POLYNOMIAL SINGLE COMPONENT ANALYSIS USING MULTIVARIATE MOMENTS

JAN DE LEEUW

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

Date: Tuesday 30th July, 2013 — 17h 17min — Typeset in LUCIDA BRIGHT.

2000 *Mathematics Subject Classification.* 62H25.

Key words and phrases. Component Analysis, Polynomials, Array Decmposition.

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 polynomial single component analysis we assume there exist a random variable $\underline{x}$ such that

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

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

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

Closely related previous work on the polynomial factor analysis model, which differs from (1) because it includes unique factors, is in Mooijaart and Bentler [1986], who also discuss earlier work by Gibson and McDonald. Mooijaart and Bentler [1986] use higher order product moments, specifically third order moments, 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 nonlinear 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 single component model, we use higher-order statistics, specifically multivariate moments, and we use decomposition methods for multidimensional arrays to fit the models to sample moments. Thus, compared to Mooijaart and Bentler [1986], we develop the case of non-normal factors, provide algorithms and software to deal with moments of arbitrary order, and develop algorithms and software based on fitting multilinear

array decompositions. All software is written in R [R Development Core Team, 2012].

In a companion paper [De Leeuw, 2013] we have worked out the equations in terms of cumulants, instead of moments. Cumulants are useful for laying the groundwork for generalizations to approximations with multiple components, and to factor analysis models. In the special case of single-factor component analysis, however, working with moments is much simpler.

2. Product Moments

From the multilinearity property of moments for any r variables, not necessarily distinct, we have

$$\mathbf{E}(\mathcal{P}_{j_1}(\underline{x})\cdots\mathcal{P}_{j_r}(\underline{x}))=\sum_{p_1=0}^{s}\cdots\sum_{p_r=0}^{s}\beta_{j_1p_1}\cdots\beta_{j_rp_r}c_{p_1\cdots p_r},\tag{3}$$

where

$$c_{p_1\cdots p_r}=\mu_{p_1+\cdots+p_r},\tag{4}$$

and where we write μ_t for the t^{th} raw moment of $\underline{x}$.

The product moments of order r of the polynomials are a super-symmetric multilinear function of the elements of $B=\{\beta_{jp}\}$, for a given kernel $C=\{c_{p_1\cdots p_r}\}$. Supper-symmetry means that the array elements are invariant under all permutations of the subscript. Moreover the kernel must have the moment structure (4).

The s^r elements of the kernel array are simple linear functions of the first sr moments of $\underline{x}$. In fact we can write

$$C=\sum_{t=1}^{sr}\mu_t I_t,\tag{5}$$

where the I_t are super-symmetric orthogonal binary arrays with $\{I_t\}_{p_1\cdots p_r}=1$ only if $p_1+\cdots+p_r=t$.

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 the complicated moment inequalities we can use the parametrization $\mu_t = \int x^t dF(x)$ and think of the kernel as a function of the distribution function F. By Mulholland and Rogers [1958] we can limit ourselves, without loss of generality, to step functions with at most $sr + 1$ steps.

When dealing with multivariate moments it is convenient to introduce an additional variable $\underline{y}_0$ which is a.s. equal to one and a corresponding polynomial $\mathcal{P}_0$ which has $\beta_{0p} = 0$ for $p = 1, \cdots, s$. If we include $\mathcal{P}_0$ in the expectations (3), then the moment array includes all moments up to order r, i.e. also the moments of order $1, \cdots, r-1$.

3. Loss Function

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

We first compute an array of moments $A^{(r)} = \{a^{(r)}_{j_1 \cdots j_r}\}$ of order up to r from a given data matrix. This can be done with the code in segment 1.

INSERT CODE SEGMENT 1 ABOUT HERE

The data is a super-symmetric moment array of order $(m+1)^r$. The kernel is a super-symmetric array of order s^r, and B is an $(m+1) \times s$ matrix of loadings. The first row of B is zero, except for its first element.

We then define the least squares loss function by

$$\sigma(B,C) = \tag{6}$$

$$\sum_{j_1 \in J_1} \cdots \sum_{j_r \in J_r} w_{j_1 \cdots j_r} (a_{j_1 \cdots j_r} - \sum_{p_1=0}^{s} \cdots \sum_{p_r=0}^{s} \beta_{j_1 p_1} \cdots \beta_{j_r p_r} c_{p_1 \cdots p_r})^2.$$

and minimize it over both B and C.

The $w_{j_1 \cdots j_r}$ are given non-negative weights. They can be used to give lower or higher weights to the moments of various orders, or even to incorporate information about the standard errors of the moments.

Note that the loss function can be written in matrix form as

$$\sigma(B,C) = \tag{7}$$

$$\{\mathbf{vec}(A) - (\underbrace{B \otimes \cdot \otimes B}_{\text{r times}})\mathbf{vec}(C)\}' V \{\mathbf{vec}(A) - (\underbrace{B \otimes \cdot \otimes B}_{\text{r times}})\mathbf{vec}(C))\},$$

where $V = \mathbf{diag}(\mathbf{vec}(W))$. In actual computation and reporting we normalize the loss function to its root-mean-square form

$$\overline{\sigma}(B,C) = \sqrt{\frac{\sigma(B,C)}{\mathbf{tr}(V)}}. \tag{8}$$

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

3.1. **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 segment 2 can be used to compute moments up to order r, including $\mu_0 = 1$, of the standard normal distribution.

