**The Set of All Real, Monotonically-Increasing $N$th Order Polynomials that Map the $(0,1)$ Interval onto Itself, Together with Sensitivities**

William Menke, September 30, 2024

Summary: We construct the set $Z_N$ of real, $N$th order polynomials $z(x, \mathbf{r}, \mathbf{R})$ that map the $(0,1)$ interval onto itself. Here, $\mathbf{r}$ is a complex vector of length $K$ and $\mathbf{R}$ is a real vector of length $L$ that parameterize the map (with $N = 2K + L + 1$). We also derive the sensitivities of such a polynomial to perturbations in $(\mathbf{r}, \mathbf{R})$; that is, the partial derivatives of $z$ with respect to the real and imaginary parts of the elements of $\mathbf{r}$ and the elements of $\mathbf{R}$.

Theory

Consider a real $N$th order polynomial $p(x)$. Its slope $s(x) \equiv dp/dx$ is a real $(N-1)$-order polynomial, with roots that either are purely real or occur in complex conjugate pairs. In order for $p(x)$ to be monotonically-increasing on the interval $(0,1)$, its slope must be everywhere non-negative on that interval. Consequently, $s(x)$ may have no zeros on the $(0,1)$ interval of the real axis, unless they occur in pairs (in which case $p(x)$ has an inflection point).

Such a polynomial can be constructed from the product of $K$ quadratic polynomials (each with its two roots occurring in complex conjugate pairs) and $L$ monomials (each with a real root)

$$s(x, \mathbf{r}, \mathbf{R}) \equiv \prod_k^K q(x, r_k) \prod_l^L m(x, R_l)$$

with $\quad q(x, r_k) \equiv (x - r_k)(x - r_k^*) = \left[\left(r_k^R\right)^2 + \left(r_k^I\right)^2\right] - 2r_k^R x + x^2 \quad$ and $\quad m(x, R_l) \equiv (x - R_l)$

Here, we assume that the $r$s are distinct and the $R$s are outside the $(0,1)$ interval of the real axis. This polynomial has order $N = 2K + L$.

The polynomials $p(x, \mathbf{r}, \mathbf{R})$ can be constructed using Vieta's formula to calculate the polynomial coefficients of $s(x, \mathbf{r}, \mathbf{R})$, and then integrating to $p(x, \mathbf{r}, \mathbf{R})$ by modifying the coefficients

$$\text{given } c_i \text{ such that } \frac{ds}{dr_i^R} = c_1 + c_2 x + c_3 x^2 + \cdots \text{ then } \frac{dp}{dr_i^R} = c_0 + c_1 x + \frac{c_2}{2} x^2 + \frac{c_3}{3} x^3 + \cdots$$

The integration constant $c_0$ is zero because, by assumption, $p(x = 0, \mathbf{r}) = 0$. The polynomial

$$z(x, \mathbf{r}, \mathbf{R}) \equiv p(x, \mathbf{r}, \mathbf{R}) / p(x = 1, \mathbf{r}, \mathbf{R})$$

Is of order $N = 2K + L + 1$, is monotonically-increasing and satisfies both $z(x = 0, \mathbf{r}, \mathbf{R}) = 0$ and $z(x = 1, \mathbf{r}, \mathbf{R}) = 1$; that is, it maps the $(0,1)$ interval onto itself. Notice that the linear increasing mapping function $z(x) = x$ corresponds to a single quadratic with $r_1^R = 0$ and $r_1^I \gg 1$.

The set $Z_N$ contains all such real, $N$th order polynomials, with $N = (2K + L + 1)$, that map the $(0,1)$ interval onto itself. For $N$ odd, this set contains $(N/2 + 1)$ polynomials; that it $(K, L) = (0, N), (1, N-2), \cdots (N/2, 0)$. For $N$ even, it contains $((N-1)/2 + 1)$ polynomials.

In order to use the nonlinear least squares method to estimate the values of $(\mathbf{r}, \mathbf{R})$ that best-fit $z(x, \mathbf{r}, \mathbf{R})$ to $N_x$ observations $z_i^{obs}(x_i)$, one must be able to calculate the sensitivity of $z(x, \mathbf{r}, \mathbf{R})$ due to perturbations in $(\mathbf{r}, \mathbf{R})$. These partial derivatives are calculated in a sequence of steps.

First, consider the real monomial $m(x, R_j)$, with real root $R_j$ lying outside the $(0,1)$ interval. The derivative with respect to the root is

$$\frac{\partial m}{\partial R_j} = -1$$

Only one of the monomials in $s(x, \mathbf{r})$ contains a given $R_j$; the others are not dependent upon it. Consequently

$$\frac{\partial s}{\partial r_j^R} = -\prod_{k}^{K} q(x, r_k) \prod_{l \neq j}^{L} m(x, R_l)$$

Note that the derivative is a polynomial.

