SINGLE-FACTOR POLYNOMIAL COMPONENT ANALYSIS

JAN DE LEEUW

ABSTRACT. Polynomial component and factor analysis are defined. Algorithms, based on multidimensional array approximation methods, to fit a model with a single common factor to observed multivariate cumulants are described and applied to a psychometric example. Software in R is included.

CONTENTS

Date: Sunday 17th February, 2013 — 11h 24min — Typeset in LUCIDA BRIGHT.

2000 *Mathematics Subject Classification.* 62H25.

Key words and phrases. Factor Analysis, Component Analysis, Polynomials.

1. Model

Suppose $\underline{y}_j$ are random variables, with $j = 1, \cdots, m$. We use the convention of underlining random variables [Hemelrijk, 1966]. In our example we assume there exist *independent* and *centered* random variables $\underline{x}$ and $\underline{u}_j$, with $j = 1, \cdots, m$, such that

$$\underline{y}_j = \mathcal{P}_j(\underline{x}) + \underline{u}_j, \tag{1}$$

where the $\mathcal{P}_j$ are univariate polynomials of degree s. Thus

$$\mathcal{P}_j(x) = \sum_{p=0}^{s} \beta_{jp} x^p. \tag{2}$$

Equation (1) is sometimes called a *two-factor* model, because each variable is generated additively by the contribution of a single common factor $\underline{x}$ and a unique factor $\underline{u}_j$.

Note that if $\underline{u}_j \equiv 0$ for all $j = 1, \cdots, m$ the structure (1) is a *polynomial single-component* model, if some or all of the $\underline{u}_j$ are nonzero then (1) is a *polynomial single-factor* model.

Closely related previous work on the polynomial factor analysis model is in Mooijaart and Bentler [1986], who also discuss earlier work by Gibson and McDonald. In Mooijaart and Bentler [1986] higher order product moments, specifically third order moments, are used, and the special case of one-dimensional polynomial factor analysis is worked out and applied to a real example. Since Mooijaart and Bentler [1986] the emphasis has shifted away from higher order statistics to using instrumental variables and other more specialized techniques, and from component and factor analysis to general structural equation models. Good historical overviews of these more recent developments are in Yalcin and Amemiya [2001] and Mooijaart and Bentler [2010].

In this paper we go back to the simple one-dimensional component or factor model, we use higher-order statistics, specifically multivariate cumulants and generalized cumulants, and we use decomposition methods for multidimensional arrays to fit the models to sample cumulants. Thus, compared to Mooijaart and Bentler [1986], we switch from moments to cumulants, develop the case of non-normal factors, provide algorithms and software to deal with cumulants of arbitrary order, and develop algorithms and software based on fitting that use multilinear array modeling technology. All software is written in R [R Development Core Team, 2012].

2. Cumulants

We use the notation $\kappa(\underline{z}_1, \cdots, \underline{z}_n)$ for the multivariate cumulants of the random variables $\underline{z}_1, \cdots, \underline{z}_n$. Key results on cumulants needed for our purposes are in Brillinger [1981, Section 2.3], Speed [1983], McCullagh [1987, Chapters 2-4], and in Peccati and Taqqu [2011, Chapter 3].

From the Corollary to Proposition 4.2 in Speed [1983, p. 384] we see that for all r-tuples $(j_1, \cdots, j_r)$ with $1 \leq j_t \leq m$,

$$\kappa(\underline{y}_{j_1}, \cdots, \underline{y}_{j_r}) = \kappa(\mathcal{P}_{j_1}(\underline{x}), \cdots, \mathcal{P}_{j_r}(\underline{x})) + \kappa(\underline{u}_{j_1}, \cdots, \underline{u}_{j_r}), \tag{3}$$

and, by Speed [1983, Proposition 4.1], we have $\kappa(\underline{u}_{j_1}, \cdots, \underline{u}_{j_r}) = 0$ unless $j_1 = \cdots = j_r$, in which case it is $\kappa_r(\underline{u})$, the univariate cumulants of order r. This simple result illustrates one advantage of using cumulants instead of moments. The advantages will become more pronounced if there are more independent common factors.

We can now use the multilinear property of the cumulant operator given in Speed [1983, Proposition 4.2] to obtain

$$\kappa(\mathcal{P}_{j_1}(\underline{x}),\cdots,\mathcal{P}_{j_r}(\underline{x})) = \sum_{p_1=1}^{s}\cdots\sum_{p_r=1}^{s}\beta_{j_1p_1}\cdots\beta_{j_rp_r}\kappa(\underline{x}^{p_1},\cdots,\underline{x}^{p_r}). \tag{4}$$

For cumulants of order two or higher if one of the p_ν is zero then $\kappa(\underline{x}^{p_1},\cdots,\underline{x}^{p_r}) = 0$, and consequently the summations in (4) can start at all $p_\nu = 1$. This means we cannot recover β_{j0} from higher order cumulants. We can, however, combine (4) with

$$\mu(\mathcal{P}_j(\underline{x})) = \beta_{j0} + \sum_{p=1}^{s}\beta_{jp}\mu_p, \tag{5}$$

where the μ_p are the raw moments of $\underline{x}$.

3. Kernel

We derive a convenient general expression for $\kappa(\underline{x}^{p_1},\cdots,\underline{x}^{p_r})$, using a sum over all partitions to compute cumulants from raw moments. This is also used in De Leeuw and Kukuyeva [2012]. We leave the heavy combinatorial lifting to the `setparts()` function for set partitioning [West and Hankin, 2007], implemented in the `R` package `partitions` [Hankin, 2006]. Our cumulant calculations, using the partitions given by the `setparts()` function, are illustrated nicely in Table A-1 of Mendel [1991, page 297]. The key formula is

$$\kappa(x_1,\cdots,x_p) = \sum_{\pi\in\Pi}(-1)^{n(\pi)-1}(n(\pi)-1)!\prod_{\sigma\in\pi}\mu(\sigma), \tag{6}$$

where the summation is over all partitions of $\{1,2,\cdots,p\}$ and $n(\pi)$ is the number of subsets in the partition. The subsets in the partition are σ, and $\mu(\sigma)$ is the multivariate moment of the variables in subset σ.

Applied to our special case this gives

$$\kappa(\underline{x}^{p_1},\cdots,\underline{x}^{p_r}) = \sum_{\pi\in\Pi}(-1)^{n(\pi)-1}(n(\pi)-1)!\prod_{\sigma\in\pi}\mu_{n(\sigma)}, \tag{7}$$

where $n(\sigma)$ is the number of elements of σ, and μ_s is the raw moment of order s of $\underline{x}$.