INSERT CODE SEGMENT 2 ABOUT HERE

The code in 3 can be used to put moments into the kernel array, using the formula (5).

INSERT CODE SEGMENT 3 ABOUT HERE

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

3.2. **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 formed of moments of the same variable. Thus we ignore the structure imposed by (5). We merely imposing super-symmetry on C. 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.

3.3. **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 (4). We minimize loss over B and over these vectors of moments. There are two variations. In the first we minimize over all vectors μ, without imposing the condition that the vector consists of moments of a single variable. In the second we minimize over all distributions F, which determine the moments. In fact, it usually makes more sense to minimize over a finite-dimensional convex set of distribution functions, for example the set of all convex combinations of B-splines of a given order with a given set of knots. In this case we can write

$$\mu_t = \sum_{b=1}^{B} \pi_b \int_{-\infty}^{+\infty} x^t dF_b(x). \tag{9}$$

Collecting the moments of the basis distributions in a matrix M allows us to write $\mu = M\pi$.

The code in segment 4 computes the basis I_t of indicators for the kernel used in formula (5). It is wasteful to compute and store the indicators, but for small examples it will be both feasible and computationally efficient.

INSERT CODE SEGMENT 4 ABOUT HERE

The code in segment 5 uses Neuman [1981, Proposition 3.2] to compute the moments of B-splines for any knot sequence to any order.

INSERT CODE SEGMENT 5 ABOUT HERE

4. ALGORITHMS

Various segments of code needed for our minimizations, written in R, to minimize (6) are given in De Leeuw [2008, 2012]; De Leeuw and Kukuyeva [2012]. We adopt and extend them to deal with our form of component analysis.

4.1. **Initial Solution.** Because of the clear possibility of local minima we want to start with a good initial configuration. For the kernel we always use the one based on moments of the standard normal. For the loadings we approximate the first moments $\overline{y}$ and second moments$S$ by an equation the form

$$\begin{bmatrix} 1 & \overline{y}' \\ \overline{y} & S \end{bmatrix} \approx \begin{bmatrix} 1 & 0 \\ b & B \end{bmatrix} \begin{bmatrix} 1 & c' \\ c & C \end{bmatrix} \begin{bmatrix} 1 & b' \\ 0 & B' \end{bmatrix}.$$

The product on the right works out to

$$\begin{bmatrix} 1 & b' + c'B' \\ b + Bc & bb' + Bcb' + bc'B' + BCB' \end{bmatrix}$$

Now choose $b = \overline{y} - Bc$, so that we fit the means exactly. Then the fit becomes

$$\begin{bmatrix} 1 & \overline{y}' \\ \overline{y} & \overline{y}\overline{y}' + B(C - cc')B', \end{bmatrix}$$

and we find B by using unweighted least squares to approximate the covariance $S-\overline{yy}'$ by $B(C-cc')B$. This is a simple eigenvalue-eigenvector problem.

4.2. **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 [Takane et al., 1977]. It is currently 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. The polynomial is minimized by using the `optimize()` function in R.

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.

4.3. **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].

Note that from (7) we see, using Moore-Penrose inverses, that the optimal kernel for fixed B is

$$\mathbf{vec}(C) = (\underbrace{B^+ \otimes \cdot \otimes B^+}_{\text{r times}})\mathbf{vec}(A). \tag{10}$$

4.4. **Structured Kernel.** From (5)

$$(\underbrace{B \otimes \cdot \otimes B}_{\text{r times}})\mathbf{vec}(C) = \sum_{t=1}^{sr} \mu_t (\underbrace{B \otimes \cdot \otimes B}_{\text{r times}})\mathbf{vec}(I_t) \tag{11}$$

Collecting the vectors $(\underbrace{B \otimes \cdot \otimes B}_{\text{r times}})\mathbf{vec}(I_t)$ in the columns of a matrix H allows us to write the loss function as

$$\sigma(B, \mu) = (\mathbf{vec}(A) - H\mu)'V(\mathbf{vec}(A) - H\mu), \tag{12}$$

while using (9) gives

$$\sigma(B, \pi) = (\mathbf{vec}(A) - HM\pi)'V(\mathbf{vec}(A) - HM\pi). \tag{13}$$

Thus the two variations for a structured kernel are to minimize (12) over all μ and to minimize (13) over all π in the unit simplex. Both are linear least squares problems, in the second case with simple linear inequality and equality constraints.

We solve the quadratic programming problem with the Frank-Wolfe linearization algorithm using the optimal step-size. The code is in segment 11.