Second, consider the real quadratic polynomial $q(x, r_j)$, with roots $r_j \equiv r_j^R + i r_j^I$ and $r_i^* \equiv r_j^R - i r_j^I$. The derivative with respect to the real and imaginary parts of the roots are

$$\frac{\partial q}{\partial r_j^R} = 2r_j^R - 2x \quad \text{and} \quad \frac{\partial q}{\partial r_j^I} = 2r_j^I$$

Only one of the quadratics in $s(x, \mathbf{r})$ contains a given $r_j$; the others are not dependent upon it. Consequently

$$\frac{\partial s}{\partial r_j^R} = -2\left(x - r_j^R\right) \prod_{k \neq j}^{K} q(x, r_k) \prod_{l}^{L} m(x, R_l) \quad \text{and} \quad \frac{\partial s}{\partial r_j^I} = 2r_j^I \prod_{k \neq j}^{K} q(x, r_k) \prod_{l}^{L} m(x, R_l)$$

Note that both derivatives are polynomials.

Third, the derivatives of $p(x, \mathbf{r})$ are most efficiently found by reversing the order of integration and differentiation

$$\frac{\partial p}{\partial r_j^R} = \frac{\partial}{\partial r_j^R} \int_0^x s(x', \mathbf{r})\, dx' = \int_0^x \frac{\partial s}{\partial r_j^R}\, dx'$$

and similarly for $\partial p / \partial r_j^I$ and $\partial p / \partial R_i$, where the integration constant is chosen so that $\partial p / \partial r_i^R = 0$ at $x = 0$.

Finally, defining

$$z(x, \mathbf{r}, \mathbf{R}) \equiv AB^{-1} \quad \text{with} \quad A \equiv p(x, \mathbf{r}, \mathbf{R}) \quad \text{and} \quad B \equiv p(x = 1, \mathbf{r}, \mathbf{R})$$

the chain rule gives the derivative with respect to the real part of $r_i$ as

$$\frac{\partial z}{\partial r_i^R} = \frac{\partial A}{\partial r_i^R} B^{-1} - A \frac{\partial B}{\partial r_i^R} B^{-2} \quad \text{with} \quad \frac{\partial A}{\partial r_i^R} \equiv \frac{\partial}{\partial r_i^R} p(x, \mathbf{r}, \mathbf{R}) \quad \text{and} \quad \frac{\partial B}{\partial r_i^R} = \frac{\partial}{\partial r_i^R} p(x = 1, \mathbf{r}, \mathbf{R})$$

and similarly for $\partial z / \partial r_i^I$ and $\partial z / \partial R_i$.

Test of formulas. We examine an exemplary case (Fig. 1), with

$$K = 3 \quad \text{and} \quad \begin{bmatrix} r_0^R \\ r_1^R \\ r_2^R \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.2 \\ 0.8 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} r_0^I \\ r_1^I \\ r_2^I \end{bmatrix} = \begin{bmatrix} 4.0 \\ 0.5 \\ 0.25 \end{bmatrix} \quad \text{and} \quad L = 2 \quad \text{and} \quad \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 1.7 \end{bmatrix}$$

The corresponding function $z(x)$ is shown in Fig. 2 and selective sensitivities are shown in Figs. 3 and 4. The sensitivities agree well with a numerical result calculated with finite differences.

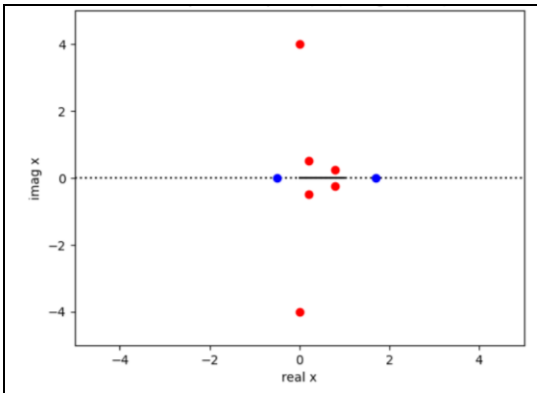


Fig. 1. Positions of complex pairs of zeros (red) and unpaired zeros (blue) for the exemplary case discussed in the text.

