Univariate Polynomials in R

Bill Venables

2019-04-09

A Univariate Polynomial Class for R

Introduction and summary

The following started as a straightforward programming exercise in operator overloading, but seems to be more generally useful. The goal is to write a polynomial class, that is a suite of facilities that allow operations on polynomials: addition, subtraction, multiplication, "division", remaindering, printing, plotting, and so forth, to be conducted using the same operators and functions, and hence with the same ease, as ordinary arithmetic, plotting, printing, and so on.

The class is limited to univariate polynomials, and so they may therefore be uniquely defined by their numeric coefficient vector. Coercing a polynomial to numeric yields this coefficient vector as a numeric vector.

For reasons of simplicity it is limited to REAL polynomials; handling polynomials with complex coefficients would be a simple extension. Dealing with polynomials with polynomial coefficients, and hence multivariate polynomials, would be feasible, though a major undertaking and the result would be very slow and of rather limited usefulness and efficiency.

General orientation

The function `polynom()`, alias `polynomial()`, creates an object of class `"polynom"` from a numeric coefficient vector. Coefficient vectors are assumed to apply to the powers of the carrier variable in increasing order, that is, in the *truncated power series* form, and in the same form as required by `polyroot()`, the system function for computing zeros of polynomials. [1]

Polynomials may also be created by specifying a set of (x, y) pairs and constructing the Lagrange interpolation polynomial that passes through them (`poly_calc(x, y)`). If `y` is a matrix, an interpolation polynomial is calculated for each column and the result is a list of polynomials (of class `polylist`).

The third way polynomials are commonly generated is via its zeros using `poly_calc(z)`, which creates the monic polynomial of lowest degree with the values in `z` as its zeros.

The core facility provided is the group method function `Ops.polynom()`, which allows arithmetic operations to be performed on polynomial arguments using ordinary arithmetic operators.

Changes in version 2.0.0

In this release, functions in previous releases using a period, `"."` in their name which were *not* S3 methods have been replaced by names using an underscore in place of the period. Thus the previously named function `poly.calc` has become `poly_calc`. The old names have been retained as aliases, for now, but will issue a `Deprecated` warning. A full list is given in Table 1 below.

Orthogonal polynomials

One possibly useful addition has been the function `poly_orth_general` which allows sets of orthogonal polynomials to be generated using an inner product definition using an R function. This must be a function where the first two arguments are polynomial objects with the second having a default value equal to the

[1] As a matter or terminology, the *zeros* of the polynomial $P(x)$ are the same as the *roots* of equation $P(x) = 0$.

first. Hence if the inner product function is called with just one polynomial argument it returns the squared L_2-norm.

Weighted discrete orthogonal polynomials can be included as a special case, and the inner product definition for this case serves as a template, as in this example, a special case of the Poisson-Charlier polynomials:

```
Discrete <- function(p, q = p, x, w = function(x) rep(1, length(x))) {
  sum(w(x)*p(x)*q(x))
}
poly_orth_general(inner_product = Discrete, degree = 5,
                  x = 0:20, w = function(x) dpois(x, 1))
List of polynomials:
$P0
1

$P1
-1 + x

$P2
1 - 3*x + x^2

$P3
-1 + 8*x - 6*x^2 + x^3

$P4
1 - 24*x + 29*x^2 - 10*x^3 + x^4

$P5
-1 + 89*x - 145*x^2 + 75*x^3 - 15*x^4 + x^5
```

This inner product function, along with some others for the classical orthogonal polynomials of mathematical physics are included, although if formulae for the recurrence relation are known, using them directly would be more efficient.

Table 1: Function name changes in version 2.0.0

Old name	New name
as.polylist	as_polylist
as.polynom	as_polynom
change.origin	change_origin
is.polylist	is_polylist
is.polynom	is_polynom
poly.calc	poly_calc
poly.from.roots	poly_from_roots
poly.from.values	poly_from_values
poly.from.zeros	poly_from_zeros
poly.orth	poly_orth

Notes

1. +, - and * have their obvious meanings for polynomials.
2. ^ is limited to non-negative integer powers.
3. / returns the polynomial quotient. If division is not exact the remainder is discarded, (but see 4.)
4. %% returns the polynomial remainder, so that if all arguments are polynomials then, provided p1 is not

the zero polynomial, `p1 * (p2 / p1) + p2 %% p1` will be the same polynomial as `p2`.

