Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. doi: 10.18637/jss.v000.i00

Monotone Polynomials using **BUGS** and **Stan**

A.A. Manderson
University of Western Australia

E. Cripps
University of Western Australia

K. Murray
University of Western Australia

B.A. Turlach
University of Western Australia

Abstract

We present methods to fit monotone polynomials in a Bayesian framework, including implementations in the popular, readily available, modeling languages BUGS and Stan. The sum-of-squared polynomials parametrisation of monotone polynomials previously considered in the frequentist framework by Murray *et al.* (2016), is again considered here, due to its increased flexibility compared to other parametrisations. The specifics of our implementation are discussed, enabling the end user to adapt either implementation to their application. Testing was undertaken on real and simulated data sets, the output and diagnostics of which are presented. We demonstrate that Stan is preferable for high degree polynomials, with the component-wise nature of Gibbs sampling being potentially inappropriate for such highly connected models.

Keywords: convolution operation, monotone polynomials, BUGS, JAGS, OpenBUGS, Stan, WinBUGS.

1. Introduction

In many regression settings there is a need to place constraints on the shape of the regression function, for example it is known from some underlying physical or economic theory, that the regression curve is monotone. Furthermore, in some research problems it might be necessary to impose monotonicity constraints on the regression curve while the main interest is actually some derived quantity of the regression curve, e.g. the location of its inflection points (Firmin *et al.* 2011, 2012).

As monotone polynomials have many useful properties, e.g. that they are naturally strictly monotone and have easily identifiable derivatives, Murray *et al.* (2013) revisited the idea of using monotone polynomials and provided algorithms for fitting them to data based on the isotonic parameterisations proposed by Elphinstone (1983) and Hawkins's (1994) semi-definite programming algorithm. Murray *et al.* (2016) use a different isotonic parameterisation of monotone polynomials which leads to more efficient algorithms for fitting monotone polynomials to data and, more crucially, allows for fitting such polynomials to data when the function is only constrained to be monotone over a compact or semi-compact region.

All of these approaches for fitting monotone polynomials to data were developed in a frequentist framework, but to our knowledge, no work has been done in a Bayesian framework beyond the linear case. While it is possible to implement monotone polynomials in a Bayesian framework using tailor-made Markov chain Monte Carlo (MCMC) algorithms, we discuss here how the isotonic parameterisations studied in Murray *et al.* (2016) can be used to implement monotone polynomials in probabilistic programming languages such as BUGS (Lunn *et al.* 2000, 2009, 2012) and Stan (Carpenter *et al.* 2017).

Specifically, we consider inference about the regression parameters β_j, and the regression function, in the linear polynomial regression model

$$Y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_q x^q + \varepsilon, \tag{1}$$

where q is the degree of the polynomial, assumed to be monotone, and ε is an error term that is assumed to follow a Gaussian distribution with mean 0 and variance σ_ε^2. Denoting the vector of regression parameters by $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_q)^T$, the usual parameterisation for the polynomial regression function is

$$p(x) = p(x; \boldsymbol{\beta}) = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_q x^q. \tag{2}$$

However, this parameterisation is not convenient to use if there are monotonicity constraints on the polynomial over a set $\mathcal{R} \subseteq \mathbb{R}$. Previously published work (Elphinstone 1983; Hawkins 1994; Heinzmann 2008; Murray *et al.* 2013) on monotone polynomials only considered the case $\mathcal{R} = (-\infty, \infty)$. Murray *et al.* (2016) proposed to use another isotonic parameterisation which allows, in addition to $(-\infty, \infty)$, the specification of more general regions, namely semi-compact intervals $[a, \infty)$ and compact intervals $[a, b]$ for finite $a, b \in \mathbb{R}$, on which the fitted polynomial satisfies a monotonicity constraint.

The structure of this paper is as follows: In Section 2 we specify the mathematical framework used to ensure monotonicity. The specific implementation in BUGS and Stan is presented in Section 3, with the complete code provided in the appendices. Numerical experiments were carried out on simulated and real data sets and a summary of the results are given in Section 4, along with a discussion of the diagnostic outputs. Concluding remarks and discussion of future work are provided in Section 5.

2. Isotonic parameterisation

An alternative parameterisation to (2) which is suitable for monotonic polynomials, is given by an isotonic parameterisation which writes the polynomial in the form

$$p(x) = \beta_0 + \alpha \int_0^x \check{p}(u)\, du, \tag{3}$$

where $\check{p}(u)$ is required to be non-negative on $\mathcal{R}$. Non-negative polynomials are well studied in the literature (see for example Marshall 2008, and references therein). From Equation (3) it follows immediately that the first derivative of the polynomial $p(x)$ is $p'(x) = \alpha\check{p}(x)$, which ensures the monotonicity of the polynomial in either an increasing or decreasing manner, depending on whether α is positive or negative.

The isotonic parameterisations considered by Murray *et al.* (2013), based on and extending work by Elphinstone (1983), wrote $\check{p}(x)$ as a product of quadratic polynomials, where each of these quadratics had either conjugate complex roots or a real root of multiplicity 2. These parameterisations of non-negative polynomials are not suitable for computations in a Bayesian framework, as it is difficult to specify suitable priors for the location of the roots of the quadratics terms, and to design efficient MCMC schemes. Furthermore, these parameterisations cannot be readily extended to more general forms of $\mathcal{R}$.

Based on Murray *et al.* (2016), we investigate another isotonic parameterisation for $\check{p}(u)$ which is based on the following proposition.

Proposition 2.1 *A polynomial $\check{p}(x)$ of degree $q-1 \geq 0$ is non-negative*

1. *on $\mathcal{R} = (-\infty, \infty)$ if and only if $q = 2K+1$ and it can be written as the sum of two squared polynomials*
$$\check{p}(x) = p_1(x)^2 + p_2(x)^2, \quad \forall x \in \mathbb{R} \tag{4}$$
where $p_1(x)$ and $p_2(x)$ are polynomials whose degrees are at most K.
2. *on $\mathcal{R} = [a, \infty)$ if and only if it can be written as*
$$\check{p}(x) = p_1(x)^2 + (x-a)p_2(x)^2, \quad \forall x \in \mathbb{R} \tag{5}$$
where,
   (a) *if $q = 2K+1$, $p_1(x)$ and $p_2(x)$ are polynomials whose degrees are at most K and $K-1$, respectively, and,*
   (b) *if $q = 2K$, both degrees are at most $K-1$.*
3. *on $\mathcal{R} = [a, b]$ if and only if it can be written as*
   (a) *if $q = 2K+1$:*
$$\check{p}(x) = p_1(x)^2 + (x-a)(b-x)p_2(x)^2, \quad \forall x \in \mathbb{R} \tag{6}$$
   where $p_1(x)$ and $p_2(x)$ are polynomials whose degrees are at most K and $K-1$, respectively.
   (b) *if $q = 2K$:*
$$\check{p}(x) = (b-x)p_1(x)^2 + (x-a)p_2(x)^2, \quad \forall x \in \mathbb{R} \tag{7}$$
   where $p_1(x)$ and $p_2(x)$ are polynomials with their degree at most $K-1$.

For the purpose of Proposition 2.1, $p_2(x) \equiv 0$ if $p_2(x)$ has a negative degree (cases 2(a) and 3(a) when $q = 1$ and $K = 0$). This proposition can be proved using the theory of Tchebycheff systems (Karlin and Studden 1966) or the theory of canonical moments (Dette and Studden 1997). A proof that does not utilise such deep mathematical theories can be found in Brickman and Steinberg (1962). The latter authors also point out that these parameterisations of monotone polynomials are not unique; a point that is further discussed in Section 4.