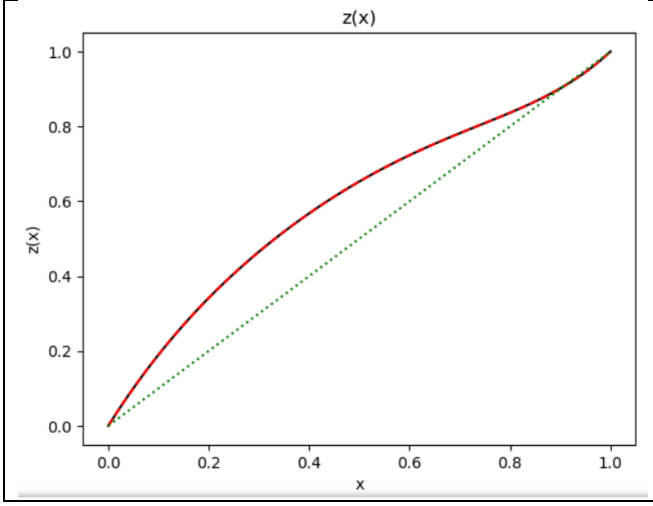


Fig. 2. Exemplary function $z(x)$ calculated using the algorithm described in this paper. The two identical versions are for the implementations in the `mapping()` method (black curve) and the `mappingderiv()` method (red dashed curve). The function $z(x) = x$ (green curve) is shown for comparison.

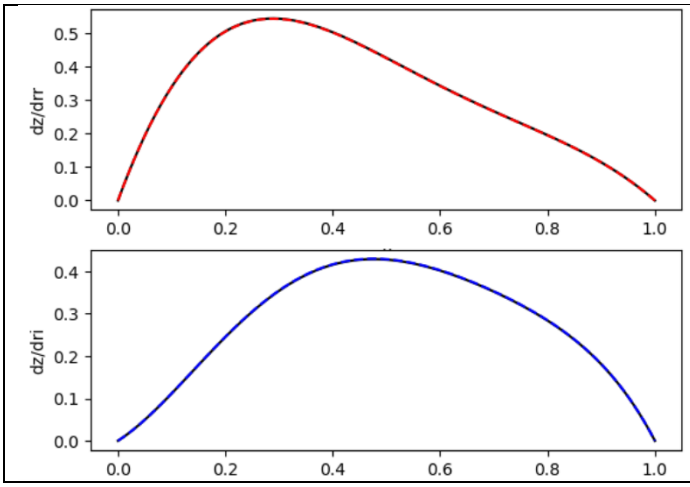


Fig. 3. (Top plot) Partial derivative $\partial z/\partial r_i^R$ (red dashed curve) for the exemplary function $z(x)$ and $i = 1$, calculated using the algorithm described in the paper as implemented by the `mappingderiv()` method. The partial derivative agrees well with a finite difference approximation (black curve). (Bottom plot) Same as top plot, except for $\partial z/\partial r_i^I$ (blue dashed curve).

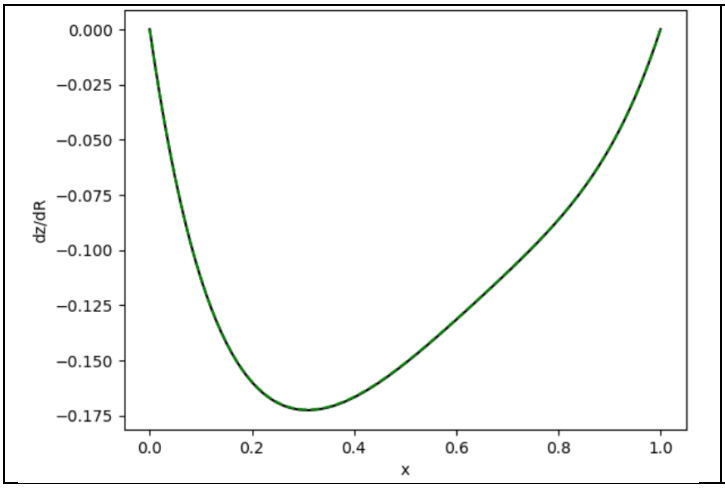


Fig. 4. Partial derivative $\partial z/\partial R_i$ (green dashed curve) for the exemplary function $z(x)$ and $i = 0$, calculated using the algorithm described in the paper as implemented by `mappingderiv()` method. The partial derivative agrees well with a finite difference approximation (black curve).