5. If numeric vectors are used in polynomial arithmetic they are coerced to polynomial, which could be a source of surprise. In the case of scalars, though, the result is natural.
6. Some logical operations are allowed, but not always very satisfactorily. `==` and `!=` mean exact equality or not, respectively, however `<`, `<=`, `>`, `>=`, `!`, `|` and `&` are not allowed at all and cause stops in the calculation.
7. Most Math group functions are disallowed with polynomial arguments. The only exceptions are `ceiling`, `floor`, `round`, `trunc`, and `signif`.
8. Summary group functions are not implemented, apart from `sum` and `prod`.
9. Polynomial objects, in this representation, are fully usable R functions of a single argument `x` , which may be a numeric vector, in which case the return value is a numerical vector of evaluations, or `x` may itself be a polynomial object, in which case the result is a polynomial object, the composition of the two, `p(x)`. More generally, the only restriction on the argument is that it belong to a class that has methods for the arithmetic operators defined, in which case the result is an object belonging to the result class.
10. The print method for polynomials can be slow and is a bit pretentious. The plotting methods (`plot`, `lines`, `points`) are fairly nominal, but may prove useful.

Examples

1. Miscellaneous computations using polynomial arithmetic.

```
(p1 <- poly_calc(1:6))
720 - 1764*x + 1624*x^2 - 735*x^3 + 175*x^4 - 21*x^5 + x^6
(p2 <- change_origin(p1, 3))
-12*x + 4*x^2 + 15*x^3 - 5*x^4 - 3*x^5 + x^6
p1(0:7)
[1] 720   0   0   0   0   0   0 720
p2(0:7)
[1]     0     0     0     0   720  5040 20160 60480
p2(0:7 - 3)
[1] 720   0   0   0   0   0   0 720
(p3 <- (p1 - 2 * p2)^2)          # moderate arithmetic expression.
518400 - 2505600*x + 5354640*x^2 - 6725280*x^3 + 5540056*x^4 - 3137880*x^5
+ 1233905*x^6 - 328050*x^7 + 53943*x^8 - 4020*x^9 - 145*x^10 + 30*x^11 +
x^12
p3(0:4)                          # should have 1, 2, 3 as zeros
[1]  518400       0       0       0 2073600
```

2. Find the orthogonal polynomials on $x_0 = (0,1,2,3,5)$ and construct R functions to evaluate them at arbitrary x values.

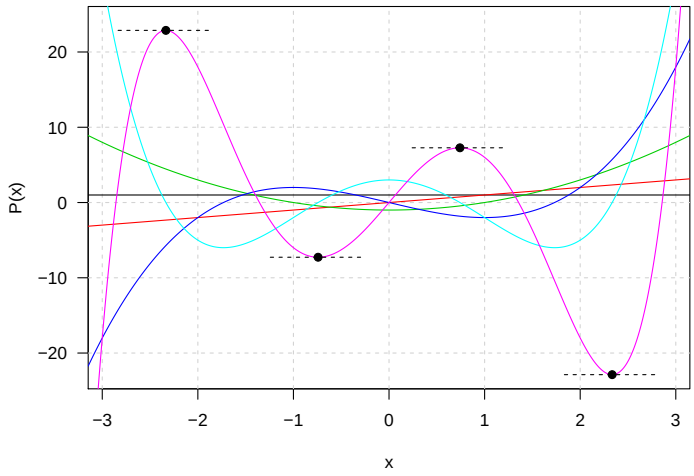
```
x0 <- c(0, 1, 2, 3, 5)
(op <- poly_orth(x0, norm = TRUE))
List of polynomials:
$P0
0.4472136

$P1
-0.5718628 + 0.2599376*x

$P2
0.5576469 - 0.8387009*x + 0.1650635*x^2

$P3
```

```
-0.3747527 + 2.015118*x - 1.178499*x^2 + 0.1594343*x^3

$P4
0.1468446 - 4.506906*x + 5.672485*x^2 - 2.090088*x^3 + 0.2269973*x^4
fop <- as.function(op)       # Explicit coercion needed for polylist
zapsmall(crossprod(fop(x0)))  # Verify orthonormality
   P0 P1 P2 P3 P4
P0  1  0  0  0  0
P1  0  1  0  0  0
P2  0  0  1  0  0
P3  0  0  0  1  0
P4  0  0  0  0  1
```