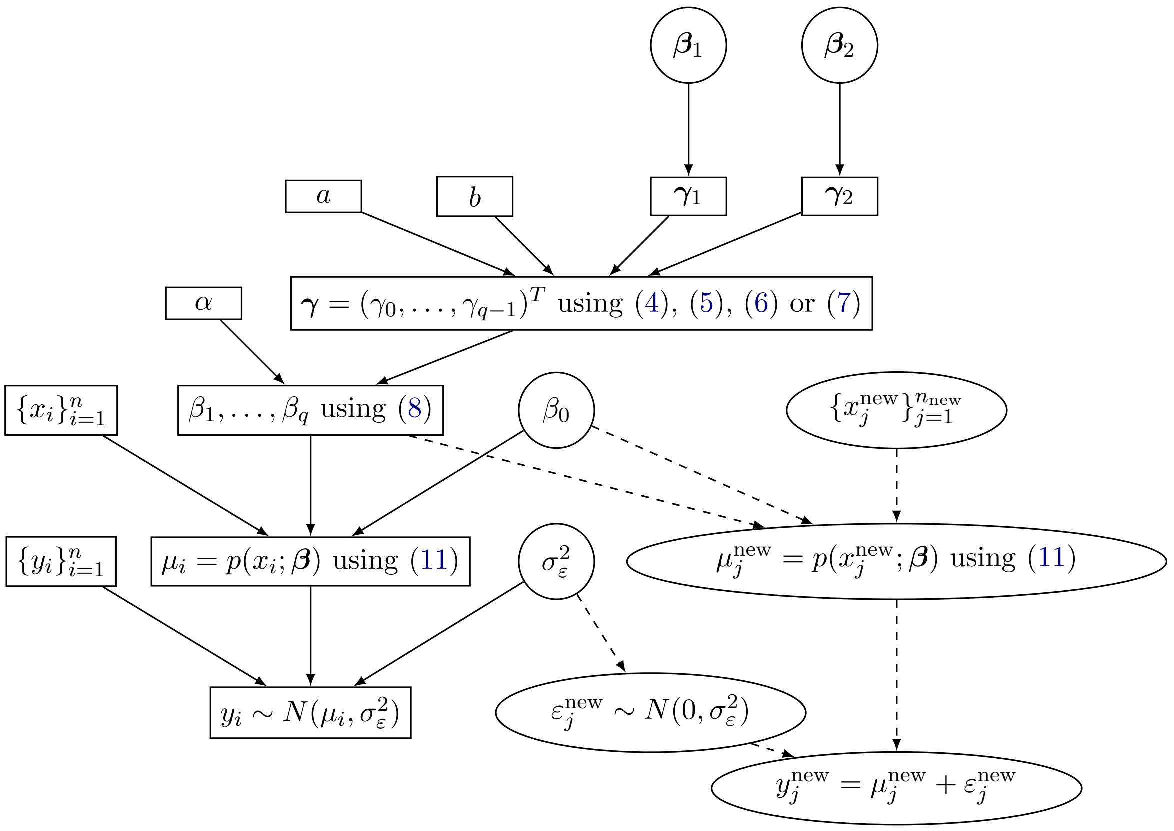
$\boldsymbol{\beta}_1$
$\boldsymbol{\beta}_2$
a
b
$\boldsymbol{\gamma}_1$
$\boldsymbol{\gamma}_2$
α
$\boldsymbol{\gamma} = (\gamma_0, \dots, \gamma_{q-1})^T$ using (4), (5), (6) or (7)
$\{x_i\}_{i=1}^n$
$\beta_1, \dots, \beta_q$ using (8)
β_0
$\{x_j^{\text{new}}\}_{j=1}^{n_{\text{new}}}$
$\{y_i\}_{i=1}^n$
$\mu_i = p(x_i; \boldsymbol{\beta})$ using (11)
σ_ε^2
$\mu_j^{\text{new}} = p(x_j^{\text{new}}; \boldsymbol{\beta})$ using (11)
$y_i \sim N(\mu_i, \sigma_\varepsilon^2)$
$\varepsilon_j^{\text{new}} \sim N(0, \sigma_\varepsilon^2)$
$y_j^{\text{new}} = \mu_j^{\text{new}} + \varepsilon_j^{\text{new}}$

Figure 1: Directed acyclic graph of the fitted model. Quantities in circles are parameters of the model for which priors can be specified. Quantities in rectangles are either observed data or derived quantities. Quantities in ellipses are optional and would be used to calculate credibility intervals and/or prediction intervals.

We use Proposition 2.1, by fixing α in (3) to be either -1 or 1, depending on whether the polynomial should be monotone decreasing or monotone increasing, over $\mathcal{R}$. We denote the coefficients of $p_1(x)$ and $p_2(x)$ in (4)–(7) by $\boldsymbol{\beta}_1 = (\beta_{01}, \beta_{11}, \dots, \beta_{q_1 1})^T$ and $\boldsymbol{\beta}_2 = (\beta_{02}, \beta_{12}, \dots, \beta_{q_2 2})^T$, with $q_1, q_2 \in \{K-1, K\}$. The coefficients $\boldsymbol{\gamma}_1$ and $\boldsymbol{\gamma}_2$ of the polynomials $p_1(x)^2$ and $p_2(x)^2$, can be calculated by convolution of $\boldsymbol{\beta}_1$ with itself and by convolution of $\boldsymbol{\beta}_2$ with itself. By adding the corresponding entries in $\boldsymbol{\gamma}_1$ and $\boldsymbol{\gamma}_2$ we can determine the coefficients $\boldsymbol{\gamma} = (\gamma_0, \dots, \gamma_{q-1})^T$ of the polynomial

$$\check{p}(x) = \gamma_0 + \gamma_1 x + \cdots + \gamma_{q-1} x^{q-1}.$$

From $\boldsymbol{\gamma}$ we can readily calculate $\boldsymbol{\beta}$ as

$$\boldsymbol{\beta} = \left(\beta_0, \alpha\gamma_0, \alpha\frac{\gamma_1}{2}, \dots, \alpha\frac{\gamma_{q-1}}{q} \right)^T. \tag{8}$$

Once $\boldsymbol{\beta}$ is determined we can easily evaluate $p(x)$ for arbitrary values of x using Horner's scheme (see, among others, Fausett 2003) for numerical stability, see Equation (11) in Section 3.4. Figure 1 depicts the model that we are fitting as a directed acyclic graph and illustrates the flow of calculations just discussed. Details of the implementation of these calculations in either BUGS or Stan are discussed in more detail in the next section.

3. Implementation

This section discusses some aspects of implementing the fitting of monotone polynomials in a Bayesian Framework using BUGS and Stan; full listings of the code are given in Appendices A and B. For the BUGS language, the five cases listed in Proposition 2.1 were implemented in separate files while for Stan, due to its better program flow commands, a singe file suffices. While several other programming languages can be interfaced to BUGS and Stan, we focus here on their interfaces to R, and describe in our online supplementary material appropriate R scripts.

3.1. Supplying the necessary data

BUGS

The code for the five BUGS files was designed such that they all have the same user interface. That is, the user will either have to specify the same information in the data section when running this code directly in BUGS, or pass the same data from R to BUGS when using any of the packages that interface R with WinBUGS (Sturtz *et al.* 2005), OpenBUGS (Thomas *et al.* 2006; Yan and Prates 2013) or JAGS (Plummer 2016). Specifically, the observed x and y values have to be passed to vectors called `x` and `y`, and the number of observed data points have to be passed to `N`. Given q and K are such that $q = 2K$ or $2K + 1$, and q is the degree of the polynomial, then q and K should be passed to `q` and `K`.[1]. Furthermore, $\alpha \in \{-1, 1\}$ should be passed to `alpha`, and, if required (i.e. when $\mathcal{R}$ is compact or semi-compact), a and/or b should be passed to `a` and `b`. Finally, to obtain posterior means and/or predictions at specific points of the abscissa, these should be passed to an object named `xnew` and the number of such points to `Nnew`. This covers the discussion of all the observed data quantities in Figure 1. How to specify starting values for the parameter of the models will be discussed below when we propose suitable priors for these parameters in Section 3.5.

Stan

The Stan implementation requires the same input, and uses the same names for the objects, as the BUGS implementations; see the data section in lines 38–73 in Appendix B. However, for the Stan implementation values for a and b *always* have to be passed to `a` and `b`, respectively, although they can be specified as $-\infty$ or ∞. Our Stan implementation also requires the user to specify the form of the region over which the polynomial should be monotone, whether it be compact, semi-compact or unconstrained. This is done through an additional input parameter named `operation_mode`[2].

3.2. Determining γ_1 and γ_2 by convolution

The coefficients of, say, the polynomial $p_1(x)^2$, i.e. the vector $\boldsymbol{\gamma}_1 = (\gamma_{01}, \gamma_{01}, \dots, \gamma_{2q_1,1})^T$, can be determined from the coefficients of the polynomial $p_1(x)$, the vector $\boldsymbol{\beta}_1 = (\beta_{01}, \beta_{11}, \dots, \beta_{q_1 1})^T$,

[1]The JAGS implementations also define, for convenience, a node called `Kp1` holding the value $K+1$; see the last lines of Appendices A.1–A.5. As will become clear from the discussion in Section 3.2 this is not necessary for the implementations in WinBUGS or OpenBUGS.

[2]The operation modes are 1, 2, 3 and 4, which correspond to the regions $(-\infty, \infty)$, $(-\infty, b]$, $[a, \infty)$ and $[a, b]$ respectively.

by the following convolution formula:

$$\gamma_{j1} = \sum_{i=\max(0,j-q_1)}^{\min(q_1,j)} \beta_{i,1}\,\beta_{j-i,1} \qquad j = 0, 1, \ldots, 2q_1, \tag{9}$$

with $\boldsymbol{\gamma}_2$ being determined in the same manner. Since BUGS and Stan both use one-based indexing for vectors, matrices and arrays, Equation (9) needs some re-writing as it is stated for a language that uses zero-based indexing.

BUGS

Assume $q_1 = K$ and the $K + 1$ entries of the vector $\boldsymbol{\beta}_1$ are stored in a vector called `b1` of length $K + 1$. Then the $2q_1 + 1 = 2K + 1$ entries of the vector $\boldsymbol{\gamma}_1$, stored in the object `g1`, can be determined as follows:

$$\texttt{g1}[j] = \sum_{i=\max(0,j-(K+1))+1}^{\min(K+1,j)} \texttt{b1}[i] * \texttt{b1}[j-i+1] \qquad j = 1, 2, \ldots, 2K+1. \tag{10}$$

Essentially, each entry of $\boldsymbol{\gamma}_1$ is the inner product of two specific slices of the vector $\boldsymbol{\beta}_1$, with the order of the entries in the second slice being reversed. Unfortunately, currently neither BUGS nor Stan seem to have a function that reverses a (sub-)vector, and neither language seems to support sub-setting by range indexes that are in reverse order (i.e. unlike R, neither language supports sub-setting of the form `b1[3:1]`).