3.1. **A Very Simple Example.**

Suppose we have m polynomials of degree three, and we look at cumulants of order two (i.e. covariances). See Mooijaart and Bentler [1986, p. 243] for basically the same example. Then (4) gives $K = BCB'$, with B an $m\times 3$ matrix of loadings, and C the 3×3 kernel matrix. The elements of C, in terms of raw moments, are

$$C = \begin{bmatrix} \mu_2-\mu_1^2 & \mu_3-\mu_1\mu_2 & \mu_4-\mu_1\mu_3 \\ \mu_3-\mu_1\mu_2 & \mu_4-\mu_2^2 & \mu_5-\mu_2\mu_3 \\ \mu_4-\mu_1\mu_3 & \mu_5-\mu_2\mu_3 & \mu_6-\mu_3^2 \end{bmatrix}.$$

For identification purposes we can suppose, without loss of generality, that $\mu_1 = 0$ and $\mu_2 = 1$. Then

$$C = \begin{bmatrix} 1 & \mu_3 & \mu_4 \\ \mu_3 & \mu_4-1 & \mu_5-\mu_3 \\ \mu_4 & \mu_5-\mu_3 & \mu_6-\mu_3^2 \end{bmatrix}.$$

For symmetric $\underline{x}$ we have vanishing odd moments, and thus

$$C = \begin{bmatrix} 1 & 0 & \mu_4 \\ 0 & \mu_4-1 & 0 \\ \mu_4 & 0 & \mu_6 \end{bmatrix},$$

and for standard normal $\underline{x}$ this becomes

$$C = \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & 0 \\ 3 & 0 & 15 \end{bmatrix}.$$

Note that the raw moments in C will have to satisfy the usual moment inequalities. Also note that even in the standard normal case, where the kernel C is completely known, the equation $K = BCB'$

does not identify B. In fact if S is a cogredient automorph [MacDuffee, 1946, p.65] of C, i.e if $SCS' = C$, then BS^{-1} is a solution of $K = BCB'$ whenever B is.

3.2. **Code.** For the general case of equation (7) we have written some R code, which can be undoubtedly improved a great deal, but is sufficiently fast for our purposes. See code segment 1. It is certainly faster and, perhaps more importantly, less error-prone, than manual calculation.

INSERT CODE SEGMENT 1 ABOUT HERE

As an example, consider $\kappa(\underline{x}, \underline{x}, \underline{x}^2, \underline{x}^2)$. We have

```
> fap(1,1,2,2)
     [,1] [,2] [,3] [,4] [,5]
[1,]    6    0    0    0    1
[2,]    5    1    0    0   -2
[3,]    4    2    0    0   -3
[4,]    4    1    1    0    2
[5,]    3    3    0    0   -2
[6,]    3    2    1    0    8
[7,]    2    2    2    0    2
[8,]    2    2    1    1   -6
```

and thus

$$\kappa(\underline{x}, \underline{x}, \underline{x}^2, \underline{x}^2) = \\ \mu_6 - 2\mu_1\mu_5 - 3\mu_2\mu_4 - 2\mu_3^2 + 2\mu_1^2\mu_4 + 8\mu_1\mu_2\mu_3 + 2\mu_2^3 - 6\mu_1^2\mu_2^2$$

Code segment 3 uses (7) to transform the univariate moments to a kernel array of cumulants.

INSERT CODE SEGMENT 3 ABOUT HERE

Instead of relating the multivariate kernel to the univariate moments of $\underline{x}$ we can also relate the cumulants in the kernel to the univariate cumulants of $\underline{x}$. We do not really use this in the current version of the paper, but we give the formulas and code in appendix A for later reference.

4. Loss Function

Given the results (3) and (4) it becomes straightforward to generalize the methods for linear component analysis in De Leeuw and Kukuyeva [2012] to the polynomial case.

We first compute, using De Leeuw [2012], one or more arrays of cumulants $A^{(r)} = \{a^{(r)}_{j_1\cdots j_r}\}$ from a given data matrix. We choose subsets $J_1, \cdots, J_r$ from $\{1, \cdots, m\}$, and then let j_t vary in J_t. It is suggested, by the code in De Leeuw [2012], to use cumulants up to order four (i.e. choose $r = 1, 2, 3, 4$), and to choose $J_1 = \cdots = J_r$, which means the data are super-symmetric cumulant arrays of order m^r. The kernel is a super-symmetric array of order s^r, and B is an $m \times s$ matrix of loadings.

We then define the least squares loss function by

$$\sigma_r(B, C^{(r)}) = \sum_{j_1 \in J_1} \cdots \sum_{j_r \in J_r} \left(a^{(r)}_{j_1\cdots j_r} - \sum_{p_1=0}^{s} \cdots \sum_{p_r=0}^{s} \beta_{j_1 p_1} \cdots \beta_{j_r p_r} c^{(r)}_{p_1 \cdots p_r}\right)^2, \tag{8}$$

and

$$\sigma_\star(B, C) = \sum_{r=1}^{\infty} w_r \sigma_r(B, C^{(r)}), \tag{9}$$

and minimize it over both B and C. The w_r are given non-negative weights. In our examples and code we always have $w_r = 0$ for $r > 4$, i.e. we do not use cumulants of order larger than four. In most cases w_1, w_2, w_3 and w_4 are either one or zero. But we could also consider taking the w_r inversely proportional to the number of elements in the cumulant array. Or we could take, say, w_2 very large and $w_3 = w_4 = 1$, which basically means we select from the subspace of solutions minimizing σ_2 the one that minimize $\sigma_3+\sigma_4$.

Note that this version of the loss function fits a component model, not a factor model. There are no special provisions in the loss function for the diagonal elements of the cumulant arrays. We concentrate on fitting components in this paper, and no unique factors.

For the fitting of C three different cases are of interest.

4.1. Fixed Kernel.

The problem is relatively simple if the kernel C is completely known. This requires, for example, the additional assumption that $\underline{x}$ has a standard normal distribution. The code in segments 4 and 5 can be used to compute moments and cumulants of univariate normal distributions.

$\boxed{\text{Insert Code Segment 4 about here}}$

$\boxed{\text{Insert Code Segment 5 about here}}$

In the case of a fixed kernel the loss function only needs to be minimized over B.

4.2. Free Kernel.

In the second special case we ignore the fact that the kernel array $C = \{c_{p_1\cdots p_r}\}$ is, according to (4), an array of cumulants of powers of the same variable. Thus we ignore the structure imposed by (7). We merely imposing super-symmetry on C, i.e. its elements are invariant under permutation of indices. We now minimize the loss function over both B and super-symmetric C.

Ignoring the structure of the kernel array is convenient, but it does imply we will not be able to distinguish a linear model with $s+1$ correlated components from a polynomial model of degree s. This is a major disadvantage of the method. Nevertheless, if using higher order moments forces identification, then we will be able to estimate factor loadings (or polynomial coefficients) consistently, up to a linear transformation.

4.3. **Structured Kernel.**

This is the more complicated case, and in several important ways the most interesting one. We minimize over B and over all C with the structure (7). Thus the kernel array, which for cumulants of order r and polynomials of order s has s^r elements, but all these elements are nonlinear functions of the first sr univariate moments (or cumulants). We minimize loss over B and over these vectors of moments.

Of course not all vectors are possible vectors of the first moments of a random variable. Defining all possible vectors of moments is the truncated version of the classical Moment Problem, which requires that the Hankel matrix formed from the moments is positive semi-definite. See, for example, Curto and Fialkow [1991] for the details. Instead of dealing with these complicated inequalities we can use $\mu_s = \int x^s dF(x)$ and minimize the loss function over all distribution functions F in some suitably large subspace.