INSERT CODE SEGMENT 11 ABOUT HERE

5. 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. On the basis of these results the authors eliminate item 6 from further analysis, and we will do the same thing. The data are generate by the following code.

```
library (psychotools)
data ("YouthGratitude")
gq6 <- as.matrix (YouthGratitude[,4:8])
```

We then load the required packages and source the required files.

```
require (abind)
require (apl)
require (polynom)

source ("tuckerCCD.R")
source ("kernelInitial.R")
source ("kernelUpdate.R")
source ("tuckerMore.R")
source ("tuckerFinal.R")
source ("tuckerKernel.R")
source ("tuckerAux.R")
source ("tuckerBasis.R")
source ("tuckerPlot.R")
source ("utilities.R")
source ("simplexLS.R")
```

For the runs we set the iteration precision, the decrease of successive loss function values, to 0.0001 and we set the maximum number of outer iterations to 500. The number of inner iterations, which are the coordinate descent cycles to fit B, is set at 5.

If we use a mixture of B-spline densities for the moments of the distribution of $\underline{x}$ we use equally spaced knots at the positive and negative integers from -6 to +6 and choose order 3 (i.e. we use piecewise quadratic polynomials differentiable at the knots).

The code below does 48 analyses, 12 for each of the four kernel types, using moments of order $r = 2, \cdots 4$ and polynomial degrees $s = 1, \cdots, 4$.

```
loss <- array (0, c(4, 3, 4))
itel <- array (0, c(4, 3, 4))

R <- 2 : 4
S <- 1 : 4
K <- 1 : 4

for (r in R) {
    for (s in S) {
        for (k in K) {
            kerType <- switch (k, "fixed", "free", "
               moment", "cdf")
            vname <- paste (kerType, r, s, sep = "")
            pname <- paste (vname, ".pdf", sep = "")
            assign (paste (kerType, r, s, sep = ""),
               tuckerCCD (gq6, r, s, kerType = kerType
               ))
            llist <- eval (as.name (vname))
            pdf (pname)
            plotPol (llist $ b, vname)
            dev.off ()
            loss [s, r - 1, k] <- llist $ tloss
            itel [s, r - 1, k] <- llist $ otel
        }
    }
}

dump (c("loss", "itel"), file = "results.R" )
```

5.1. **Results for Fixed Normal Kernel.**

6. Extensions and Elaborations

6.1. **Weights and Efficiency.** Adding weights $W = \{w_{j_1\cdots j_r}\}$ to (6) 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 (**??**).

6.2. **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.

6.3. **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.

6.4. **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 (**??**) and (**??**) continue to apply.

6.5. **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 (**??**) no longer applies, but we can continue to rely on (**??**). This more general description, together with work on more than one factor, will become a topic for further research.

6.6. **Convergence Accelleration.** In further work we plan to use the methods discussed in De Leeuw [2008] 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

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 Algorithms in Statistics. In H.H. Bock, W. Lenski, and M.M. Richter, editors, *Information Systems and Data Analysis*, pages 308–324, Berlin, 1994. Springer Verlag. URL http://www.stat.ucla.edu/~deleeuw/janspubs/1994/chapters/deleeuw_C_94c.pdf.

J. De Leeuw. Polynomial extrapolation to accelerate fixed point algorithms. Preprint Series 542, UCLA Department of Statistics, Los Angeles, CA, 2008. 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, 2008. 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. Single Factor Polynomial Component Analysis. Unpublished, 2013. URL http://www.stat.ucla.edu/~deleeuw/janspubs/2013/notes/deleeuw_U_13a.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.

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

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.

H.P. Mulholland and C.A. Rogers. Representation Theorems for Distribution Functions. *Proceedings of the London Mathematical Society*, 8:177–223, 1958.

E. Neuman. Moments and Fourier Transforms of B-splines. *Journal of Computational and Applied Mathematics*, 7:51–62, 1981.

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.

Y. Takane, F.W. Young, and J. De Leeuw. Nonmetric individual differences in multidimensional scaling: An alternating least squares method with optimal scaling features. *Psychometrika*, 42:7–67, 1977. URL http://www.stat.ucla.edu/~deleeuw/janspubs/1977/articles/takane_young_deleeuw_A_77.pdf.

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. CODE

Code Segment 1 Raw Moments up to Order p

```
rawMomentsUpToP <- function (x, p = 4) {
    n <- nrow (x)
    m <- ncol (x)
    if (p == 1) {
        return (c (1, apply (x, 2, mean)))
    }
    y <- array (0, rep (m + 1, p))
    for (i in 1 : n) {
        xi <- c (1, x[i, ])
        z <- xi
        for (s in 2:p) {
            z <- outer (z, xi)
        }
        y <- y + z
    }
    return (y / n)
}
```

Code Segment 2 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))
}

normalMoments <- function (n) {
    m <- rep (0, n)
    ind <- which(((1 : n) %% 2) == 0)
    m [ind] <- sapply (ind, normalEvenMoment)
    return (c (1, m))
}
```

Code Segment 3 Kernel from Moments

```
require("apl")

makeMomentKernel <- function (s, r, mu) {
    if (length (mu) != ((s * r) + 1)) {
        stop ("vector of moments wrong length")
    }
    c <- array(0, rep (s + 1, r))
    sr <- (s + 1) ^ r
    for (i in 1 : sr) {
        l <- aplEncode (i, rep (s + 1, r)) - 1
        k <- sum (l)
        c [i] <- mu [k + 1]
        }
    return (c)
}
```

Code Segment 4 Kernel Indicators