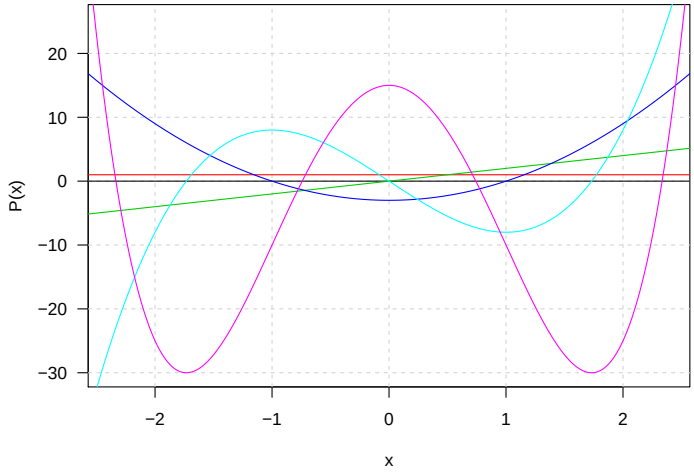
3. *Find the Hermite polynomials up to degree 5 and plot them. Also plot their derivatives and integrals on separate plots.* The Hermite polynomials, like all orthogonal sets, satisfy a 2-term recurrence relation:

$$He_0(x) = 1, \quad He_1(x) = x,$$
$$He_n(x) = xHe_{n-1}(x) - (n-1)He_{n-2}(x), \qquad n = 2, 3, \ldots$$

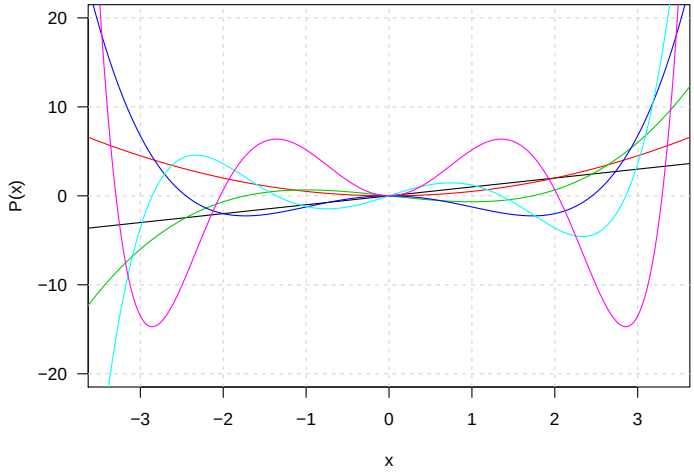
```
x <- polynomial()
He <- polylist(polynomial(1), x)
for (n in 3:6) {
  He[[n]] <- x * He[[n-1]] - (n-2) * He[[n-2]] # R indices start from 1, not 0
}
(He <- setNames(He, paste0("He", seq_along(He) - 1)))
List of polynomials:
$He0
1

$He1
x

$He2
-1 + x^2

$He3
-3*x + x^3

$He4
3 - 6*x^2 + x^4

$He5
15*x - 10*x^3 + x^5
He0 <- poly_orth_general(Hermite, 5)  # using brute computation
sapply(He-He0, function(p) max(abs(coef(p)))) # accuracy check
         He0          He1          He2          He3          He4
0.000000e+00 0.000000e+00 2.058411e-10 1.760654e-08 2.733579e-08
         He5
4.309431e-08
plot(He, lty = "solid")    # plots, with a bit of annotation
He5 <- He[["He5"]]
stat <- solve(deriv(He5))
points(stat, He5(stat), pch = 19)
lines(tangent(He5, stat), limits = cbind(stat-0.5, stat+0.5),
      lty = "dashed", col="black")
```




```
plot(deriv(He), lty = "solid")
```



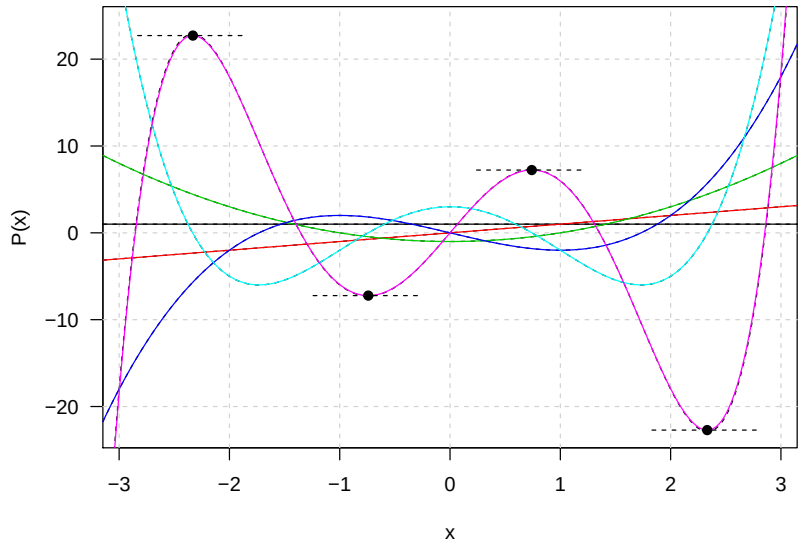

```
plot(integral(He), lty = "solid")
```