Thus, to implement Equation (10) in BUGS we first create a matrix, say `bnrev`, to store the entries of $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$ in reverse order. This matrix is created from the matrix `bn`[3] which, for all five files listed in Appendix A, is a $(K + 1) \times 2$ matrix containing the entries of $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$. The code for this reversing is straightforward:

```
for(j in 1:(K+1)){
  for(k in 1:2){
    bnrev[j,k] <- bn[Kp1-j+1,k]
  }
}
```

Subsequently, we can implement Equation (9), for both $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$, via the following code[4]:

```
for(j in 1:q){
  g1[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),1],
                  bnrev[(max(0,Kp1-j)+1):(min(q-j,K)+1),1])
  g2[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),2],
                  bnrev[(max(0,Kp1-j)+1):(min(q-j,K)+1),2])
}
```

Unfortunately, JAGS is the only implementation of the BUGS modelling language in which it is possible to calculate the indices with which to subset a vector or matrix in the manner

[3]Initial values can be specified for this object, as further discussed in Section 3.5.

[4]See lines 32–37 in Appendices A.1 and A.4, and lines 37–42 in Appendix A.5. Due to the decision that `bn` should always be a `Kp1` $\times$ 2 matrix, the sub-setting of `bnrev` in lines 32–27 of Appendices A.2 and A.3 is slightly different.

above. WinBUGS and OpenBUGS are not able to compile and execute the above `for` loop. To run the code in Appendix A in WinBUGS or OpenBUGS, the `for` loop has to be explicitly unrolled. The following R snippet produces the necessary code that needs to be substituted for this `for` loop, and is illustrated for an odd degree polynomial with $q = 3$ and $K = 1$:

```
K <- 1
## q <- 2*K
q <- 2*K+1
Kp1 <- K+1
even.deg <- q%%2 == 0

for(j in 1:q){
  print(bquote(g1[.(as.double(j))] <-
                 inprod(bn[.(max(0,j-Kp1)+1):.(min(Kp1,j)),1],
                        bnrev[.(max(0,Kp1-j)+1):.(min(q+even.deg-j,K)+1),1])))
  print(bquote(g2[.(as.double(j))] <-
                 inprod(bn[.(max(0,j-Kp1)+1):.(min(Kp1,j)),2],
                        bnrev[.(max(0,Kp1-j)+1):.(min(q+even.deg-j,K)+1),2])))
}

## g1[1] <- inprod(bn[1:1, 1], bnrev[2:2, 1])
## g2[1] <- inprod(bn[1:1, 2], bnrev[2:2, 2])
## g1[2] <- inprod(bn[1:2, 1], bnrev[1:2, 1])
## g2[2] <- inprod(bn[1:2, 2], bnrev[1:2, 2])
## g1[3] <- inprod(bn[2:2, 1], bnrev[1:1, 1])
## g2[3] <- inprod(bn[2:2, 2], bnrev[1:1, 2])
```

To run this snippet, the user has to specify K and decide whether the polynomial should have an even degree ($q = 2K$) or an odd degree ($q = 2K + 1$) in the initial three lines of the code; the remainder of the code is generic. The output of this snippet has to replace the corresponding part in the files at lines 32–37 in Appendices A.1, A.2, A.3 and A.4, and lines 37–42 in Appendix A.5.

Finally, once the coefficients of $\boldsymbol{\gamma}_1$ and $\boldsymbol{\gamma}_2$ are calculated from $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$ the additional convolutions, if required, of $\boldsymbol{\gamma}_2$ with $(-a, 1)$ (Equations (5) and (7)) and $\boldsymbol{\gamma}_1$ with $(b, -1)$ (Equation (7)) can be implemented in a straightforward manner; see lines 27–30 in Appendices A.2, A.3 and A.4. To implement Equation (6), the code in Appendix A.5 calculates first the convolution of $\boldsymbol{\gamma}_2$ with $(b, -1)$ and then the convolution of this result with $(-a, 1)$; see lines 27–35 in Appendix A.5. This code also determines at the same time the vector $\boldsymbol{\gamma}$, i.e. the coefficients of the polynomial $\check{p}(x)$, and stores them in the object called `gamma`.

Stan

In Stan it is more convenient to write a general convolution function in the `function definition` block of the code. Our implementation follows the ideas used for the BUGS implementation and is given in lines 2–23 in Appendix B.

Due to the more rigid structure of Stan, in which the order of statements matters, the remainder of the calculations described above for BUGS have to be distributed over several blocks.

The `transformed data` block (lines 75–125 in Appendix B) defines variables `p1_length`, `p2_length`, `gamma_1_length` and `gamma_2_length` that contain the length of the vectors $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, $\boldsymbol{\gamma}_1$ and $\boldsymbol{\gamma}_2$, respectively. In the case that additional convolutions as per equations (5)–(7) are necessary, the variable `gamma_2_temp_length` contains the length of the vector of coefficients of $(x-a)p_2(x)^2$ or $(x-a)(b-x)p_2(x)^2$, while the variable `gamma_1_temp_length` contains the length of the vector of coefficients of $(b-x)p_1(x)^2$. The exact values of these variables are calculated according to the inputs `q`, `K` and `operation_mode` in lines 85–116 of Appendix B, in line with Proposition 2.1. In this block we also define two vectors of length 2 named `upper_bound_vector` and `lower_bound_vector` which contain the coefficient of the polynomials $(b-x)$ and $(x-a)$ (lines 118–124 in Appendix B).

The actual space for the vectors $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$ is allocated in the `parameters` block; see lines 127–133 of Appendix B. Other parameters defined in this block, as per Figure 1, are `sd_y` for σ_ε and `beta_zero` for β_0.

Finally, the space for the vectors $\boldsymbol{\gamma}_1$, $\boldsymbol{\gamma}_2$, $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, as well as space for some temporary vectors in the case that additional convolutions as per equations (5)–(7) are necessary, is allocated in the `transformed parameters` block; see lines 135–195 in Appendix B. After initialising some of these vectors to zero (lines 144–148) we calculate $\boldsymbol{\gamma}_1$ by convoluting $\boldsymbol{\beta}_1$ with itself in line 151, and likewise for $\boldsymbol{\gamma}_2$ and $\boldsymbol{\beta}_2$ in line 152. In lines 157–186, we calculate $\boldsymbol{\gamma}$ from $\boldsymbol{\gamma}_1$ and $\boldsymbol{\gamma}_2$ while performing additional convolutions as required per Proposition 2.1.

3.3. Determining $\boldsymbol{\beta}$

Once the vector $\boldsymbol{\gamma}$ is calculated it is easy to determine the components of $\boldsymbol{\beta}$ and store them in an object called `beta`[5]. The corresponding BUGS code is:

```
for(j in 1:q){
  beta[j+1] <- alpha * gamma[j]/j
}
beta[1] <- beta0
```

While the Stan code is (lines 188–191 of Appendix B):

```
beta_final[1] = beta_zero;
for(i in 1:q) {
  beta_final[i+1] = alpha * gamma[i] / i;
}
```

3.4. Implementing Horner's scheme for evaluating polynomials

Horner's scheme is a numerically stable method to efficiently evaluate polynomials, by re-arranging the calculations in the following manner:

$$\begin{aligned} p(x) &= \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_q x^q = \beta_q x^q + \beta_{q-1}x^{q-1} + \cdots + \beta_1 x + \beta_0 \\ &= (((\cdots(((\beta_q x + \beta_{q-1})x + \beta_{q-2})x + \beta_{q-3})\cdots)x + \beta_2)x + \beta_1)x + \beta_0. \end{aligned} \tag{11}$$

[5]The object `beta0` (in BUGS, `beta_zero` in Stan) is another parameter for which initial values can be specified and it is further discussed in Section 3.5.

BUGS

A difficulty when implementing this scheme in BUGS is that the language does not allow logical nodes to appear multiple times on the left hand side of an assignment. Hence, we have to create a matrix to hold all the intermediate results for evaluating $p(x)$ at the observed x values for the current value of $\boldsymbol{\beta}$. The following code to implement Horner's scheme can be found in lines 2–10 of all files in Appendices A.1–A.5:

```
for(i in 1:N){
  y[i] ~ dnorm(mu[i], tauy)

  horner[i,1] <- beta[q+1]
  for(k in 1:q){
    horner[i,k+1] <- horner[i,k]*x[i] + beta[q+1-k]
  }
  mu[i] <- horner[i,q+1]
}
```

Note that we also define the model for the y observations at the same time, namely, according to Equation (1), that $y_i \sim N(\mu_i, \sigma_\varepsilon^2)$ where $\mu_i = p(x_i)$, $i = 1, \ldots, n$. However, note that BUGS parameterises the normal distribution using a precision parameter, as opposed to R which uses the standard deviation. That is, the quantity `tauy` holds the values $\frac{1}{\sigma_\varepsilon^2}$. A summary on how BUGS and R parameterise common distributions is given in LeBauer *et al.* (2013).

Stan

In Stan, it is most convenient to implement Horner's scheme via a function in the `function definition` block. The necessary code is a straightforward implementation of Equation (11) and is given in lines 25–35 in Appendix B.

Note that in Stan the model for the y observations has to be specified in the `model`, given in lines 197–199 of Appendix B, and that in Stan, just as in R, the (univariate) normal distribution is parameterised by the mean and standard deviation.