```
makeIndicatorBasisMatrix <- function (s, r) {
    ii <- repList (array (0, rep (s + 1, r)), (s * r)
       + 1)
    sr <- (s + 1) ^ r
    for (i in (1 : sr)) {
        l <- aplEncode (i, rep (s + 1, r)) - 1
        k <- sum (l)
        ii [[k + 1]] [i] <- 1
    }
    return (ii)
}
```

Code Segment 5 Moments of B-splines

```
makeBsplineMoments <- function (lup, t) {
    kup <- length (t) - 1
    m <- matrix (0, kup, lup + 1)
    for (l in (0 : lup)) {
        s1 <- (t [2] ^ (l + 1)) - (t [1] ^ (l + 1))
        s2 <- (l + 1) * (t [2] - t [1])
        m [1, l + 1] <- s1 / s2
        for (k in (2 : kup)) {
            ss <- 0
            for (j in (0 : l)) {
                s1 <- t [k + 1] ^ (l - j)
                s2 <- choose (k + j - 1, j)
                s3 <- m [k - 1, j + 1]
                ss <- ss + s1 * s2 * s3
            }
        m [k, l + 1] <- ss / choose (k + l, k)
        }
    }
    return (m [kup, ])
}

makeBsplineMomentMatrix <- function (lup, t, order) {
    k <- length (t) - order
    m <- matrix (0, k, lup + 1)
    for (i in (1 : k)) {
        m[i, ] <- makeBsplineMoments (lup, t [i : (i +
            order)])
    }
    return (m)
}
```

Code Segment 6 CCD Algorithm for Fixed/Free Kernel

```
tuckerCCD <- function (data, r, s,  w = array (1, dim (a)),
     kerType = "fixed", order = 3, knots = -6:6, inmax = 5,
    ineps = 1e-3, outmax = 500, outeps = 1e-4, verbose = 1)
    {
    a <- rawMomentsUpToP (data, r)
    vassign (i, m, p, q, k, values = kernelInitial (s, r, k
        , kerType, knots, order))
    vassign (b, otel, values = list (tuckerInitial (a, k),
        1))
    oloss <- tuckerValue (a, k, w, b)
    repeat {
        vassign (itel, bold, values = list (1, b))
        repeat {
            bnew <- oneCCDCycle (a, k, w, bold)
            chng <- max (abs (bnew - bold))
            if (verbose > 1)
                cat ("Ineration: ", formatC (itel, 3, 3),
                     "IneChange: ", formatC (chng, 6, 10, "
                        f"), "\n")
            if ( (chng  < ineps) || (itel == inmax))
                break ()
            vassign (bold, itel, values = list (bnew, itel
                + 1))
        }
        b <- bnew
        xloss <- tuckerValue (a, k, w, b)
        vassign (k, p, q, values = kernelUpdate (a, k, i, m
            , p, w, b, kerType))
        nloss <- tuckerValue (a, k, w, b)
        if (verbose >= 1)
            cat ("Iteration:", formatC (otel, 3, 3),
                 "OLoss: ", formatC (oloss, 6, 10, "f"),
                 "BLoss: ", formatC (xloss, 6, 10, "f"),
                 "Kloss: ", formatC (nloss, 6, 10, "f"), "\
                    n")
        if (((oloss - nloss) < outeps) || (otel == outmax))
            break ()
        vassign (oloss, otel, values = list (nloss, otel +
            1))
    }
    return (list(b = b, otel = otel, tloss = nloss, k = k,
        p = p, q = q))
}
```

Code Segment 7 Tucker Subroutines

```
tuckerValue <- function (a, k, w, b) {
    r <- length (dim (a))
    res <- as.vector (a) - arrKronecker (repList (b, r)) %
        *% as.vector (k)
    return (sqrt (sum (as.vector (w) * res * res) / sum (w)
        ))
}

tuckerInitial <- function (a, k) {
    r <- length (dim (a))
    s <- dim (a)[1]
    t <- dim (k) [1]
    h <- asub (a, c (repList (1 : s, 2), repList (1, r - 2)
        ))
    m <- (h - outer (h [1, ], h[1, ])) [-1, -1]
    e <- eigen (m)
    v <- asub (k, c (repList (1 : t, 2), repList (1, r - 2)
        ))
    p <- nrow (v) - 1
    u <- (v - outer (v [1, ], v[1, ])) [-1, -1]
    if (p == 1) {
        x <- as.matrix (e $ vectors [, 1]) %*% sqrt (e $
            values [1])
    } else {
        x <- e $ vectors [, 1 : p] %*% diag (sqrt (e $
            values [1 : p]))
    }
    bb <- x %*% solve (t (chol (u)))
    bv <- h [-1, 1] - bb %*% v [-1, 1]
    return (rbind (c (1, rep (0, t - 1)), cbind (bv, bb)))
}
```

Code Segment 8 More Tucker Subroutines