4. For weighted discrete orthogonal polynomials the function `poly_orth_general` can be used with the `Discrete` inner product function. E.g. 'approximate' Hermite polynomials may be calculated as follows:

```
Hea <- poly_orth_general(Discrete, degree = 5,
                         x = seq(-5, 5, length.out = 101),
                         w = function(x) exp(-x^2/2), norm = FALSE)
plot(He, lty = "dashed", col = "black") # accurate polynomials
```

```
lines(Hea, lty = "solid")                # discrete approximation
He5 <- Hea[[6]]
stat <- solve(deriv(He5))
points(stat, He5(stat), pch = 19)
lines(tangent(He5, stat), limits = cbind(stat-0.5, stat+0.5),
      lty = "dashed", col="black")
```

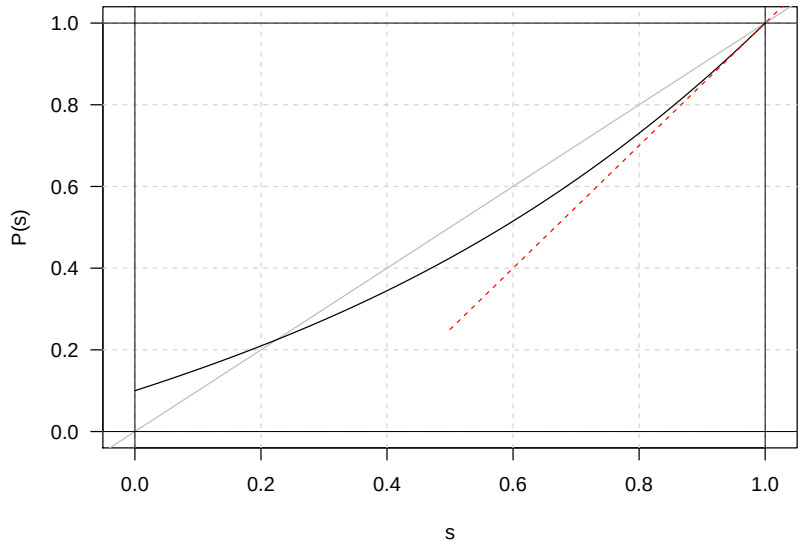


If the weights are all equal, then `poly_orth` suffices and does a slightly faster job than `poly_orth_general`.

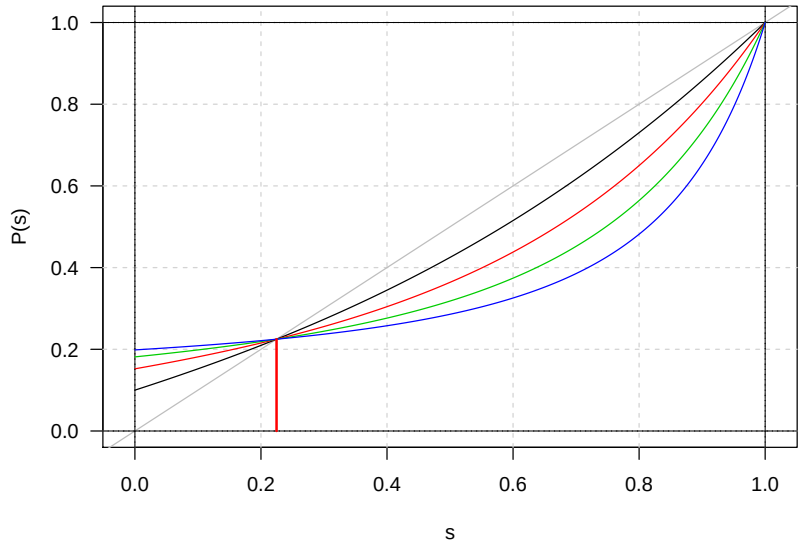
5. A simple branching process. If an organism can have $0,1,2,\ldots$ offspring with probabilities $p_0, p_1, p_2, \ldots$ the generating function for this discrete distribution is defined as $P(s) = p_0 + p_1 s + p_2 s^2 + \ldots$. If only a (known) finite number of offspring is possible, the generating function is a polynomial. In any case, if all offspring themselves reproduce independently according to the same offspring distribution, then the generating function for the size of the second generation can be shown to be $P(P(s))$, and so on. There is a nice collection of results connected with such simple branching processes: in particular the chance of ultimate extinction is the (unique) root between 0 and 1 of the equation $P(s) - s = 0$. Such an equation clearly has one root at $s = 1$, which, if $P(s)$ is a finite degree polynomial, may be "divided out". Also the expected number of offspring for an organism is clearly the slope at $s = 1$, that is, $P'(1)$.