5. Algorithms

Various segments of code needed for our minimizations, written in R, to minimize (9) are given in De Leeuw [2008b, 2012]; De Leeuw and Kukuyeva [2012]. We adopt and extend them to deal with our form of component analysis.

5.1. **Fixed Kernel.**

To minimize over B we use cyclic coordinate descent (CCD), i.e. we change one coordinate of B at a time while keeping all other fixed at their current values. One iteration to update B is a cycle in which all coordinates are changed. CCD is discussed, for example, in Zangwill [1969, Page 111-112], and in a similar context it has been used in ALSCAL [Y. Takane and de Leeuw, 1977]. It is currently very popular in machine learning. See

```
http://en.wikipedia.org/wiki/Coordinate_descent
```

for some examples and references.

If we deal with cumulants of order four, then changing one coordinate makes the loss function a polynomial of order eight of that coordinate. If the cumulants are of order three, the polynomial is of order six, and so on. Our algorithm computes a value of the polynomial at $2r$ values spaced equally around zero and then finds the minimum of the interpolating polynomial.

Because polynomial interpolation is highly unstable, we need some safeguards. We find the real root of the interpolating polynomial closest to zero. If the value of the polynomial at that root is less than the current best loss function value, and if the value is non-negative, then we use the root to improve the coordinate. Otherwise we move to the next coordinate. This seems to work reasonable well, although for larger problems the overall algorithm tends to be very slow.

There is no guarantee CCD will find the global minimum. As in ALSCAL, it is possible, at least in principle, to use global optimization methods for multivariate (non-negative) polynomials [**?**]. We have not tried this, and for larger problems the computational requirements are quite daunting.

5.2. **Free Kernel.** Fitting a free kernel is a simple modification of the basic algorithm. After a CCD cycle we update the kernel for given coordinates, and this is a simple linear least squares problem. The resulting Alternating Least Squares (ALS) algorithm, which alternates CCD cycles and kernel updates, is a block relaxation algorithm in the sense of De Leeuw [1994].

5.3. **Structured Kernel.** For the structured kernel solution we adjust the kernel in an ALS step by using a general purpose optimization technique, such as the ones given in the `optimx` package [Nash and Varadhan, 2011].

6. Real Example

We use the YouthGratitude data from the R package `psychotools` [Zeileis et al., 2012]. They are described in detail in the article by Froh et al. [2011]. The six seven-point Likert scale variables of the GQ-6 scale are used, with responses from 1405 students aged 10-19 years. In Table A1 of Froh et al. [2011] we see that a linear factor analysis gives loadings between .8 and .9 for items 1 and 2, between .6 and .7 for items 3,4, and 5, and between .2 and .4 for item 6.

```
library (psychotools)
library (cumulants)
data ("YouthGratitude")
gq6 <- as.matrix (YouthGratitude[,4:9])
gq6cums <- four_cumulants_direct (gq6)
cums <- ifelse (2 == 1:16, .1, 0)
moms <- univariateCumulantsToMoments (cums)
kers <- fourKernelMoments (4, moms)
```

6.1. Fixed Kernel.

For the kernel in this example we use the normal distribution with mean zero and variance 0.1. Throughout we use polynomials of order four and cumulants up to order four.

In our first run we set weights equal to the reciprocal of the number of elements in the cumulant array, i.e. to 1/36, 1/216 and 1/1296. Then input

```
library (polynom)
source ("tuckerCCD.R")
source ("tuckerMore.R")
source ("tuckerMost.R")
source ("tuckerAux.R")
invWeight<-tuckerCCD(gq6cums, kers, wts=c(0,1/36,1
   /216,1/1296), verbose=TRUE, itmax=500, eps=0.0001)
source ("plotPol.R")
pdf ("invWeight.pdf")
plotPol (invWeight $ x, gq6cums[[1]], "Inverse
   Weighted")
dev.off()
```

This produces the output

```
> invWeight<-tuckerCCD(gq6cums, kers,wts=c(0,1/36,1
   /216,1/1296),verbose=TRUE,itmax=500,eps=0.0001)
Iteration:    1 Old Loss:  1102.240089 New Loss:
   25.303006
Iteration:    2 Old Loss:    25.303006 New Loss:
   4.318540
Iteration:    3 Old Loss:     4.318540 New Loss:
   2.500216
.......
Iteration:  122 Old Loss:     0.460966 New Loss:
   0.460842
Iteration:  123 Old Loss:     0.460842 New Loss:
   0.460735
Iteration:  124 Old Loss:     0.460735 New Loss:
   0.460641
> invWeight
$x
         [,1]      [,2]      [,3]       [,4]
[1,] 3.168238 -5.977740 -3.527108  5.4318905
[2,] 3.354323 -7.020229 -3.094013  5.8583038
[3,] 4.646809 -3.798954 -7.753133  0.5136299
[4,] 2.489562 -5.671756 -1.670119  4.4158751
[5,] 1.641788 -6.590783 -0.822380  4.5471123
[6,] 5.403561  5.307253 -6.802472 -2.4778970
```

```

$itel
[1] 124

$tloss
[1] 0.4606414

$closs
$closs$loss2
[1] 5.413

$closs$loss3
[1] 41.62322

$closs$loss4
[1] 152.384
```

Figure 1 gives a plot of the six quartic polynomials from B. This uses the range $(-.5, +.5)$, which contains 90% of the normal distribution with variance 0.1. Using larger ranges shows annoying polynomial tails in regions in which there are no observations anyway.

INSERT FIGURE 1 ABOUT HERE

Six more analysis were done on these data with binary weights. Total loss in case TFT, for example, is just σ_2, while TTT is $\sigma_2 + \sigma_3 + \sigma_4$. Table 1 gives the loss function values, and Figure 2 plots the polynomials.

INSERT TABLE 1 ABOUT HERE

INSERT FIGURE 2 ABOUT HERE

The solutions show a great deal of variation, which may be due to local minima, to instability of sample cumulants, or to choice of binary weights. We cannot really expect to see a reasonable TFF solution, for example, because of the lack of identifiability. Solutions depending on higher order cumulants only may be inherently unstable. In our example the TTT solution, and the solution based on reciprocal weights, make the most sense. This may be because they use the maximum available amount of information.