Appendix 1: Formula for the quadratic polynomial

$$q(x, r_i) = (x - r_i)(x - r_i^*) = \left(x - r_i^R - ir_i^I\right)\left(x - r_i^R + ir_i^I\right) =$$

$$= x^2 - xr_i^R + ixr_i^I - \; xr_i^R + \left(r_i^R\right)^2 - ir_i^R r_i^I \; - ir_i^I x + ir_i^R r_i^I - ii\left(r_i^I\right)^2 =$$

$$= \left[\left(r_i^R\right)^2 + \left(r_i^I\right)^2\right] - 2r_i^R x + x^2$$

Appendix 2: Effect of reversing the order of integration in the $(K, L) = (1,0)$ (quadratic, only) case.

Case 1: Integration first

$$s(x, r_1) = [(r_1^R)^2 + (r_1^I)^2] - 2r_1^R x + x^2$$

$$p(x, r_1) = [(r_1^R)^2 + (r_1^I)^2]x - r_1^R x^2 + \frac{1}{3}x^3$$

$$z(x, r_1) = AB^{-1} \quad \text{with} \quad A \equiv p(x, r_1) = [(r_1^R)^2 + (r_1^I)^2]x - r_1^R x^2 + \frac{1}{3}x^3$$

$$\text{and} \quad B \equiv p(x = 1, r_1) = [(r_1^R)^2 + (r_1^I)^2] - r_1^R + \frac{1}{3}$$

$$\frac{dz}{dr_1^R} = \frac{dA_R}{dr_1^R}B_R^{-1} - A_R\frac{dB_R}{dr_1^R}B_R^{-2} \quad \text{with} \quad \frac{dA_R}{dr_1^R} = \frac{dp}{dr_i^R} = 2r_1^R x - x^2 \quad \text{and} \quad \frac{dB_R}{dr_1^R} = \left.\frac{dp}{dr_i^R}\right|_{x=1} = 2r_1^R - 1$$

At $x = 0$, $A_R = dA_R/dr_1^R = 0$ so $dz/dr_1^R = 0$.

At $x = 1$, $A_R = B_R$ so $dz/dr_1^R = \frac{dA_R}{dr_1^R}A_R - \frac{dA_R}{dr_1^R}A_R = 0$

$$\frac{dz}{dr_1^I} = \frac{dA_I}{dr_1^I}B_I^{-1} - A_I\frac{dB_I}{dr_1^I}B_I^{-2} \quad \text{with} \quad \frac{dA_I}{dr_1^I} = \frac{dp}{dr_i^I} = 2r_1^I x \quad \text{and} \quad \frac{dB_R}{dr_1^I} = \left.\frac{dp}{dr_i^I}\right|_{x=1} = 2r_1^I$$

At $x = 0$, $A_I = dA_I/dr_1^I = 0$ so $dz/dr_1^I = 0$.

At $x = 1,\ A_I = B_I$ so $dz/dr_1^R = \frac{dA_I}{dr_1^I} A_I - \frac{dA_I}{dr_1^I} A_I = 0.$

Case 2: Differentiation first:

$$\frac{ds}{dr_i^R} = 2r_i^R - 2x \quad \text{and} \quad \frac{ds}{dr_i^I} = 2r_i^I$$

$$\frac{dp}{dr_i^R} = 2r_i^R x - x^2 + C_1 \quad \text{and} \quad \frac{dp}{dr_i^I} = 2r_i^I x + C_2$$

The requirement at $x = 0$, $dA/dr_1^R = 0$ inplies $C_1 = C_2 = 0$

$$\frac{dA_R}{dr_1^R} \equiv \frac{dp}{dr_i^R} = 2r_i^R x - x^2 \quad \text{and} \quad \frac{dA_I}{dr_1^I} \equiv \frac{dp}{dr_i^I} = 2r_i^I x$$

Note that this result matches the integration-first case.