3.5. Starting values for MCMC chains

As depicted in Figure 1, the parameters for which starting values for the MCMC chains may be specified by the user are $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, β_0 and σ_ε^2 (or, equivalently, on the precision $\tau_y = 1/\sigma_\varepsilon^2$). One possibility for choosing such starting values is to fit, analogously to Murray *et al.* (2016), unconstrained polynomials to the data and derive some initial values for $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$ and β_0 from these fits; detailed formulae are given in Murray (2015). While Murray *et al.* (2016) only needed starting values for the regression parameters, in a Bayesian framework we may also choose to specify starting values for σ_ε^2. If so, these starting values can be conveniently initialised by the square of the residual sum of squares of the initial polynomial fits.

It is worthwhile to mention that within our current framework, which uses MCMC for model fitting, we have more liberties regarding the choice of starting values. In particular, it is possible to choose as starting values $\beta_0 = \bar{y} = \frac{1}{n}\sum_{i=1}^n y_i$, $\boldsymbol{\beta}_1 = \boldsymbol{\beta}_2 = \mathbf{0}$ and σ_ε^2 as the sample variance of the y_is. In the framework of Murray *et al.* (2016) these starting values would not be suitable as they would start the numerical minimisation of the residual sum of squares at a saddle point, i.e. a critical point, of that objective function.

Finally, we recommend to run at least two chains, starting from different initial values, when fitting monotone polynomials to data, as further discussed in Section 4.

BUGS

We recommend to specify starting values when running the BUGS code in Appendices A.1–A.5, regardless of whether it is run directly from one of the implementations of the BUGS language (WinBUGS, OpenBUGS, JAGS) or from one of the interfaces between R and one of these implementations. The R scripts provided in the supplementary material illustrate how initial values could be determined.

Initial values for β_0 should be passed to `beta0`, the initial values for $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$ should be passed to the $(K+1) \times 2$ matrix `bn`, and initial values for $1/\sigma_\varepsilon^2$ should be passed to `tauy`.

Stan

Due to the more robust nature of Stan's sampling algorithms, we can rely on its built in initialisation procedure. For specific details see Stan Development Team (2016, pp. 113-114).

3.6. Choice of priors

For the purposes of this exposition, we propose to use by default relatively vague priors on all parameters. It is trivial for the user to change the hyperparamters of the priors, or even the priors themselves.

BUGS

In BUGS priors can be specified by using relatively vague normal priors on the regression parameters $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$ and β_0 and a relatively vague gamma prior on the precision parameter τ_y (or, equivalently, an inverse gamma prior on σ_ε^2). These can be specified as follows:

```
beta0 ~ dnorm(0, 0.001)
for(j in 1:(K+1)){
  for(k in 1:2){
    bn[j,k] ~ dnorm(0, 0.001)
  }
}
tauy ~ dgamma(0.01, 0.01)
sigy <- 1/sqrt(tauy)
```

See the code towards the end of each listing in Appendices A.1–A.5. Alternatively, if the user prefers to put a vague prior on σ_ε, the last two lines in the above snippet could be replaced with the following code:

```
sigy ~ dunif(0,10)
tauy <- 1/pow(sigy, 2)
```

We found that these priors are suitably vague if the x and y observations are rescaled such that they fall roughly into the interval $[-1, 1]$, the rescaling also recommended by Murray *et al.* (2013, 2016) to avoid numerical precision problems.

As described in Section 3.2, the values of $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$ are stored in an $(K+1)\times 2$ matrix called `bn` in BUGS and initial values for these vectors should be passed to this object. An issue not yet addressed arises from the fact that for some parameterisations in (5)–(7) either $\boldsymbol{\beta}_1$ and/or $\boldsymbol{\beta}_2$, is only of length $K-1$, i.e. has K parameters. In this case, the corresponding entry in row $(K+1)$ of `bn` has to be initialised to zero *and* has to be kept fixed at zero during the MCMC iterations. The latter can be achieved in various ways in BUGS, for example by putting a Bernoulli prior with success probability zero on the parameter(s) in question:

```
bn[K+1,1] ~ dbern(0)
bn[K+1,2] ~ dbern(0)
```

See lines 45–46 of Appendices A.2 and A.4, but also line 46 of Appendix A.3, and line 51 of Appendix A.5.

Stan

Stan uses by default "flat" (improper) priors (Stan Development Team 2016, Chapter 8.3). When these priors are used, the results from an analysis using Stan are similar to that of an analysis with BUGS which utilises the vague (proper) priors described above, which is encouraging.

Alternatively, a user might want to specify other priors for some, or all, of the parameters $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, β_0 and σ_ε. This should be done within the `model` block, given in lines 197–199 of Appendix B.

3.7. Credible intervals and prediction intervals

In a Bayesian framework it is easy to calculate either credible intervals for $p(x)$ or prediction intervals for future observations, at arbitrary x values. The vector of x values at which we desire credible and/or prediction intervals should be passed to the object `xnew` and the length of this vector to `Nnew`. Then, during the MCMC calculations, we evaluate the polynomial at all values in `xnew` given the current value of $\boldsymbol{\beta}$ in each MCMC iteration. These values are realisations of the posterior distribution of $p(x)$ and allow us to calculate credible intervals. By adding random noise to these evaluations, using in each MCMC iteration the current value for σ_ε^2 (or its appropriate transformation), we obtain realisations from the posterior predictive distribution of $Y = p(x) + \varepsilon$ which allows us to calculate prediction intervals.

BUGS

Given the discussion above and in the previous sections, the following BUGS code needed for these calculations should be self-explanatory:

```
for(i in 1:Nnew){
  hnew[i,1] <- beta[q+1]
  for(k in 1:q){
    hnew[i,k+1] <- hnew[i,k]*xnew[i] + beta[q+1-k]
  }
  mupred[i] <- hnew[i,q+1]
  ypred[i] <- mupred[i] + eps[i]
  eps[i] ~ dnorm(0, tauy)
}
```

See lines 12–20 in Appendices A.1–A.5. If the analyses does not require the determination of credible intervals for $p(x)$, or prediction intervals for new observations, these lines can be deleted from the code.

Stan

In Stan the quantities `Nnew` and `xnew` have to be defined in the `data` block (lines 38–73), see lines 71–72 of Appendix B. The code for generating, at each MCMC iteration, a fitted value and a predicted value for each element of `xnew` has to be specified in the `generated quantities` block; see lines 201–207 of Appendix B.

While Stan parameterises functions related to *univariate* normal distributions by the mean and *standard deviation*, the *multivariate* normal distributions are parameterised by the mean vector and the *variance-covariance matrix*. Fortunately, for the latter there are also implementations that accept the Cholesky factorisation of the variance-covariance matrix as an argument. In our case this Cholesky factorisation is easily determined. Consequently, we use the function `multi_normal_cholesky_rng()` instead of `multi_normal_rng()`. Alternatively, we could replace line 206 by a loop similarly to the examples in Stan's manual (Stan Development Team 2016, pp. 150, 156, 157, 258 and 260):

```
for(i in 1:Nnew)
  ypred = normal_rng(mupred[i], sd_y);
```

4. Numerical experiments and results

We first illustrate our methods with the Stan implementation, using the data from Ramsay (1998) on the growth of a 10 year old boy. The measurements were taken over a period of 312 days during a school year, and there are a total of 83 height measurements. Given the period over which these measurements were taken, one would expect the underlying growth curve to be monotone. Due to the noise in the data, flexible regression methods that lack monotonicity constraints could result in periods of estimated decline in height. Figure 2 shows the data together with two fitted monotone polynomials of degree 7 and 9. Both fits also show (pointwise) 95% credible intervals and (pointwise) 95% prediction intervals. There is little difference between the two fitted regression curves, with the higher degree polynomial perhaps capturing the "flat" part around $x = 0.35$ more appropriately. Increasing the degree of polynomial further produced essentially the same fit as in the right panel of Figure 2, which is comparable to those obtained by monotone non-parametric fits to the same data; see, for example, Curtis (2008, Figure 4.3, p. 90).

When considering the aforementioned example using the BUGS implementation, similar fits and credible/predictive intervals to those observed in Figure 2 were obtained. However, the traceplots of multiple chains of $\boldsymbol{\beta}$ showed insufficient mixing for the purposes of statistical inference on these parameters. Consequently, based on this and other numerical experiments (not reported here), we recommend to fit only relatively low order polynomials, up to around degree 5, in BUGS. We conjecture that the reason for this behaviour is that small changes to one parameter in either $\boldsymbol{\beta}_1$ or $\boldsymbol{\beta}_2$ can have quite an effect on many parameters in $\boldsymbol{\beta}$ due to the way these quantities are related. The higher the degree of the monotone polynomial that