6.2. **Free Kernel.** The results with inverse weights and a free kernel are given below.

```
> invWeight<-tuckerCCD(gq6cums, kers, wts=c(0,1/36,1
    /216,1/1296), kerFree= TRUE, verbose=TRUE, itmax
    =10, eps=0.0001)
Iteration:    1 Old Loss:  1102.240089 New Loss:
    0.753592
Iteration:    2 Old Loss:    0.753592 New Loss:
    0.095711
Iteration:    3 Old Loss:    0.095711 New Loss:
    0.073499
Iteration:    4 Old Loss:    0.073499 New Loss:
    0.071955
Iteration:    5 Old Loss:    0.071955 New Loss:
    0.071834
Iteration:    6 Old Loss:    0.071834 New Loss:
    0.071812
> invWeight $ x
          [,1]       [,2]       [,3]       [,4]
[1,] -1.975730 -12.443439   1.756074  8.1350576
[2,] -3.125244 -15.776952   2.912936 10.4515736
[3,] 12.304633   1.273219 -12.904921 -2.7115675
[4,] -3.413021   9.605715   5.233811 -7.2420908
[5,] -2.579544   9.583920   4.060020 -7.3160897
[6,] -2.719400   1.590880   5.801833 -0.2695233
> invWeight $ closs
$loss2
[1] 0.636493

```

```
$loss3
[1] 3.82678

$loss4
[1] 47.19397
```

There is obviously a huge improvement of the fit, but this is only natural because we added so many free parameters. The six polynomials for this solution are in Figure 3.

INSERT FIGURE 3 ABOUT HERE

As we indicated in Section 4.2 the fourth-order polynomial model can not really be distinguished from a linear component analysis in which the four components have a multivariate distribution whose cumulants we are estimating along with the loadings. Interpretation becomes slightly easier if we find the linear transformation that transforms the second order kernel cumulants to those of the normal distribution we used earlier. We then use the inverse of that linear transformation to transform B. The corresponding polynomials are in Figure 4.

INSERT FIGURE 4 ABOUT HERE

6.3. **Structured Kernel.** It turns out that the ALS approach which uses general optimization methods to adjust the structured kernel is not yet practical. Basically, function evaluations over high-dimensional arrays and long lists of partitions take too much time, and optimization methods become intolerably slow. It will be necessary to improve various components of the basic algorithm, and to design specialized optimimization methods to improve kernel estimates. Again, we could use CCA with a polynomial solver. This will be investigated in subsequent research.

7. Extensions and Elaborations

7.1. **Weights and Efficiency.** Adding weights $W = \{w_{j_1 \cdots j_r}\}$ to (8) is straightforward. It is more complicated to use the standard errors of the multivariate cumulants to improve (asymptotically distribution free) efficiency of the estimates. Again we will rely on (6).

7.2. **Constraints on Loadings.** Adding linear constraints (some elements are zero, some elements are equal) on B is again straightforward. This makes it possible to use different polynomial degrees for different variables. We can also impose ordinal constraints, for instance that polynomials are monotonic.

7.3. **Consistency.** Sample cumulants converge in probability to population cumulants. Minimizers of the loss function (or stationary points of the algorithm) are, under regularity conditions, continuous functions of the sample cumulants. Thus, under additional identifiability conditions, the estimates computed by the algorithm are consistent estimates of the population parameters.

7.4. **Multiple Common Factors.** Additional work is needed to adapt this approach to structures with more than one common factor. The common part of each variable will be a multivariate polynomial of the common factors, i.e. a symmetric multilinear form. It is consequently still true that the multivariate polynomial is a linear combination of powers and products of powers, and thus the basic results (3) and (4) continue to apply.

7.5. **Non-additive Unique Factors.** The additive combination rule in (1) is not as compelling as in the linear case. This is also the reason we have not paid much attention to factor analysis, as opposed to component analysis, in this paper. The more general decomposition $\underline{y}_j = \mathcal{P}_j(\underline{x}, \underline{u}_j)$ becomes interesting, but this is more naturally handled by the extension to mutiple common factors. For this case (3) no longer applies, but we can continue to rely on (4). This more general description, together with work on more than one factor, will become a topic for further research.

7.6. **Convergence Accelleration.** In further work we plan to use the methods discussed in De Leeuw [2008a] to speed up convergence. It must be emphasized, however, that there are many more parts of the algorithm that can be accelerated in various ways.

References

D.R. Brillinger. *Time Series: Data Analysis and Theory*. Holden-Day Series in Time Series Analysis. Holden-Day Inc., San Francisco, CA, 1981. Expanded Edition.

R.E. Curto and L.A. Fialkow. Recursiveness, Positivity, and Truncated Moment Problems. *Houston Journal of Mathematics*, 17: 603–635, 1991.

J. De Leeuw. Block Relaxation Methods in Statistics. In H.H. Bock, W. Lenski, and M.M. Richter, editors, *Information Systems and Data Analysis*, Berlin, 1994. Springer Verlag. URL `http://preprints.stat.ucla.edu/131/131.ps.gz`.

J. De Leeuw. Polynomial Extrapolation to Accelerate Fixed Point Algorithms. Preprint Series 542, UCLA Department of Statistics, Los Angeles, CA, 2008a. URL `http://www.stat.ucla.edu/~deleeuw/janspubs/2008/reports/deleeuw_R_08i.pdf`.

J. De Leeuw. The Multiway Package. Preprint Series 535, UCLA Department of Statistics, Los Angeles, CA, 2008b. URL `http://www.stat.ucla.edu/~deleeuw/janspubs/2008/reports/deleeuw_R_08j.pdf`.

J. De Leeuw. Multivariate Cumulants in R. Unpublished, 2012. URL `http://www.stat.ucla.edu/~deleeuw/janspubs/2012/notes/deleeuw_U_12a.pdf`.

J. De Leeuw and I. Kukuyeva. Component Analysis Using Multivariate Cumulants. Unpublished, 2012. URL `http://www.stat.ucla.edu/~deleeuw/janspubs/2012/notes/deleeuw_kukuyeva_U_12.pdf`.

J.J. Froh, J. Fan, R.A. Emmons, G. Bono, E. S. Huebner, and P. Watkins. Measuring Gratitude in Youth: Assessing the Psychometric Properties of Adult Gratitude Scales in Children and Adolescents. *Psychological Assessment*, 23:311–324, 2011.

R.K.S. Hankin. Additive Integer Partitions in R. *Journal of Statistical Software*, 16(Code Snippet 1), 2006.

J. Hemelrijk. Underlining Random Variables. *Statistica Neerlandica*, 20:1–7, 1966.

K. Hornik. A CLUE for CLUster Ensembles. *Journal of Statistical Software*, 14(12), 2005. URL `http://www.jstatsoft.org/v14/i12/`.

K. Hornik. *clue: Cluster ensembles*, 2012. URL `http://CRAN.R-project.org/package=clue`. R package version 0.3-45.

C. C. MacDuffee. *The Theory of Matrices*. Chelsea, New York, 1946.

P. McCullagh. *Tensor Methods in Statistics*. Chapman & Hall, Boca Raton, Florida, 1987.

P. McCullagh and A.R. Wilks. Complementary Set Partitions. *Proceedings of the Royal Society of London*, A 415:347–362, 1988.

J. Mendel. Tutorial on Higher-Order Statistics (Spectra) in Signal Processing and System Theory: Theoretical Resulys and Some Applications. *Proceedings of the IEEE*, 79(3):278–305, 1991.

A. Mooijaart and P. Bentler. Random Polynomial Factor Analysis. In E. Diday et al., editor, *Data Analysis and Informatics*, volume IV, pages 241–250, 1986.