```
1  oneCCDCycle <- function (a, k, w, b) {
2      p <- dim (k) [1]
3      m <- dim (a) [1]
4      f <- function (x, j, s) tuckerValue (a, k, w,
           addElement (b, j, s, x))
5      loss <- tuckerValue (a, k, w, b)
6      for (j in 2 : m) {
7          for (s in 1 : p) {
8              oo <- optimize (f, c(-1, 1), j, s)
9              ls <- oo $ objective
10             ip <- oo $ minimum
11             if (ls > loss) {
12                 next ()
13             }
14             loss <- ls
15             b <- addElement (b, j, s, ip)
16         }
17     }
18     return (b)
19 }
20
21 addElement <- function (x, i, j, s) {
22      x [i, j] <- x[i, j] + s
23      return (x)
24 }
```

Code Segment 9 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()
  if (n == 0) {
    return (z)
  }
  for (i in 1 : n)
    z <- c (z, list (x))
  return (z)
}

vassign <- function(..., values, envir=parent.frame()) {
  vars <- as.character(substitute(...()))
  values <- rep(values, length.out=length(vars))
  for(i in seq_along(vars)) {
    assign(vars[[i]], values[[i]], envir)
  }
}
```

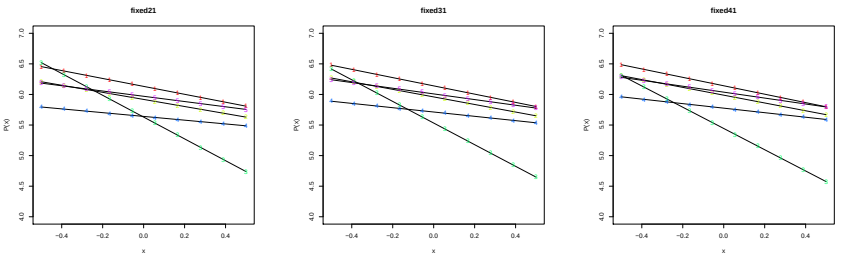
Code Segment 10 Plotting Polynomials

```
plotPol <- function (x, title) {
    m <- nrow (x)
    s <- seq (-.5, .5, length = 100)
    h <- seq (-.5, .5, length = 10)
    v <- matrix (0, m - 1, 100)
    u <- matrix (0, m - 1, 10)
    cols <- rainbow (m - 1)
    for (i in 1 : (m - 1)) {
        v[i, ] <- predict (polynomial (x[i + 1, ]), s)
        u[i, ] <- predict (polynomial (x[i + 1, ]), h)
   }
    ymax <- 7
    ymin <- 4
    plot (0, xlim = c (-.5, .5), ylim = c (ymin, ymax),
        main = title, xlab = "x", ylab = "P(x)", type = "n")
    for (i in 1 : (m - 1)) {
        lines (s, v[i, ])
        points (cbind (h, u[i, ]), pch = as.character (i),
            col = cols[i])
    }
}
```

Code Segment 11 Frank-Wolfe Algorithm

```
qpFrankWolfe <- function (x, y, w = rep (1, length (y)), b
    = rep (1, ncol (x)) / ncol (x), itmax = 100, eps = 1e-6,
     verbose = FALSE) {
    xwx <- crossprod (x, w * x)
    fold <- Inf
    itel <- 1
    repeat {
        res <- c (y - x %*% b)
        fnew<- sum (w * res ^ 2) / 2
        g <- - colSums (w * res * x)
        mg <- min (g)
        ig <- which.min (g)
        bnew <- rep (0, length (b))
        bnew [ig] <- 1
        d <- bnew - b
        tau <- sum (d * crossprod (x, w * res)) / sum (xwx
            * outer (d, d))
        tau <- min (1, max (0, tau))
        b <- b + tau * d
        if (verbose) {
                cat ("itel: ", formatC (itel, 3, 3),
                     "fold: ", formatC (fold, 10, 15, "f"),
                     "fnew: ", formatC (fnew, 10, 15, "f"),
                         "\n")
        }
        if (((fold - fnew) < eps) || (itel == itmax)) {
            break ()
        }
        itel <- itel + 1
        fold <- fnew
    }
    return (list (b = b, f = fnew))
}
```

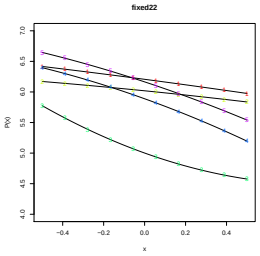
APPENDIX B. FIGURES



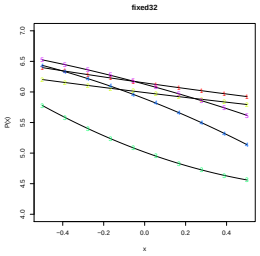
(A)

(B)

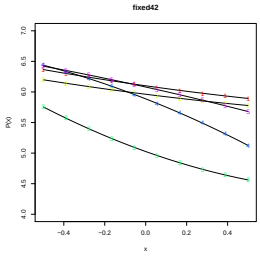
(C)



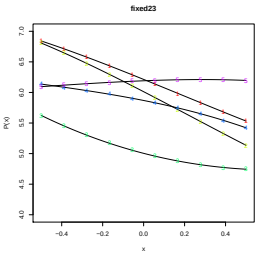
(D)