Python methods

```
# 2024/09/26 -  Monotonic Polynomial code, by W. Menke

import numpy as np                      # matrices & etc
from numpy.polynomial import polynomial as pol   # polynomicals

# Mapping function z(x)
# Input:
# srr, sri: real and imaginary parts of complex roots,
#           need to be the same length, say K, and may be empty
# sR: real roots of length, say L, may be empty
# x: positions at which to evaluate z(x) and its derivatives, say of length Nx
# Output:
# zx          (Nx,1) vector of z(x)`
def mapping( srr, sri, sR, x ):
    Nx, i = np.shape(x);
    sr = croots(srr,sri,sR);
    sC = pol.polyfromroots(sr);
    sx = pol.polyval( x, sC );
    pIC = pol.polyint(sC);
    pIx = pol.polyval( x, pIC);
    zx = pIx / pIx[Nx-1,0];
    return( np.real(zx) );

# Mapping function z(x) and its partial derivatives
# Input:
# srr, sri: real and imaginary parts of complex roots, need to be the same length,
#           say K, and may be empty
# sR: real roots of length, say L, may be empty
# x: positions at which to evaluate z(x) and its derivatives, say of length Nx
# Output:
# zx          (Nx,1) vector of z(x)
# dzdrr_xm:   (Nx,K) vetor of d z / d srr
# dzdri_xm:   (Nx,K) vetor of d z / d sri
# dzdR_xm:    (Nx,L) vetor of d z / d srr
def mappingderiv( srr, sri, sR, x ):
    Nx, i = np.shape(x);

    # map function z(x)
```

```
    K, = np.shape(srr);
    L, = np.shape(sR);
    sr = croots(srr,sri,sR);        # roots of slope polynomial
    sC = pol.polyfromroots(sr);  # coefficients of slope polynomial
    pC = pol.polyint(sC);        # integrate, but results will not obey p(x=1)=1
    px = pol.polyval( x, pC);   # evaluate polunomial
    zx = px / px[Nx-1,0];      # normalize so z(x=1)=1

    # sensitivities are returned in matrices dzdrr_xm(x, k),
    # matrices dzdri_xm(x, k), dzdR_xm(x,L)
    # (which can be empty if K, L are zero)
    dzdrr_xm = np.zeros( (Nx, K) );
    dzdri_xm = np.zeros( (Nx, K) );
    dzdR_xm = np.zeros( (Nx, L) );

    # derivatives with respect to real and imaginary parts of
    # complex root associated with quadratic
    for k in range(K):
        # setup for dp/droot; part of polynomial not containing that root
        srrnotk = np.concatenate( (srr[0:k],srr[k+1:K]), axis=0 );
        srinotk = np.concatenate( (sri[0:k],sri[k+1:K]), axis=0 );
        srnotk = croots(srrnotk,srinotk,sR);
        # real part of dp/droot has the leading factor -2(x-rr)
        v =  np.zeros((1),dtype=complex);
        v[0] = complex(srr[k],0);
        # append root rr onto polynomial
        dsdrr_roots = np.concatenate( (srnotk, v), axis=0 );
        dsdrr_coefs = -2.0*pol.polyfromroots(dsdrr_roots);
        dpdrr_coefs = pol.polyint(dsdrr_coefs);
        dpdrr_x = pol.polyval( x, dpdrr_coefs);
        dzdrr_x = dpdrr_x/px[Nx-1,0] - px*dpdrr_x[Nx-1,0]*(px[Nx-1,0]**-2); # chain rule
        dzdrr_xm[0:Nx,k:k+1] = np.real(dzdrr_x);
        # imaginary part
        dsdri_roots = np.copy( srnotk );
        dsdri_coefs = 2.0*sri[k]*pol.polyfromroots(dsdri_roots);
        dpdri_coefs = pol.polyint(dsdri_coefs);
        dpdri_x = pol.polyval( x, dpdri_coefs);
        dzdri_x = dpdri_x/px[Nx-1,0] - px*dpdri_x[Nx-1,0]*(px[Nx-1,0]**-2); # chain rule
        dzdri_xm[0:Nx,k:k+1] = np.real(dzdri_x);
    # derivatives with respect to real root associated with monomial
    # setup for dp/droot; part of polynomial not containing that root
    for l in range(L):
        sRnotl = np.concatenate( (sR[0:l],sR[l+1:L]), axis=0 );
        srnotl = croots(srr,sri,sRnotl);
        # derivative
        dsdR_roots = np.copy( srnotl );
        dsdR_coefs = -pol.polyfromroots(dsdR_roots);
        dpdR_coefs = pol.polyint(dsdR_coefs);
        dpdR_x = pol.polyval( x, dpdR_coefs);
        dzdR_x = dpdR_x/px[Nx-1,0] - px*dpdR_x[Nx-1,0]*(px[Nx-1,0]**-2); # chain rule
        dzdR_xm[0:Nx,l:l+1] = np.real(dzdR_x);

    return( zx, dzdrr_xm, dzdri_xm, dzdR_xm );

# (for internal use)
# construct array of (2K+L) complex roots where
# L occur in complex conjugate pairs with real
# and imaginary parts rr and ri, and L are on
```

```
# the real asis with values R
def croots( rr, ri, R ):
    K, = np.shape(rr);
    L, = np.shape(R);
    r = np.zeros((2*K+L),dtype=complex);
    j=0;
    for i in range(K):
        r[j] = rr[i] + complex(0.0,1.0)*ri[i];
        j=j+1;
        r[j] = rr[i] + complex(0.0,-1.0)*ri[i];
        j=j+1;
    for i in range(L):
        r[j] = R[i];
        j=j+1;
    return(r);
```