A. Mooijaart and P.M. Bentler. An Alternative Approach for Nonlinear Latent Variable Models. *Structural Equation Modeling*, 17: 357–373, 2010.

J.C. Nash and R. Varadhan. Unifying Optimization Algorithms to Aid Software System Users: optimx for R. *Journal of Statistical Software*, 43(9):1–13, 2011. URL `http://www.jstatsoft.org/v43/i09/paper`.

G. Peccati and M.S. Taqqu. *Wiener Chaos: Moments, Cumulants and Diagrams*. Springer, Milan, Italy, 2011.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

T.P. Speed. Cumulants and Partition Lattices. *Journal of the Australian Mathematical Society*, 25:378–388, 1983.

L.J. West and R.K.S. Hankin. Set Partitions in R. *Journal of Statistical Software*, 23(Code Snippet 2), 2007.

F.W. Young Y. Takane and J. de Leeuw. Nonmetric Individual Differences in Multidimensional Scaling: An Alternating Least Squares Method with Optimal Scaling Features. *Psychometrika*, 42:7–67, 1977.

I. Yalcin and Y. Amemiya. Nonlinear Factor Analysis as a Statistical Method. *Statistical Science*, 16:275–294, 2001.

W. I. Zangwill. *Nonlinear Programming: a Unified Approach*. Prentice-Hall, Englewood-Cliffs, N.J., 1969.

A. Zeileis, C. Strobl, and F. Wickelmaier. *psychotools: Infrastructure for Psychometric Modeling*, 2012. URL `http://CRAN.R-project.org/package=psychotools`. R package version 0.1-4.

Appendix A. Generalized Cumulants

In (6) we express the generalized cumulants in terms of univariate moments. Alternatively we can express the generalized cumulants as sums of products of powers of univariate cumulants, which gives an alternative set of formulas. They can be used effectively, for example, in the case of a normally distributed common factor, which has vanishing cumulants of order three and larger.

We use what is sometimes called Malyshev's formula [Peccati and Taqqu, 2011, Proposition 3.2.1] to compute the cumulant expansions. We need, what McCullagh [1987, Page 59] calls the notion of partitions *complementary* to a partition π, which consists of all partitions of the same set such that their join with π in the lattice of partition is the trivial partition (which has only one block). The same notion is discussed in Brillinger [1981, Page 20-21] and Speed [1983, Page 384] as *decomposable* partitions relative to a given partition of a ragged array of random variables.

Malyshev's formula is

$$\kappa(\underline{x}^{p_1},\cdots,\underline{x}^{p_r}) = \sum_{\pi\in\Pi^\star}\prod_{\sigma\in\pi}\kappa_{n(\sigma)}(\underline{x}), \tag{10}$$

where summation is over all partitions complementary to the one of the first $p_1+\cdots+p_r$ integers defined by $(p_1,\cdots,p_r)$. Note that, unlike in (7), all product terms are added, there are no negative signs associated with some partitions.

In the appendix of McCullagh [1987] there are tables of complementary partitions. More detail on how these tables were constructed, plus some code in C, are in McCullagh and Wilks [1988]. Instead of using the tables, we have written some simplistic R code, that uses the `partitions` [Hankin, 2006; West and Hankin, 2007] and `clue` [Hornik, 2005, 2012] packages. It is given in code segment 2. Do not use it for large problems, because it is really inefficient.

$\boxed{\text{INSERT CODE SEGMENT 2 ABOUT HERE}}$

As an example, consider $\kappa(\underline{x},\underline{x},\underline{x}^2,\underline{x}^2)$. This corresponds with the partition $\{\{1\},\{2\},\{3,4\},\{5,6\}\}$. Then

```
> fcp(1,2,3,3,4,4)
     [,1] [,2] [,3] [,4]
[1,]    6    0    0    1
[2,]    5    1    0    4
[3,]    4    2    0   12
[4,]    4    1    1    4
[5,]    3    3    0    8
[6,]    3    2    1   24
[7,]    2    2    2    8
```

and thus

$$\kappa(\underline{x},\underline{x},\underline{x}^2,\underline{x}^2) =$$
$$\kappa_6 + 4\kappa_1\kappa_5 + 12\kappa_2\kappa_4 + 4\kappa_1^2\kappa_4 + 8\kappa_3^2 + 24\kappa_1\kappa_2\kappa_3 + 8\kappa_2^3.$$

For the standard normal this gives $\kappa(\underline{x},\underline{x},\underline{x}^2,\underline{x}^2) = 8\kappa_2^3 = 8$, while the corresponding formula using moments gives $\kappa(\underline{x},\underline{x},\underline{x}^2,\underline{x}^2) = \mu_6 - 3\mu_2\mu_4 + 2\mu_2^3 = 15 - 9 + 2 = 8$.

Appendix B. Code

Code Segment 1 Kernel elements expressed in raw moments

```
require (partitions)

fap <- function (...) {
    powers <- c(...)
    ns <- sum (powers)
    nl <- length (powers)
    nn <- 1 : nl
    s <- setparts (nl)
    q <- apply (s, 2, max)
    k <- ifelse (((q - 1) %% 2) == 0, 1, -1)
    f <- factorial (q - 1)
    p <- parts (ns)
    h <- ncol (p)
    m <- ncol (s)
    itel <- rep (0, h)
    for (i in 1 : m) {
        g <- drop (powers %*% ifelse (outer (s[, i], nn, "
            =="), 1 , 0))
        g <- sort (g[g != 0], decreasing = TRUE)
        g <- c (g, rep (0, ns - length (g)))
        itel <- itel + k[i] * f[i] * apply (p, 2, function
            (x) ifelse (all (x == g), 1, 0))
        }
    indi <- (1 : h)[itel != 0]
    jndi <- as.matrix((t (p))[indi,])
    jndi <- jndi[, colSums(jndi) > 0]
    return (cbind (jndi, itel[indi]))
}
```

Code Segment 2 Complementary partitions

```
require (partitions)
require (clue)

fcp <- function (...) {
    apart <- c(...)
    n <- length (apart)
    s <- setparts (n)
    p <- parts (n)
    h <- ncol (p)
    m <- ncol (s)
    k <- as.cl_partition (rep (1, n))
    l <- as.cl_partition (apart)
    itel <- rep (0, h)
    for (i in 1 : m) {
        r <- as.cl_partition (s[, i])
        u <- as.cl_partition (cl_join (r, l))
        if (u == k) {
            g <- as.vector (colSums (r$.Data))
            g <- c (g, rep (0, n - length (g)))
            itel <- itel + apply (p, 2, function (x)
                ifelse (all (x == g), 1, 0))
        }
    }
    indi <- (1 : h)[itel != 0]
    jndi <- (t (p))[indi,]
    if (is.vector (jndi)) {
        jndi <- matrix (jndi, 1, length (jndi))
    }
    jndi <- jndi[, colSums(jndi) > 0]
    print (cbind (jndi, itel[indi]))
}
```