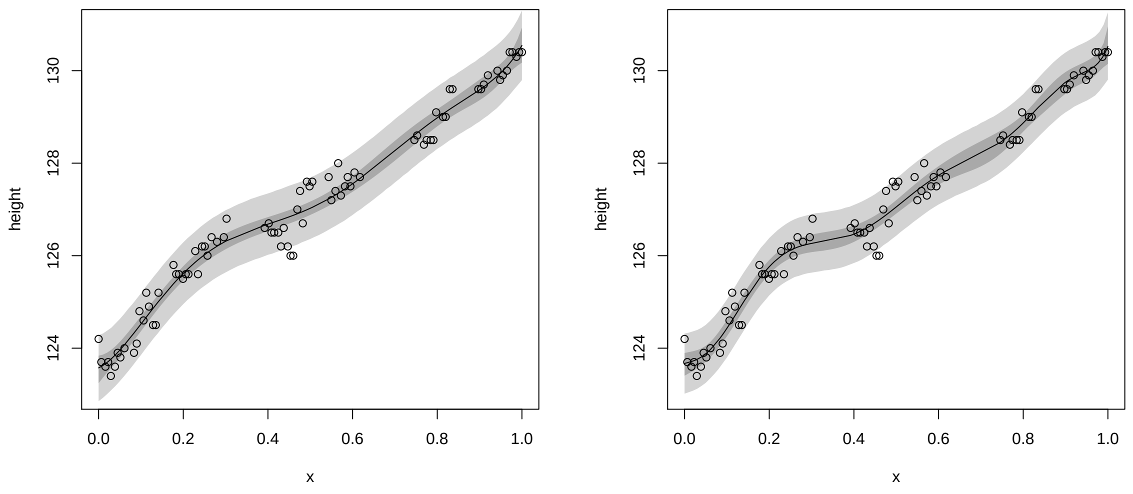

Figure 2: Plot of height of 10 year old boy against x, where x is scaled followup time over a one year period. Light grey areas indicate (pointwise) 95% prediction intervals while dark grey areas indicate (pointwise) 95% credible interval. The solid line is the fitted monotone polynomial regression curve. The left panel shows a polynomial of degree 7 that is montone on $[0, \infty)$, while the right panel uses a polynomial of degree 9.

is fitted to the data, the more parameters in $\boldsymbol{\beta}$ change when only one parameter is changed in either $\boldsymbol{\beta}_1$ or $\boldsymbol{\beta}_2$. Consequently, the Gibbs sampling algorithms underlying BUGS does not effectively explore the parameter space of $\boldsymbol{\beta}$ for high degree monotone polynomials in our parametrisation. On the other hand, the Hamiltonian Monte Carlo sampling used by Stan does not suffer from this issue.

As mentioned after Proposition 2.1, the parameterisation of non-negative polynomials via sum-of-squared polynomials is not necessarily unique. This is arguably easiest seen by considering the case $q = 1$ and $\mathcal{R} = (-\infty, \infty)$, when $p_1(x)$ and $p_2(x)$ in Equation (4) are both constants. Thus, the parameterisation of monotone polynomials using this sum-of-squared polynomials is also not unique. Consequently, to monitor whether the Markov chains for various parameters have converged, we recommend to monitor only the parameters of interest, typically $\boldsymbol{\beta}$ and σ_ε. Futhermore, for high degree monotone polynomials the trace plots of the higher order coefficients in $\boldsymbol{\beta}$ might display wild fluctations, and as such it might be more sensible to monitor fitted and predicted values (`mupred` and `ypred` in the code).

Murray *et al.* (2016) reported experiencing fewer problems with local extrema in the objective function when using the squared polynomial parameterisation of monotone polynomials for fitting such polynomials to data instead of Elphinstone-type parameterisations. Essentially, using the squared polynomial parameterisation, they did not experience problems with local extrema if $\mathcal{R} = (-\infty, \infty)$, rarely when $\mathcal{R}$ was a semi-compact interval and occasionally when $\mathcal{R}$ was a compact interval. One would hope that in a Bayesian framework, due to the additional regularisation provided by the priors, there are even less problems with multiple extrema in the objective function. However, at least when vague priors are used, the problem of multiple extrema still exists as illustrated here using the generated data set shown in Figure 3. This

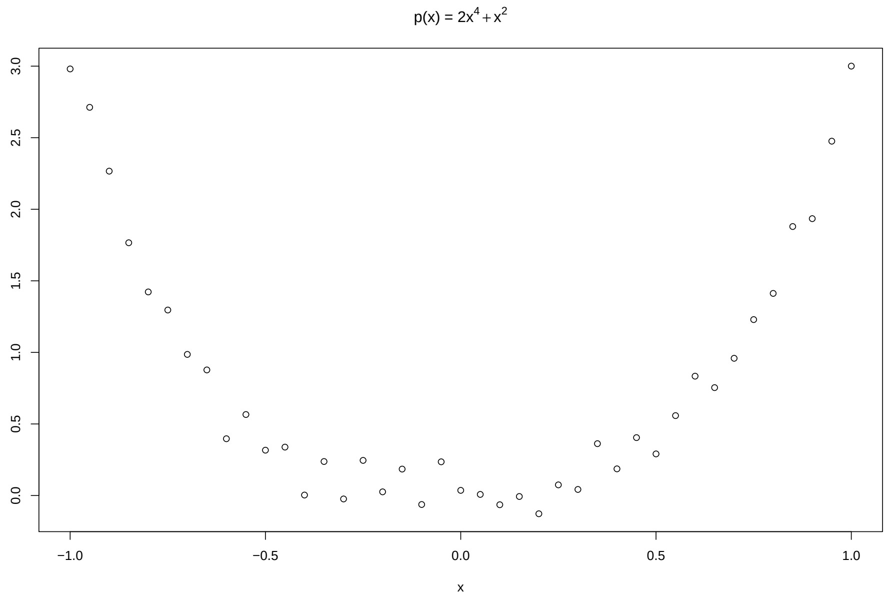

Figure 3: Data set simulated from the regression model $y = x^2 + 2x^4 + \varepsilon$, where ε is normally distributed with mean 0 and $\sigma_\varepsilon = 0.1$.

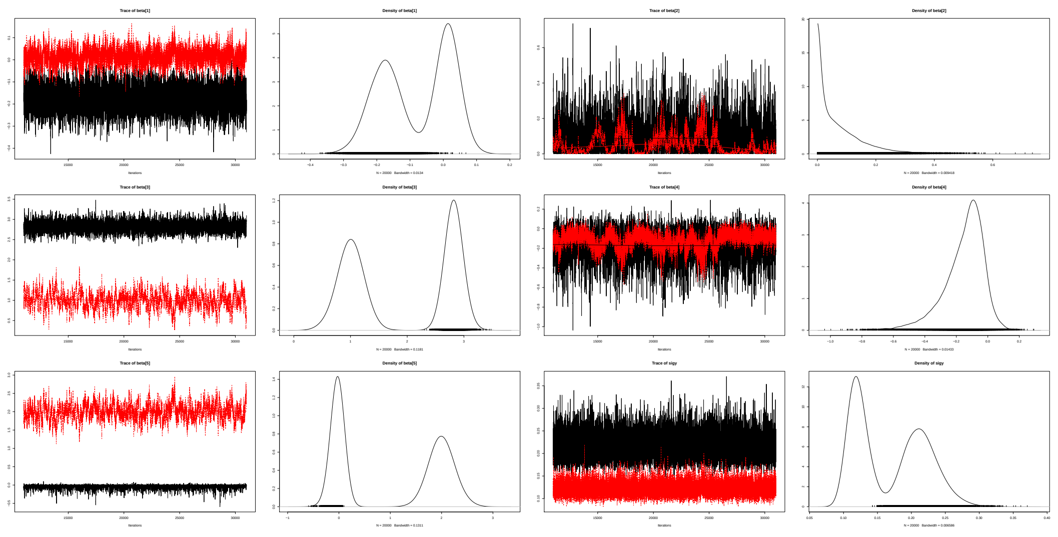

Figure 4: Diagnostic plots for parameter estimates when using two chains to fit a polynomial of degree 4 that is monotone on $[0, 1]$ to the data in Figure 3. Panels in first and third column show trace plots of the simulated values in both chains, while panels in second and fourth panel show density plots of these values. The parameters, arranged from row-wise from top left to bottom right, are $\beta_0, \beta_1, \ldots, \beta_4$ and σ_ε.

data set uses 41 equidistant x values in $[-1, 1]$ and y values were generated by the polynomial $x^2 + 2x^4$, adding normal noise with mean 0 and standard deviation $\sigma_\varepsilon = 0.1$. Figure 4 shows trace and density plots of the parameters $\beta_0, \beta_1, \ldots, \beta_4$ and σ_ε when running two chains in

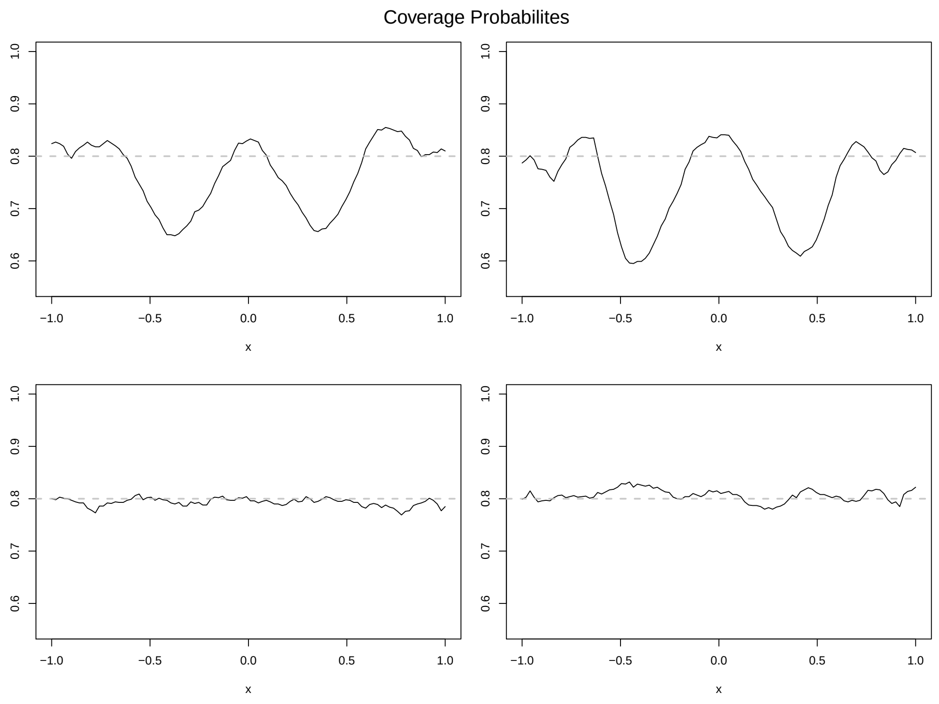

Figure 5: This plot demonstrates how well 80% credible intervals are calibrated. For the top row the underlying polynomial is x^5 while for the lower row it is $3x + 2x^2 + x^5$. Actual (pointwise) coverage of the regression curve when fitting monotone polynomials of degree 5 are shown in the first column, while the second column is based on fitting monotone polynomials of degree 7.

