be as given in Theorem 1, except f is convex and may have a maximum of 1. The approximation scheme remains valid if **fabove**(n, k) = f(k/n), rather than as given in Theorem 1.*

Proof. Follows from Theorem 1 and Jensen's inequality. □

Unfortunately, there is no easy way to solve the functional equation above in a way that works for all α. However, the following examples show solutions that lead to approximation schemes that work for any α-Hölder continuous function with certain values of α.

- If α is 1/2 or greater: $\delta(n) = (M*(2^{1/4} + 2^{2/4} + 2^{3/4} + 1))/n^{1/4}$. (Solved via SymPy: `rsolve(Eq(f(n),f(n+1)+z*(1/(2*2**n))**((S(1)/2)/2)),f(n)).subs(n,log(n,2)).simplify()`.)
- If α is 2/3 or greater: $\delta(n) = 2^{2/3}*M*(2^{1/3} + 2^{2/3} + 2)/(2*n^{1/3})$.

5 License