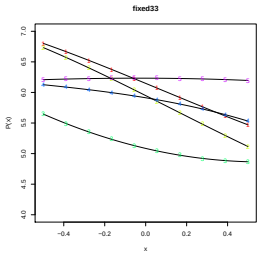
(E)



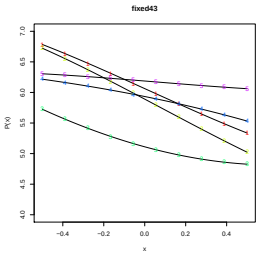
(F)



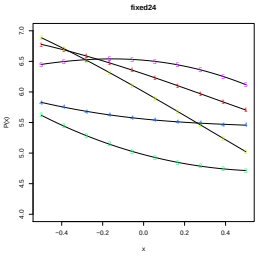
(G)



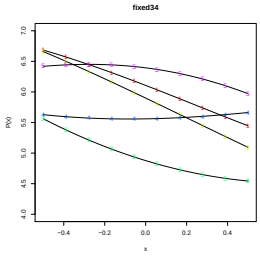
(H)



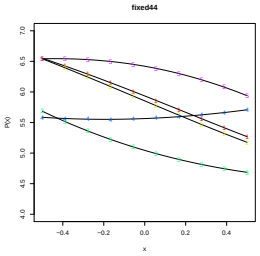
(I)



(J)

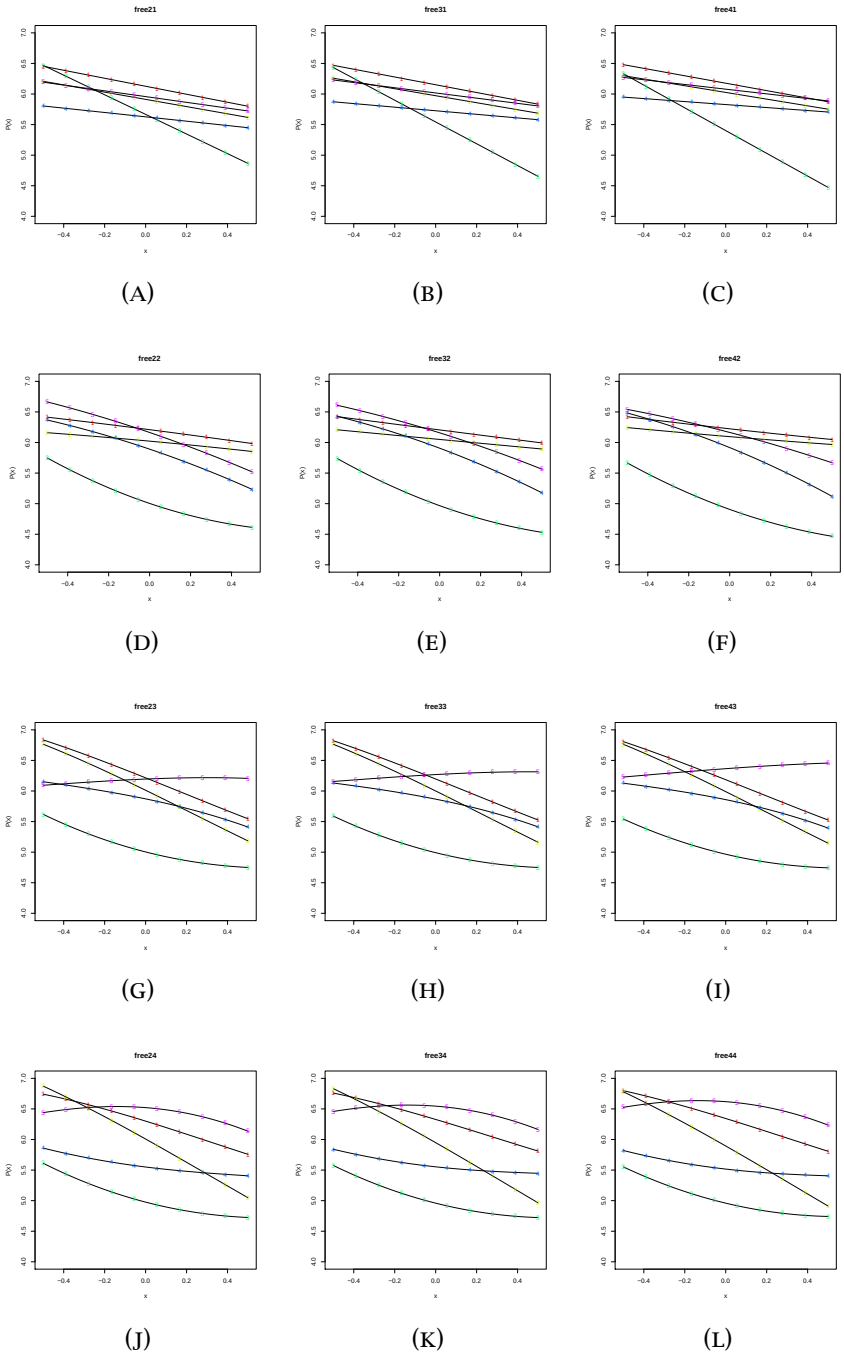


(K)



(L)

FIGURE 1. Twelve solutions with a fixed normal kernel. (a) s1r2; (b) s1r3; (c) s1r4; (d) s2r2; (e) s2r3; (f) s2r4; (g) s3r2; (h) s3r3; (i) s3r4; (j) s4r2; (k) s4r3; and, (l) s4r4.