BUGS using code that fits polynomials which are monotone on $[0, 1]$. It can be clearly seen that one chain (red) fits a quartic polynomial to the data with a low residual noise, while the other chain (black) fits a quadratic polynomial with a high residual noise to the data. Incidentally, when using these data with the function `monpol` of the R package `MonoPoly` (Turlach and Murray 2016) and various random starting values, convergence to more than two local extrema were also observed.

Thus, as mentioned in Section 3.5, we recommend to run several MCMC chains, in particular when fitting polynomials that are only constrained to be monotone over a (semi-)compact interval. Not only does running multiple chains help with assessing whether the chains have converged and are mixing, it also helps with detecting problems due to multiple extrema in the objective function.

Finally, Murray *et al.* (2016) investigated bootstrap methods for statistical inference purposes, such as selecting the degree of the polynomial and obtaining (pointwise) confidence bands, for monotone polynomials. When using bootstrap methods for obtaining confidence bands for monotone polynomials, they discovered that the confidence intervals have the wrong coverage for polynomials that lie[6] on the bounday of $\mathcal{C}^q_{\mathcal{R}}$, the (non-pointed) closed convex cone that

[6]Here we use the same terminology as Murray *et al.* (2016) and say that a polynomial lies in $\mathcal{C}^q_{\mathcal{R}}$ if its vector of coefficients lies in that set.

contains all $\boldsymbol{\beta} \in \mathbb{R}^{q+1}$ for which $p(x;\boldsymbol{\beta})$ is monotone (increasing) on $\mathcal{R} \subseteq \mathbb{R}$ and a polynomial of degree at most q. Thus, it is of interest to see whether the credible intervals produced in the Bayesian framework are well calibrated.

A small simulation study to address this issue is summarised in Figure 5. For this simulation study we used the polynomials $p(x) = x^5$ and $p(x) = 3x + 2x^2 + x^5$. The former lies on the boundary of $\mathcal{C}^5_{\mathbb{R}}$ and $\mathcal{C}^7_{\mathbb{R}}$ while the latter lies inside $\mathcal{C}^5_{\mathbb{R}}$ but on the boundary of $\mathcal{C}^7_{\mathbb{R}}$. We repeatedly simulated data sets from these two polynomials and fitted monotone polynomials of degrees 5 and 7, as well as determined 80% credible intervals using a grid of x values. For each x value in this grid we determined the proportion of times that the true value of the regression curve at that point was lying inside the 80% credible interval. The resulting proportions are shown in Figure 5. From Figure 5 it is apparent[7] that fitting monotone polynomials in a Bayesian framework, when using vague priors, suffers the same problem as fitting monotone polynomials in a frequentist framework using bootstrap methods for inference purposes. This remains an open research problem in both the Bayesian and frequentist frameworks.

5. Conclusions and future work

In this paper we have demonstrated methods to fit monotone polynomials to data within a Bayesian framework using existing and readily available probabilistic languages such as BUGS and Stan. A byproduct of using a sum-of-squared polynomials parameterisation was that we demonstrated how to implement a convolution operation in both languages; a computational operation which does not seem to be obvious to implement (Seixas 2016).

For our main topic of interest, monotone polynomials, we have provided easily implementable code, enabling the end user to adapt it to their particular scenario, and presented the results from numerical experiments from real and simulated data. The effectiveness of Stan was particularly pronounced for higher order polynomials, whilst issues with inter-chain mixing and convergence were identified in BUGS for such models. These issues were due to the inability of the Gibbs sampling algorithm to appropriately explore a highly connected and correlated parameter space.

A challenging problem in either language is to find suitable priors that address the problems with the calibration of the credible intervals if the underlying polynomial lies on the boundary of its cone. In the frequentist framework, Murray *et al.* (2016) proposed an ad-hoc way of correcting the coverage problem of the bootstrap confidence intervals, but a Bayesian framework would need a more principled solution.

Various extensions of the work presented here may be worth investigating more thoroughly and are left for future research. For example, it would be relatively easy to replace the specification of a normal distribution for the error term in Equation (1) by a t-distribution, probably with unknown degrees of freedom, which could increase the robustness in fitting regression curves in a Bayesian framework. Another potential extension is to develop a framework for linear mixed effects models or hierarchical models. The approach of Murray *et al.* (2016) was extended in a frequentist framework to mixed models by Bon (2016). We are currently investigating how the Bayesian framework employed here can be extended to mixed models by using hierarchical priors on (some of) the coefficients of $p_1(x)$ in Equations (4)–(7).

Finally, Murray *et al.* (2016) use in their frequentist approach the m-out-of-n bootstrap

[7] And perhaps also reassuring, albeit somewhat disappointing.

successfully for model selection, i.e. for determining the order of the monotone polynomial necessary for the data at hand. Presumably, in a Bayesian framework choosing the correct degree of the polynomial could be addressed via a reversible jump MCMC approach. While this might not be feasible to implement in a probabilistic language such as BUGS or Stan, hence necessitating to write custom-made MCMC code, it would be another worthwhile project for future research.

A. BUGS implementation

A.1. Monotone on the real line

```
model{
  for(i in 1:N){
    y[i] ~ dnorm(mu[i], tauy)

    horner[i,1] <- beta[q+1]
    for(k in 1:q){
      horner[i,k+1] <- horner[i,k]*x[i] + beta[q+1-k]
    }
    mu[i] <- horner[i,q+1]
  }

  for(i in 1:Nnew){
    hnew[i,1] <- beta[q+1]
    for(k in 1:q){
      hnew[i,k+1] <- hnew[i,k]*xnew[i] + beta[q+1-k]
    }
    mupred[i] <- hnew[i,q+1]
    ypred[i] <- mupred[i] + eps[i]
    eps[i] ~ dnorm(0, tauy)
  }

  for(j in 1:q){
    beta[j+1] <- alpha * gamma[j]/j
  }
  beta[1] <- beta0

  for(j in 1:q){
    gamma[j] <- g1[j] + g2[j]
  }


  for(j in 1:q){
    g1[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),1],
                    bnrev[(max(0,Kp1-j)+1):(min(q-j,K)+1),1])
    g2[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),2],
                    bnrev[(max(0,Kp1-j)+1):(min(q-j,K)+1),2])
  }

  for(j in 1:(K+1)){
    for(k in 1:2){
      bnrev[j,k] <- bn[Kp1-j+1,k]
    }
  }

  beta0 ~ dnorm(0, 0.001)
  for(j in 1:(K+1)){
    for(k in 1:2){
      bn[j,k] ~ dnorm(0, 0.001)
    }
  }
  tauy ~ dgamma(0.01, 0.01)
  sigy <- 1/sqrt(tauy)

  Kp1 <- K+1
}
```

A.2. Even degree polynomial monotone on $[a, \infty)$

```
model{
  for(i in 1:N){
    y[i] ~ dnorm(mu[i], tauy)

    horner[i,1] <- beta[q+1]
    for(k in 1:q){
      horner[i,k+1] <- horner[i,k]*x[i] + beta[q+1-k]
    }
    mu[i] <- horner[i,q+1]
  }

  for(i in 1:Nnew){
    hnew[i,1] <- beta[q+1]
    for(k in 1:q){
      hnew[i,k+1] <- hnew[i,k]*xnew[i] + beta[q+1-k]
    }
    mupred[i] <- hnew[i,q+1]
    ypred[i] <- mupred[i] + eps[i]
    eps[i] ~ dnorm(0, tauy)
  }

  for(j in 1:q){
    beta[j+1] <- alpha*gamma[j]/j
  }
  beta[1] <- beta0

  for(j in 2:q){
    gamma[j] <- g1[j] + g2[j-1] - a*g2[j]
  }
  gamma[1] <- g1[1] - a * g2[1]

  for(j in 1:q){
    g1[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),1],
                    bnrev[(max(0,Kp1-j)+1):(min(q+1-j,K)+1),1])
    g2[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),2],
                    bnrev[(max(0,Kp1-j)+1):(min(q+1-j,K)+1),2])
  }

  for(j in 1:(K+1)){
    for(k in 1:2){
      bnrev[j,k] <- bn[Kp1-j+1,k]
    }
  }


  bn[K+1,1] ~ dbern(0)
  bn[K+1,2] ~ dbern(0)
  beta0 ~ dnorm(0, 0.001)
  for(j in 1:K){
    for(k in 1:2){
      bn[j,k] ~ dnorm(0, 0.001)
    }
  }
  tauy ~ dgamma(0.01, 0.01)
  sigy <- 1/sqrt(tauy)

  Kp1 <- K+1
}
```