Code Segment 3 Kernel Moments in Array

```
kernelMoments <- function (p, r, moments) {
    a <- array(0, rep (p, r))
    for (i in 1 : (p ^ r)) {
        k <- aplEncode (i, rep (p, r))
        q <- fap (k)
        m <- ncol (q)
        n <- nrow (q)
        for (j in 1 : n) {
            a[i] <- a[i] + q[j, m] * prod (moments [q
                [j, -m]])
        }
    }
    return (a)
}
```

Code Segment 4 Univariate Moments and Cumulants

```r
univariateMomentsToCumulants <- function (moments) {
    n <- length (moments)
    x <- rep (0, n)
    x[1] <- moments[1]
    for (i in 2 : n) {
        s <- moments [i]
        for (j in 1 : (i -1)) {
            s <- s - choose (i - 1, j) * x[i - j] * moments[j]
        }
        x[i] <- s
    }
    return (x)
}

univariateCumulantsToMoments <- function (cumulants) {
    n <- length (cumulants)
    x <- rep (0, n)
    x[1] <- cumulants[1]
    for (i in 2 : n) {
        s <- cumulants [i]
        for (j in 1 : (i -1)) {
            s <- s + choose (i - 1, j) * cumulants[i - j] * x[j]
        }
        x[i] <- s
    }
    return (x)
}
```

Code Segment 5 Moments of Standard Normal

```
normalEvenMoment <- function (n) {
    if ((n %% 2) == 1) {
        stop ("even moments only")
    }
    return (doubleFactorial (n - 1))
}

doubleFactorial <- function (n) {
    if ((n %% 2) == 0) {
        stop ("odd argument only")
    }
    k <- (n + 1) / 2
    return (prod (2 * (1 : k) - 1))
}
```

Code Segment 6 CCD Algorithm for Fixed/Free Kernel

```
source("tuckerMore.R")
source("tuckerMost.R")
source("tuckerFinal.R")
source("tuckerAux.R")
source("fap.R")
require (polynom)
require (apl)
require (optimx)

tuckerCCD <- function (cums, ker, wts = c(0, 0, 1, 0),
    kerType = "fixed", itmax = 500, eps = 1e-3, verbose =
    FALSE) {
    x <- tuckerInitial (cums, ker)
    ot <- polynomBasis (wts)
    th <- drop (ot [, 2])
    itel <- 1
    oloss <- totLoss (wts, cums, ker, x)
    repeat {
        x <- oneCCDCycle (wts, cums, ker, x, th, ot)
        xloss <- totLoss (wts, cums, ker, x)
        ker <-updateKernel (wts, cums, ker, x, kerType)
        nloss <- totLoss (wts, cums, ker, x)
        if (verbose) {
            cat ("Iteration: ", formatC (itel, 3, 3),
                 "Old Loss: ", formatC (oloss, 6, 10, "f"),
                 "After X: ", formatC (xloss, 6, 10, "f"),
                 "After C: ", formatC (nloss, 6, 10, "f"),
                    "\n")
            }
        if (((oloss - nloss) < eps) || (itel == itmax))
            break()
        oloss <- nloss
        itel <- itel + 1
    }
    return (list(x = x, itel = itel, tloss = nloss, kernel
        = ker, closs = allLoss (cums, ker, x)))
}
```

Code Segment 7 Tucker Subroutines

```
tuckerValue <- function (cums, ker, x) {
    r <- length (dim (cums))
    res <- as.vector (cums) - arrKronecker (repList (x, r))
        %*% as.vector (ker)
    return (sum (res * res))
}

tuckerInitial <- function (cums, ker) {
    e <- eigen (cums[[2]])
    p <- nrow (ker[[2]])
    x <- e $ vectors [, 1 : p] %*% diag (sqrt (e $ values
       [1 : p]))
    return (x %*% solve (t (chol (ker[[2]]))))
}

polynomBasis <- function (wts) {
    k <- max (which (wts > 0))
    th <- seq (-k, k, length = 2 * k + 1) / 10
    return (outer (th, 0 : (length (th) - 1), "^"))
}

tuckerFreeKernel <- function (cums, ker, x) {
    kk <- ker
    for (i in 2 : 4)  {
        kk [[i]] <- array (qr.solve (arrKronecker (repList
            (x, i)), as.vector (cums[[i]])), dim(kk[[i]]))
    }
    return (kk)
}

tuckerStructuredKernel <- function (moms, wts, cums, x) {
    ox <- optimx(moms, tuckerValueMoments, method = "Rcgmin
       ", control = list (trace = 1), wts = wts, cums =
       cums, x = x)
    browser()
}
```

Code Segment 8 More Tucker Subroutines

```
totLoss <- function (wts, cums, ker, x) {
    tloss <- 0
    for (i in 2 : 4) {
        if (wts[i] > 0) {
            tloss <- tloss + wts[i] * tuckerValue (cums[[i
                ]], ker[[i]], x)
        }
    }
    return (tloss)
}

allLoss <- function (cums, ker, x) {
    loss <- rep (0, 4)
    for (i in 2 : 4) {
        loss[i] <- tuckerValue (cums[[i]], ker[[i]], x)
    }
    return (loss)
}

interpolRoot <- function (ot, w) {
    cf <- solve (ot, w)
    pf <- polynomial (cf)
    rt <- solve (deriv (pf))
    rr <- Re (rt [which (abs (Im (rt)) < 1e-10)])
    rcz <- rr [which.min (abs (rr))]
    pcz <- predict (pf, rcz)
    return (list (rcz = rcz, pcz = pcz))
}
```

Code Segment 9 Even More Tucker Subroutines

```
tuckerValueMoments <- function (moms, wts, cums, x) {
    kers <- list ()
    for (i in 1:4) {
        kers <- c (kers, list (kernelMoments (4, i, moms)))
    }
    return (totLoss (wts, cums, kers, x))
}

oneCCDCycle <- function (wts, cums, ker, x, th, ot) {
    p <- nrow (ker[[2]])
    m <- nrow (cums[[2]])
    w <- rep(0, length (th))
    loss <- Inf
    for (j in 1 : m) {
        for (s in 1 : p) {
            for (i in 1 : length (th)) {
                xt <- x
                xt [j, s] <- x [j, s] + th [i]
                w [i] <- totLoss (wts, cums, ker, xt)
            }
            ipr <- interpolRoot (ot, w)
            pcz <- ipr $ pcz
            if ((pcz > loss) || (pcz < 0)) {
                next ()
            }
            loss <- pcz
            x [j, s] <- x [j, s] + (ipr $ rcz)
        }
    }
    return (x)
}

updateKernel <- function (wts, cums, ker, x, kerType) {
    if (kerType == "fixed") {
        return (ker)
    }
    if (kerType == "free") {
        return (tuckerFreeKernel (cums, ker, x))
    }
    if (kerType == "structured") {
        return (tuckerStructuredKernel (moms, wts, cums, x)
            )
    }
}
```