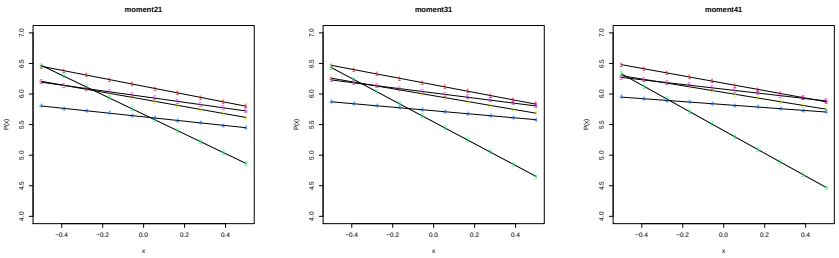
(A) (B) (C)

(D) (E) (F)

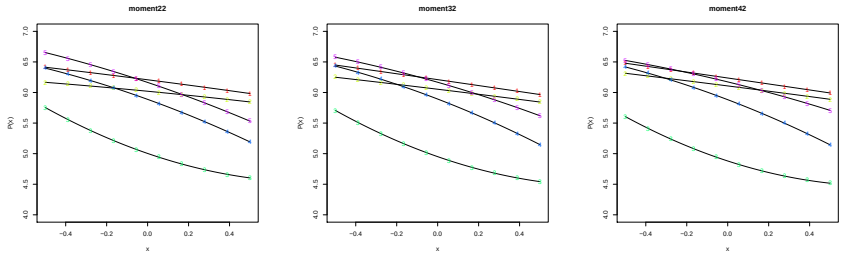
(G) (H) (I)

(J) (K) (L)

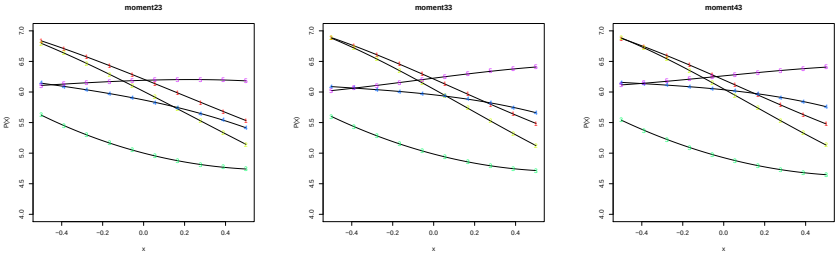
FIGURE 2. Twelve solutions with a free kernel. (a) s1r2; (b) s1r3; (c) s1r4; (d) s2r2; (e) s2r3; (f) s2r4; (g) s3r2; (h) s3r3; (i) s3r4; (j) s4r2; (k) s4r3; and, (l) s4r4.



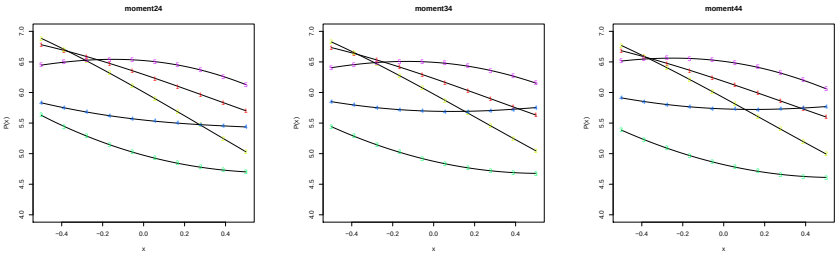
(A) (B) (C)



(D) (E) (F)

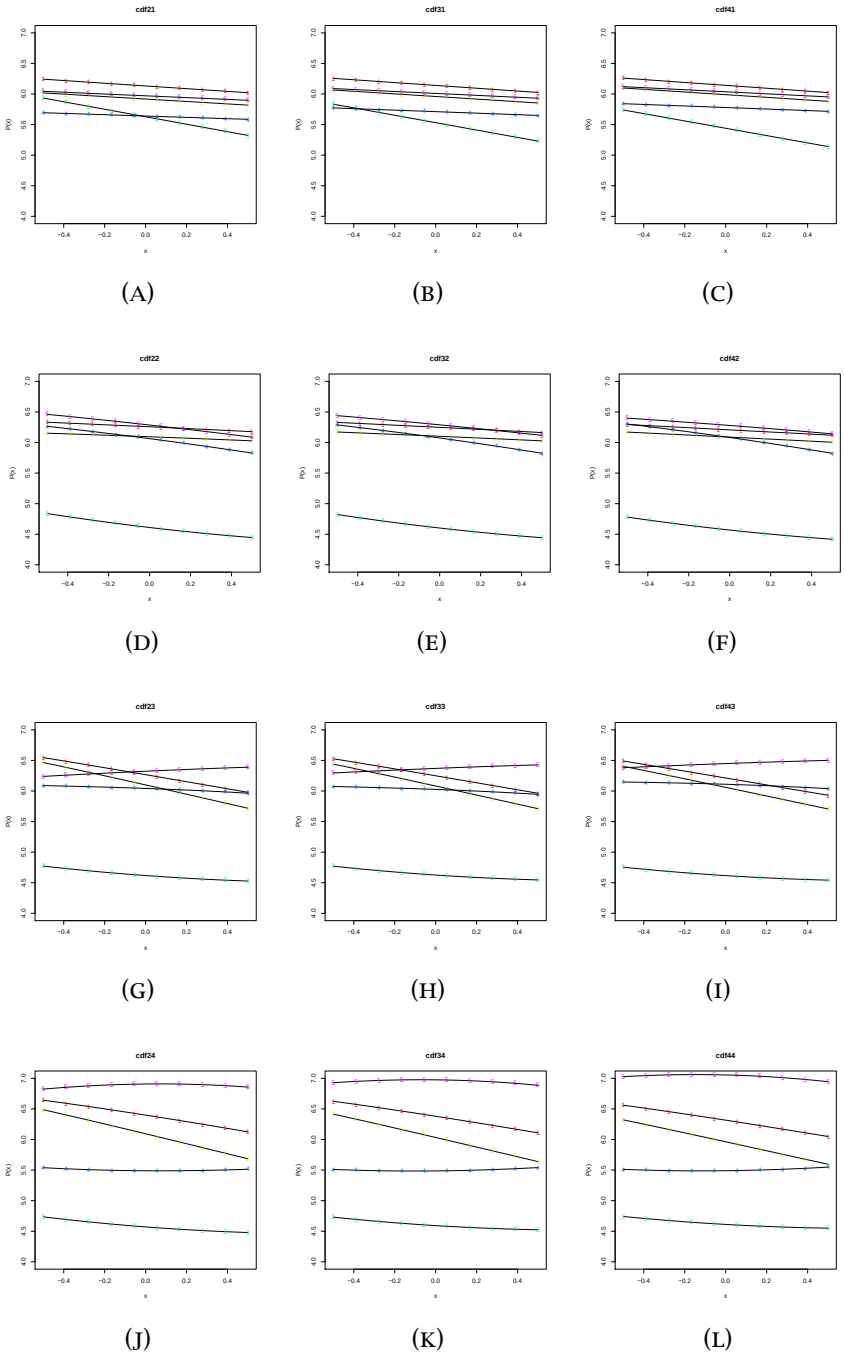


(G) (H) (I)



(J) (K) (L)

FIGURE 3. Twelve solutions with a moment kernel. (a) s1r2; (b) s1r3; (c) s1r4; (d) s2r2; (e) s2r3; (f) s2r4; (g) s3r2; (h) s3r3; (i) s3r4; (j) s4r2; (k) s4r3; and,**??** s4r4.



(A) (B) (C)

(D) (E) (F)

(G) (H) (I)

(J) (K) (L)

FIGURE 4. Twelve solutions with a CDF kernel. (a) s1r2; (b) s1r3; (c) s1r4; (d) s2r2; (e) s2r3; (f) s2r4; (g) s3r2; (h) s3r3; (i) s3r4; (j) s4r2; (k) s4r3; and, (l) s4r4.

APPENDIX C. TABLES

fixed	$r=2$	$r=3$	$r=4$
$s=1$	0.1950	1.636	11.56
$s=2$	0.1371	1.241	9.04
$s=3$	0.0664	0.823	6.50
$s=4$	0.0181	0.562	4.49
free	$r=2$	$r=3$	$r=4$
$s=1$	0.1968	1.630	11.41
$s=2$	0.1368	1.152	8.11
$s=3$	0.0658	0.529	3.58
$s=4$	0.0163	0.124	0.94
moment	$r=2$	$r=3$	$r=4$
$s=1$	0.1967	1.630	11.42
$s=2$	0.1369	1.175	8.31
$s=3$	0.0667	0.631	4.79
$s=4$	0.0183	0.377	3.22
cdf	$r=2$	$r=3$	$r=4$
$s=1$	0.1950	1.636	11.53
$s=2$	0.1376	1.191	8.55
$s=3$	0.0667	0.625	4.70
$s=4$	0.0199	0.353	3.34

TABLE 1. Minimum loss function values for different kernel types, polynomial degrees s and moment orders r.

fixed	$r=2$	$r=3$	$r=4$
$s=1$	6	14	45
$s=2$	37	76	500
$s=3$	8	85	500
$s=4$	10	431	500
free	$r=2$	$r=3$	$r=4$
$s=1$	31	134	500
$s=2$	30	77	500
$s=3$	23	105	500
$s=4$	30	304	500
moment	$r=2$	$r=3$	$r=4$
$s=1$	31	134	500
$s=2$	39	77	500
$s=3$	8	368	500
$s=4$	8	500	500
cdf	$r=2$	$r=3$	$r=4$
$s=1$	6	20	94
$s=2$	36	320	500
$s=3$	5	48	500
$s=4$	12	142	500

TABLE 2. Number of iterations for different kernel types, polynomial degrees s and moment orders r.

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