A.3. Odd degree polynomial monotone on $[a, \infty)$

```
model{
  for(i in 1:N){
    y[i] ~ dnorm(mu[i], tauy)

    horner[i,1] <- beta[q+1]
    for(k in 1:q){
      horner[i,k+1] <- horner[i,k]*x[i] + beta[q+1-k]
    }
    mu[i] <- horner[i,q+1]
  }

  for(i in 1:Nnew){
    hnew[i,1] <- beta[q+1]
    for(k in 1:q){
      hnew[i,k+1] <- hnew[i,k]*xnew[i] + beta[q+1-k]
    }
    mupred[i] <- hnew[i,q+1]
    ypred[i] <- mupred[i] + eps[i]
    eps[i] ~ dnorm(0, tauy)
  }

  for(j in 1:q){
    beta[j+1] <- alpha*gamma[j]/j
  }
  beta[1] <- beta0

  for(j in 2:q){
    gamma[j] <- g1[j] + g2[j-1] - a*g2[j]
  }
  gamma[1] <- g1[1] - a * g2[1]

  for(j in 1:q){
    g1[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),1],
                    bnrev[(max(0,Kp1-j)+1):(min(q-j,K)+1),1])
    g2[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),2],
                    bnrev[(max(0,Kp1-j)+1):(min(q-j,K)+1),2])
  }

  for(j in 1:(K+1)){
    for(k in 1:2){
      bnrev[j,k] <- bn[Kp1-j+1,k]
    }
  }

  bn[K+1,1] ~ dnorm(0, 0.001)
  bn[K+1,2] ~ dbern(0)
  beta0 ~ dnorm(0, 0.001)
  for(j in 1:K){
    for(k in 1:2){
      bn[j,k] ~ dnorm(0, 0.001)
    }
  }
  tauy ~ dgamma(0.01, 0.01)
  sigy <- 1/sqrt(tauy)

  Kp1 <- K+1
}
```

A.4. Even degree polynomial monotone on $[a, b]$

```
model{
```

```
for(i in 1:N){
  y[i] ~ dnorm(mu[i], tauy)

  horner[i,1] <- beta[q+1]
  for(k in 1:q){
    horner[i,k+1] <- horner[i,k]*x[i] + beta[q+1-k]
  }
  mu[i] <- horner[i,q+1]
}

for(i in 1:Nnew){
  hnew[i,1] <- beta[q+1]
  for(k in 1:q){
    hnew[i,k+1] <- hnew[i,k]*xnew[i] + beta[q+1-k]
  }
  mupred[i] <- hnew[i,q+1]
  ypred[i] <- mupred[i] + eps[i]
  eps[i] ~ dnorm(0, tauy)
}

for(j in 1:q){
  beta[j+1] <- alpha*gamma[j]/j
}
beta[1] <- beta0

for(j in 2:q){
  gamma[j] <- b*g1[j] - g1[j-1] + g2[j-1] - a*g2[j]
}
gamma[1] <- b*g1[1] - a * g2[1]

for(j in 1:q){
  g1[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),1],
                  bnrev[(max(0,Kp1-j)+1):(min(q+1-j,K)+1),1])
  g2[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),2],
                  bnrev[(max(0,Kp1-j)+1):(min(q+1-j,K)+1),2])
}

for(j in 1:(K+1)){
  for(k in 1:2){
    bnrev[j,k] <- bn[Kp1-j+1,k]
  }
}


bn[K+1,1] ~ dbern(0)
bn[K+1,2] ~ dbern(0)
beta0 ~ dnorm(0, 0.001)
```

```
  for(j in 1:K){
    for(k in 1:2){
      bn[j,k] ~ dnorm(0, 0.001)
    }
  }
  tauy ~ dgamma(0.01, 0.01)
  sigy <- 1/sqrt(tauy)

  Kp1 <- K+1
}
```

A.5. Odd degree polynomial monotone on $[a, b]$

```
model{
  for(i in 1:N){
    y[i] ~ dnorm(mu[i], tauy)

    horner[i,1] <- beta[q+1]
    for(k in 1:q){
      horner[i,k+1] <- horner[i,k]*x[i] + beta[q+1-k]
    }
    mu[i] <- horner[i,q+1]
  }

  for(i in 1:Nnew){
    hnew[i,1] <- beta[q+1]
    for(k in 1:q){
      hnew[i,k+1] <- hnew[i,k]*xnew[i] + beta[q+1-k]
    }
    mupred[i] <- hnew[i,q+1]
    ypred[i] <- mupred[i] + eps[i]
    eps[i] ~ dnorm(0, tauy)
  }

  for(j in 1:q){
    beta[j+1] <- alpha*gamma[j]/j
  }
  beta[1] <- beta0

  for(j in 2:q){
    gamma[j] <- g1[j] + g2b[j-1] - a*g2b[j]
  }
  gamma[1] <- g1[1] - a * g2b[1]

  for(j in 2:q){
    g2b[j] <- b*g2[j] - g2[j-1]
  }
  g2b[1] <- b*g2[1]

  for(j in 1:q){
    g1[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),1],
                    bnrev[(max(0,Kp1-j)+1):(min(q-j,K)+1),1])
    g2[j] <- inprod(bn[(max(0,j-Kp1)+1):(min(Kp1,j)),2],
                    bnrev[(max(0,Kp1-j)+1):(min(q-j,K)+1),2])
  }

  for(j in 1:(K+1)){
    for(k in 1:2){
      bnrev[j,k] <- bn[Kp1-j+1,k]
    }
  }

  bn[K+1,1] ~ dnorm(0, 0.001)
  bn[K+1,2] ~ dbern(0)
  beta0 ~ dnorm(0, 0.001)
  for(j in 1:K){
    for(k in 1:2){
      bn[j,k] ~ dnorm(0, 0.001)
    }
  }
  tauy ~ dgamma(0.01, 0.01)
  sigy <- 1/sqrt(tauy)

  Kp1 <- K+1
}
```

B. Stan implementation

```
functions {
  vector convolve(vector a, vector b){
    int na;
    int nb;
    int nc;
    vector[rows(b)] rev_b;
    vector[rows(a)+rows(b)-1] c;

    na = rows(a);
    nb = rows(b);
    nc = na+nb-1;
    for(i in 1:nb){
      rev_b[i] = b[nb-i+1];
    }
    for(j in 1:nc){
      int istart;
      int istop;
      istart = max(0, j-nb) + 1;
      istop = min(na, j);
      c[j] = sum(a[istart:istop] .* rev_b[(nb-j+istart):(nb-j+istop)]);
    }
    return c;
  }

  vector horner(vector x, vector beta){
    int nb;
    vector[rows(x)] res;

    nb = rows(beta);
    res = rep_vector(beta[nb], rows(x));
    for(i in 1:(nb-1)){
      res = res .* x + beta[nb-i];
    }
    return res;
  }
}

data {
  # number of data points
  int <lower=1> N;

  # degree of polynomial we want to fit
  int <lower=1> q;

  # degree of sub polynomials, q = 2K or q = 2K + 1
  int <lower=0> K;

  # mode of operation, derived in the R code
  # 1 == whole real line
  # 2 == (-inf, b]
  # 3 == [a, Inf)
  # 4 == [a, b]
  int <lower = 1, upper = 4> operation_mode;

  # y values
  vector[N] y;

  # x values
  vector[N] x;

  # lower bound (could be -Infty / negative_infinity() )
  real a;

  # upper bound (could be Infty / positive_infinity() )
  real <lower=a> b;

  # alpha (polynomial is increasing or decreasing)
  int <lower=-1, upper=1> alpha;

  # number of new data points at which to predict
  int <lower=1> Nnew;
  vector[Nnew] xnew;
}

transformed data {
  int p1_length;
  int p2_length;
  int gamma_1_length;
  int gamma_2_length;
  int gamma_1_temp_length;
  int gamma_2_temp_length;
  vector[2] lower_bound_vector;
  vector[2] upper_bound_vector;

  gamma_1_temp_length = 0;
  gamma_2_temp_length = 0;
  if ((operation_mode == 1) && (q % 2 != 0)) {
    p1_length = K + 1;
    p2_length = K + 1;
  } else if (operation_mode == 2 || operation_mode == 3) {
      if (q % 2 == 0) {
        p1_length = K;
        p2_length = K;
      } else {
        p1_length = K + 1;
        p2_length = K;
      }
      gamma_2_temp_length = 1;
  } else {
      if (q % 2 == 0) {
        p1_length = K;
        p2_length = K;
        gamma_1_temp_length = 1;
        gamma_2_temp_length = 1;
      } else {
        p1_length = K + 1;
        p2_length = K;
        gamma_2_temp_length = 2;
      }
  }
  gamma_1_length = (2 * p1_length) - 1;
  gamma_2_length = (2 * p2_length) - 1;
  if( gamma_1_temp_length != 0)
    gamma_1_temp_length =  gamma_1_temp_length + gamma_1_length;
  if( gamma_2_temp_length != 0)
    gamma_2_temp_length =  gamma_2_temp_length + gamma_2_length;

  # This should be the vector (-a, 1) for (a - x) convolution
  lower_bound_vector[1] = -a;
  lower_bound_vector[2] = 1;

  # This should be the vector (b, -1) for (b - x) convolution
  upper_bound_vector[1] = b;
  upper_bound_vector[2] = -1;
}

parameters {
  vector[p1_length] beta_1;
  vector[p2_length] beta_2;

  real<lower=0> sd_y;
  real beta_zero;
}

transformed parameters {
  vector[gamma_1_length] gamma_1;
  vector[gamma_2_length] gamma_2;
  vector[gamma_1_temp_length] gamma_1_temp;
  vector[gamma_2_temp_length] gamma_2_temp;
  vector[q] gamma;
  vector[q + 1] beta_final;
  vector[N] mu;

  gamma_1 = rep_vector(0, gamma_1_length);
  gamma_2 = rep_vector(0, gamma_2_length);
  gamma_1_temp = rep_vector(0, gamma_1_temp_length);
  gamma_2_temp = rep_vector(0, gamma_2_temp_length);
  gamma = rep_vector(0, q);

  # self convolute betas to get gammas
  gamma_1 = convolve(beta_1, beta_1);
  gamma_2 = convolve(beta_2, beta_2);

  # convolute with lower and upper bounds.
  # combine together to get gamma
  # do this at the same time to avoid extra control flow steps
  if (operation_mode == 1) {
    gamma = gamma_1 + gamma_2;
  } else if (operation_mode == 2) {
      gamma_2_temp = convolve(upper_bound_vector, gamma_2);
      if(q % 2 == 0) {
        gamma[1:(gamma_1_length)] = gamma_1;
        gamma = gamma + gamma_2_temp;
      } else {
        gamma[1:(gamma_2_temp_length)] = gamma_2_temp;
        gamma = gamma + gamma_1;
      }
  } else if (operation_mode == 3) {
      gamma_2_temp = convolve(lower_bound_vector, gamma_2);
      if(q % 2 == 0) {
        gamma[1:(gamma_1_length)] = gamma_1;
        gamma = gamma + gamma_2_temp;
      } else {
        gamma[1:(gamma_2_temp_length)] = gamma_2_temp;
        gamma = gamma + gamma_1;
      }
  } else {
      if (q % 2 == 0) {
        gamma_1_temp = convolve(upper_bound_vector, gamma_1);
        gamma_2_temp = convolve(lower_bound_vector, gamma_2);
        gamma = gamma_1_temp + gamma_2_temp;
      } else {
        gamma_2_temp = convolve(upper_bound_vector, convolve(lower_bound_vector
        gamma = gamma_1 + gamma_2_temp;
      }
  }

  beta_final[1] = beta_zero;
  for(i in 1:q) {
    beta_final[i+1] = alpha * gamma[i] / i;
  }

  # use horner() to evaluate and get mu
  mu = horner(x, beta_final);
}

model {
 y ~ normal(mu, sd_y);
}

generated quantities {
  vector[Nnew] mupred;
  vector[Nnew] ypred;

  mupred = horner(xnew, beta_final);
  ypred = multi_normal_cholesky_rng(mupred, diag_matrix(rep_vector(sd_y, Nnew)
}
```