Consider a simple branching process where each organism has at most *three* offspring with probabilities $p_0 = \frac{1}{10}, p_1 = \frac{5}{10}, p_2 = p_3 = \frac{2}{10}$. The following will explore some of its properties without further discussion.

```
P <- polynomial(c(1, 5, 2, 2)/10)
s <- polynomial(c(0, 1))
(mean_offspring <- deriv(P)(1))
[1] 1.5
plot(s, xlim = c(0,1), ylim = c(0,1), xlab = "s", ylab = "P(s)", col = "grey")
abline(h=0:1, v=0:1, lty = "solid", lwd = 0.1)
lines(P, limits = 0:1)
lines(tangent(P, 1), lty = "dashed", col = "red", limits = c(0.5, 1.5))
```




```
# higher generations
plot(s, xlim = c(0,1), ylim = c(0,1), xlab = "s", ylab = "P(s)", col = "grey")
abline(h=0:1, v=0:1, lty = "solid", lwd = 0.1)
lines(P, limits = 0:1)
lines(P(P), col = 2, limits = 0:1)
lines(P(P(P)), col = 3, limits = 0:1)
lines(P(P(P(P))), col = 4, limits = 0:1)
solve(P, s)                 # for the extinction probability
[1] -2.2247449  0.2247449  1.0000000
(ep <- solve((P-s)/(s-1))) # factor our the known zero at s = 1
[1] -2.2247449  0.2247449
ex <- Re(ep[length(ep)])    # extract the appropriate value (may be complex)
segments(ex, 0, ex, P(ex), col = "red", lwd = 2)
abline(h=0:1, v=0:1, lty = "dotted")
```



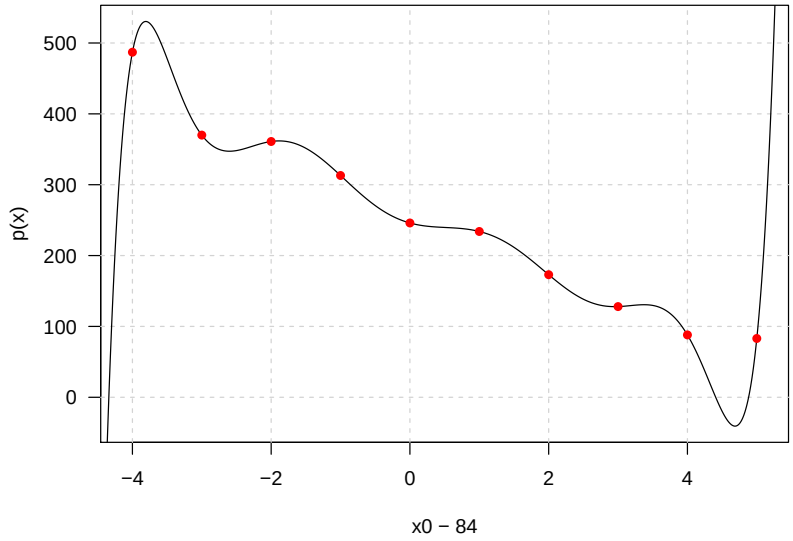
6. Polynomials can be numerically fragile. This can easily lead to surprising numerical problems.

```
x0 <- 80:89
y0 <- c(487, 370, 361, 313, 246, 234, 173, 128, 88, 83)

p <- poly_calc(x0, y0)          # leads to catastropic numerical failure!
p(x0) - y0                      # these should be "close to zero"!
```

```
 [1] 38.5 45.5 44.0 16.0 90.0 96.5 62.0 34.5 -2.5 28.0
```

```
p1 <- poly_calc(x0 - 84, y0)  # changing origin fixes the problem
p1(x0 - 84) - y0              # these are 'close to zero'.
 [1]  7.389644e-12  1.989520e-12  3.410605e-13  0.000000e+00  0.000000e+00
 [6]  5.684342e-14  1.421085e-13 -2.842171e-14 -3.296918e-12 -2.199840e-11
```

```
plot(p1, xlim = c(80, 89) - 84, xlab = "x0 - 84")
points(x0 - 84, y0, col = "red")
```




```
# Can we now write the polynomial in "raw" form?
x <- polynomial()
p0 <- p1(x - 84) # attempting to change the origin back to zero
                 # leads to severe numerical problems again
plot(p0, xlim = c(80, 89))
points(x0, y0, col = "red") # major errors due to finite precision
```