Code Segment 10 Tucker Auxilaries

```

arrKronecker <- function (x, fun="*") {
    nmat <- length (x)
    if (nmat == 0) {
        stop("empty argument in arrKronecker")
    }
    res <- x[[1]]
    if (length (x) == 1) {
        return (res)
    }
    for (i in 2 : nmat) {
        res <- kronecker (res, x[[i]], fun)
    }
    return (res)
}

repList <- function (x, n) {
  z <- list()
  for (i in 1 : n)
    z <- c (z, list (x))
  return (z)
}
```

Code Segment 11 Plotting Polynomials

```
plotPol <- function (x, means, title, moments = c(0, .1, 0,
     .03)) {
    m <- nrow (x)
    b <- as.vector (means - drop (x %*% moments))
    s <- seq (-.5, .5, length = 100)
    h <- seq (-.5, .5, length = 10)
    v <- matrix (0, m, 100)
    u <- matrix (0, m, 10)
    cols <- rainbow (m)
    for (i in 1 : m) {
        v[i, ] <- b[i] + x[i, 1] * s + x[i, 2] * (s ^ 2) +
            x[i, 3] * (s ^ 3) + x[i, 4] * (s ^ 4)
        u[i, ] <- b[i] + x[i, 1] * h + x[i, 2] * (h ^ 2) +
            x[i, 3] * (h ^ 3) + x[i, 4] * (h ^ 4)
   }
    ymax <- max (v)
    ymin <- min (v)
    plot (0, xlim = c (-.5, .5), ylim = c (ymin, ymax),
       main = title, xlab = "x", ylab = "P(x)", type = "n")
    for (i in 1 : m) {
        lines (s, v[i, ])
        points (cbind (h, u[i, ]), pch = as.character (i),
            col = cols[i])
    }
}
```

Appendix C. Figures

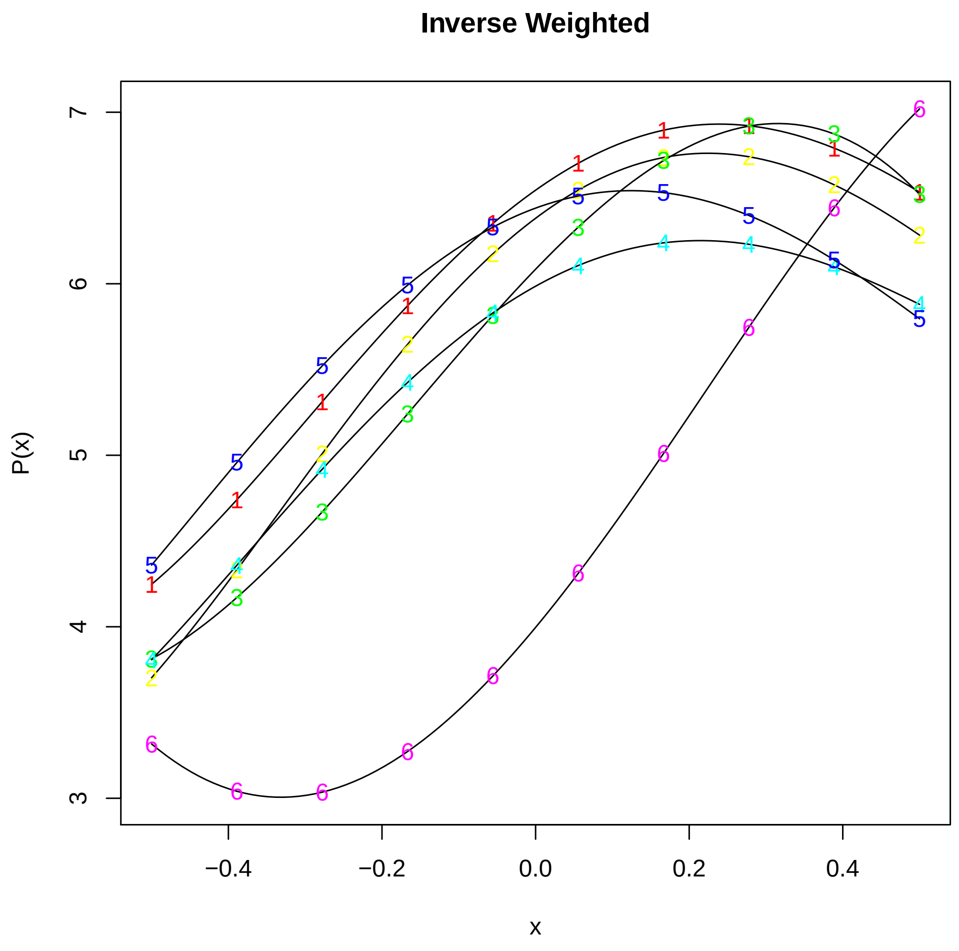
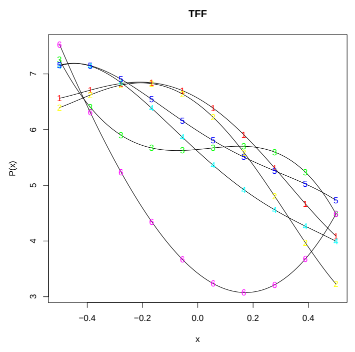
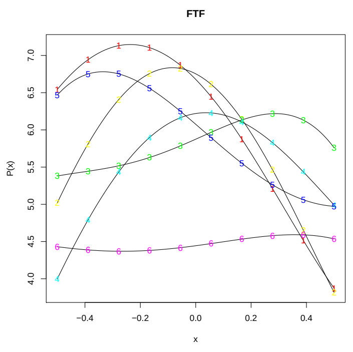


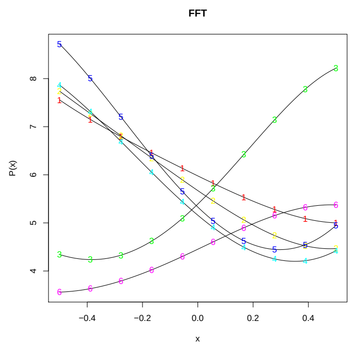
Figure 1. Polynomials with Inverse Weighting



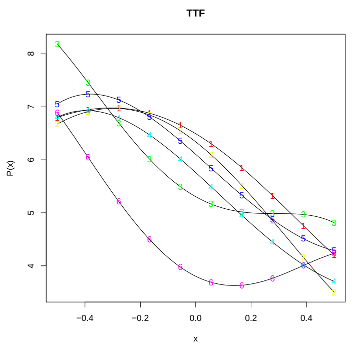
(A)



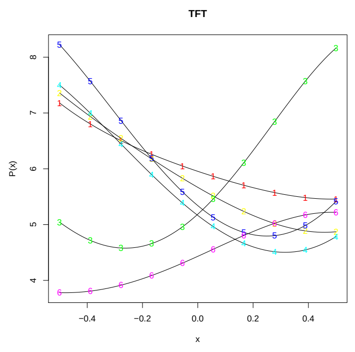
(B)



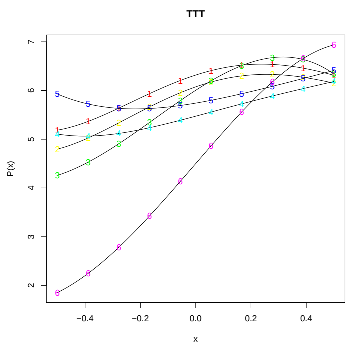
(C)



(D)



(E)



(F)

FIGURE 2. (a) TFF; (b) FTF; (c) FFT; (d) TTF; (e) TFT; and, (f) TTT.

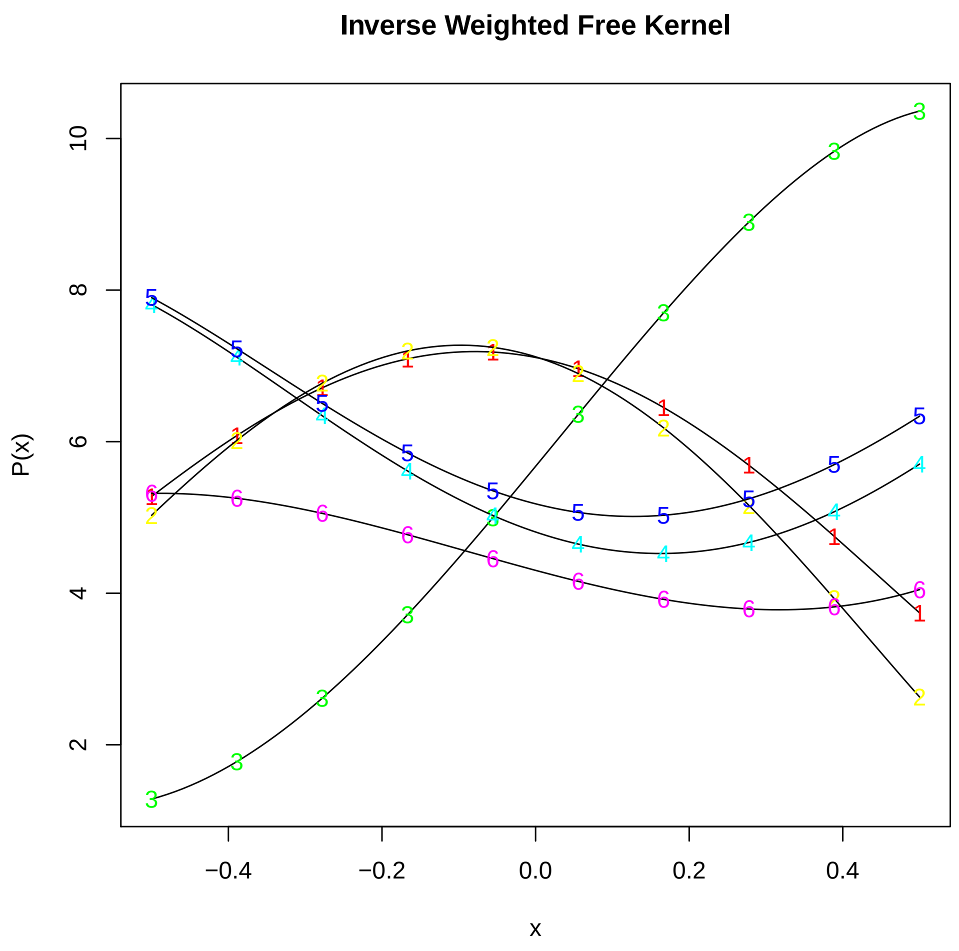


FIGURE 3. Polynomials with Inverse Weighting and Free Kernel

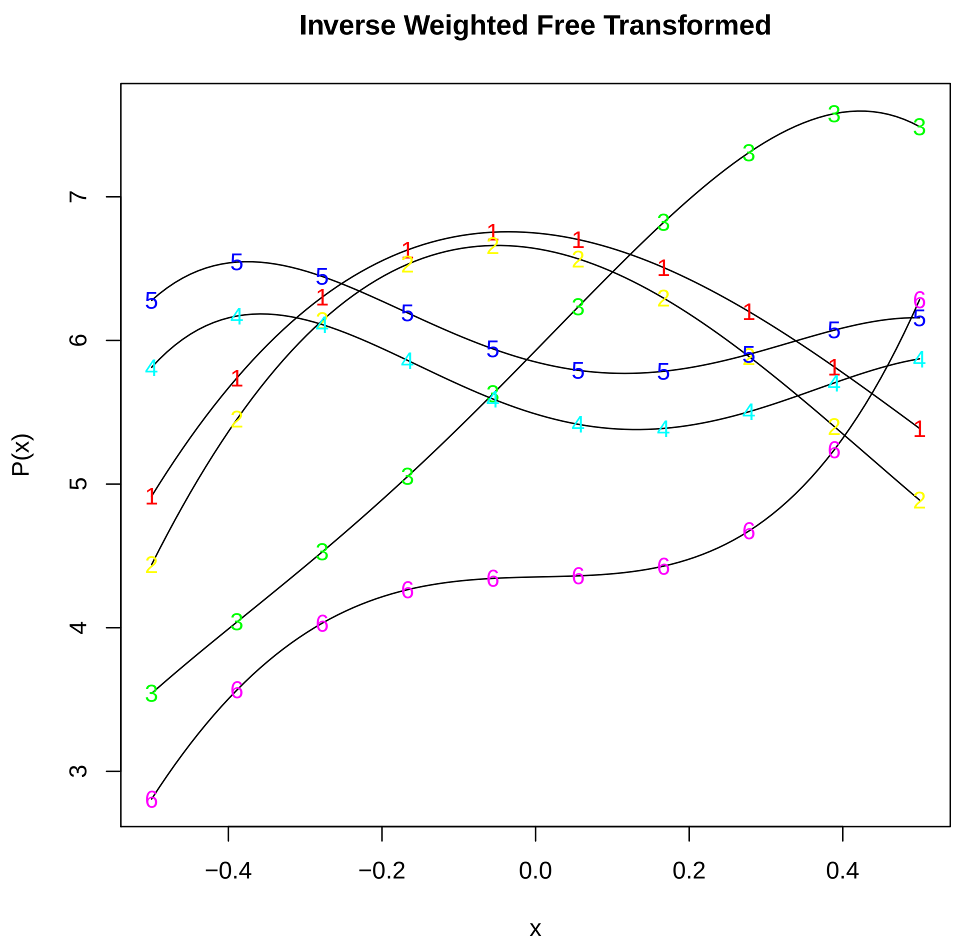


FIGURE 4. Polynomials with Inverse Weighting and Free Kernel, Transformed

Appendix D. Tables

	σ_2	σ_3	σ_4
TFF	0.56	1873.94	1417239.
FTF	17.55	12.19	10785.44
FFT	43.85	377.88	222.53
TTF	6.23	11.00	81626.65
TFT	25.00	262.07	228.77
TTT	15.68	108.57	85.62

Table 1. Loss Function Values for Six Solutions

Department of Statistics, University of California, Los Angeles, CA 90095-1554

E-mail address, Jan de Leeuw: `deleeuw@stat.ucla.edu`

URL, Jan de Leeuw: `http://gifi.stat.ucla.edu`