References

Bon JJ (2016). *Polynomial fitting in mixed effects models with closed convex constraints including monotonicity*. Honours dissertation, School of Mathematics and Statistics (M019), The University of Western Australia, 35 Stirling Highway, Crawley WA 6009, Australia.

Brickman L, Steinberg L (1962). "On nonnegative polynomials." *The American Mathematical Monthly*, **69**(3), 218–221.

Carpenter B, Gelman A, Hoffman M, Lee D, Goodrich B, Betancourt M, Brubaker M, Guo J, Li P, Riddell A (2017). "Stan: A Probabilistic Programming Language." *Journal of Statistical Software*, **76**(1), 1–32. ISSN 1548-7660. doi:10.18637/jss.v076.i01. URL https://www.jstatsoft.org/index.php/jss/article/view/v076i01.

Curtis SM (2008). *Variable Selection Methods with Applications to Shape Restricted Regression*. Ph.D. thesis, Statistics, Raleigh, North Carolina. URL https://repository.lib.ncsu.edu/handle/1840.16/4481.

Dette H, Studden WJ (1997). *The theory of canonical moments with applications in statistics, probability, and analysis*. Wiley Series in Probability and Statistics. Wiley & Sons, New York.

Elphinstone CD (1983). "A target distribution model for non-parametric density estimation." *Communications in Statistics - Theory and Methods*, **12**(2), 161–198.

Fausett LV (2003). *Numerical Methods: Algorithms and Applications*. Prentice Hall, Upper Saddle River, NJ.

Firmin L, Müller S, Rösler KM (2011). "A Method to Measure the Distribution of Latencies of Motor Evoked Potentials in Man." *Clinical Neurophysiology*, **122**(1), 176–182.

Firmin L, Müller S, Rösler KM (2012). "The Latency distribution of motor evoked potentials in patients with multiple sclerosis." *Clinical Neurophysiology*, **123**(12), 2414–2421.

Hawkins DM (1994). "Fitting monotonic polynomials to data." *Computational Statistics*, **9**(3), 233–247.

Heinzmann D (2008). "A filtered polynomial approach to density estimation." *Computational Statistics*, **23**(3), 343–360. doi:10.1007/s00180-007-0070-z.

Karlin S, Studden WJ (1966). *Tchebycheff Systems: With Applications in Analysis and Statistics*. Wiley & Sons, New York.

LeBauer D, Dietze M, Bolker B (2013). "Translating Probability Distributions: From R to BUGS and Back Again." *The R Journal*, **5**(1), 207–210. URL http://journal.r-project.org/archive/2013-1/lebauer-dietze-bolker.pdf.

Lunn D, Jackson C, Best N, Thomas A, Spiegelhalter D (2012). *The BUGS Book: A Practical Introduction to Bayesian Analysis*. Texts in Statistical Science. Chapman & Hall/CRC, Boca Raton.

Lunn DJ, Spiegelhalter D, Thomas A, Best N (2009). "The BUGS project: Evolution, critique and future directions (with discussion)." *Statistics in Medicine*, **28**(25), 3049–3082.

Lunn DJ, Thomas A, Best N, Spiegelhalter D (2000). "WinBUGS – a Bayesian modelling framework: concepts, structure, and extensibility." *Statistics and Computing*, **10**(4), 325–337.

Marshall M (2008). *Positive Polynomials and Sums of Squares*, volume 146 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, Rhode Island 02904–2294, USA.

Murray K (2015). *Improved monotone polynomial fitting with applications and variable selection*. Ph.D. thesis, School of Mathematics and Statistics, University of Sydney.

Murray K, Müller S, Turlach BA (2013). "Revisiting fitting monotone polynomials to data." *Computational Statistics*, **28**(5), 1989–2005.

Murray K, Müller S, Turlach BA (2016). "Fast and flexible methods for monotone polynomial fitting." *Journal of Statistical Computation and Simulation*, **86**(15), 2946–2966. doi:10.1080/00949655.2016.1139582.

Plummer M (2016). *rjags: Bayesian Graphical Models using MCMC*. R package version 4-6, URL https://CRAN.R-project.org/package=rjags.

Ramsay JO (1998). "Estimating smooth monotone functions." *Journal of the Royal Statistical Society, Series B*, **60**(2), 365–375.

Seixas FL (2016). "Convolution operation." BUGS mailing list. URL https://www.jiscmail.ac.uk/cgi-bin/webadmin?A2=bugs;b9381fc8.1601.

Stan Development Team (2016). *Stan Modeling Language: User's Guide and Reference Manual*. Stan Version 2.14.0, URL http://mc-stan.org/documentation/.

Sturtz S, Ligges U, Gelman A (2005). "R2WinBUGS: A Package for Running WinBUGS from R." *Journal of Statistical Software*, **12**(1), 1–16. ISSN 1548-7660. doi:10.18637/jss.v012.i03. URL https://www.jstatsoft.org/index.php/jss/article/view/v012i03.

Thomas A, O'Hara B, Ligges U, Sturtz S (2006). "Making BUGS Open." *R News*, **6**(1), 12–17. URL http://cloud.r-project.org/doc/Rnews/Rnews_2006-1.pdf.

Turlach BA, Murray K (2016). *MonoPoly: Functions to Fit Monotone Polynomials*. R package version 0.3-8, URL http://cloud.r-project.org/package=MonoPoly.

Yan J, Prates M (2013). *rbugs: Fusing R and OpenBugs and Beyond*. R package version 0.5-9, URL http://cloud.r-project.org/package=rbugs.

Affiliation:

A.A. Manderson
Centre for Applied Statistics (M019)
The University of Western Australia
35 Stirling Highway
Crawley WA 6009, Australia
E-mail: andrew.manderson@research.uwa.edu.au

E. Cripps
Centre for Applied Statistics (M019)
The University of Western Australia
35 Stirling Highway
Crawley WA 6009, Australia
E-mail: Edward.Cripps@uwa.edu.au

K. Murray
School of Population and Global Health (M431)
The University of Western Australia
35 Stirling Highway
Crawley WA 6009, Australia
E-mail: Kevin.Murray@uwa.edu.au

B.A. Turlach
Centre for Applied Statistics (M019)
The University of Western Australia
35 Stirling Highway
Crawley WA 6009, Australia
E-mail: Berwin.Turlach@gmail.com

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd