Univariate Polynomials in R

Bill Venables

2019-03-28

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

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.

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.

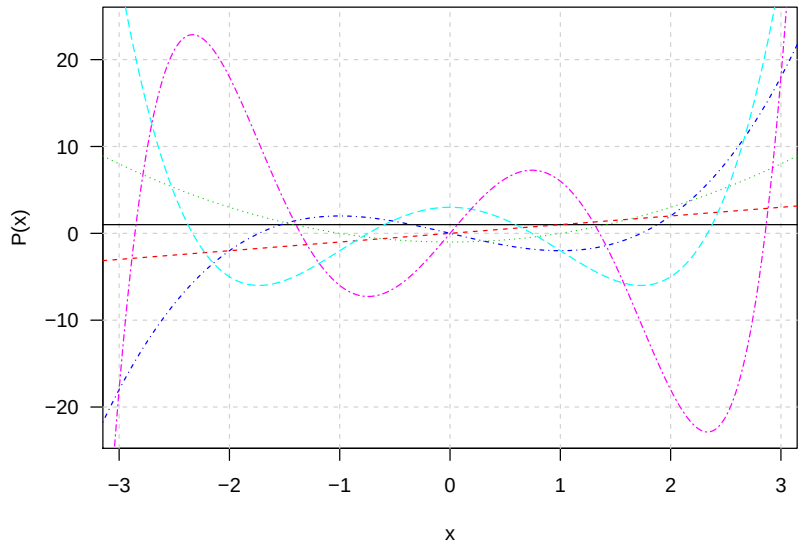
Examples

1. Find the Hermite polynomials up to degree 5 and plot them. Also plot their derivatives and integrals on separate plots.

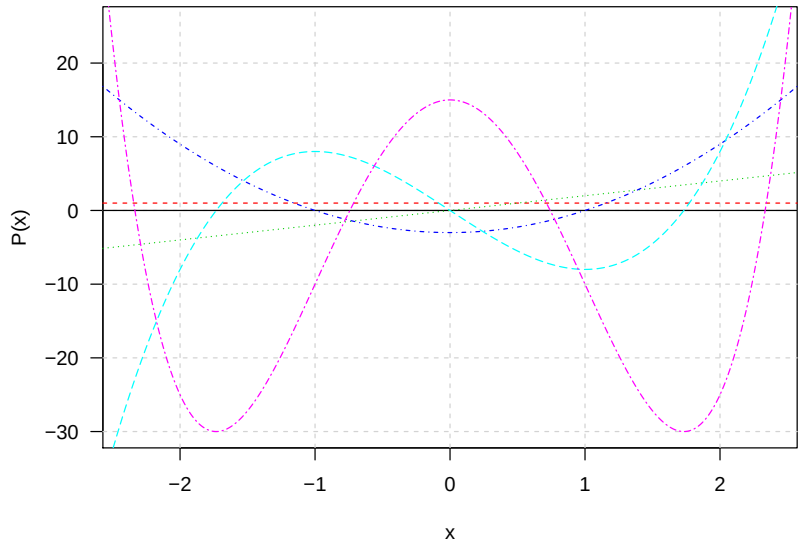
   The polynomials in question satisfy

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

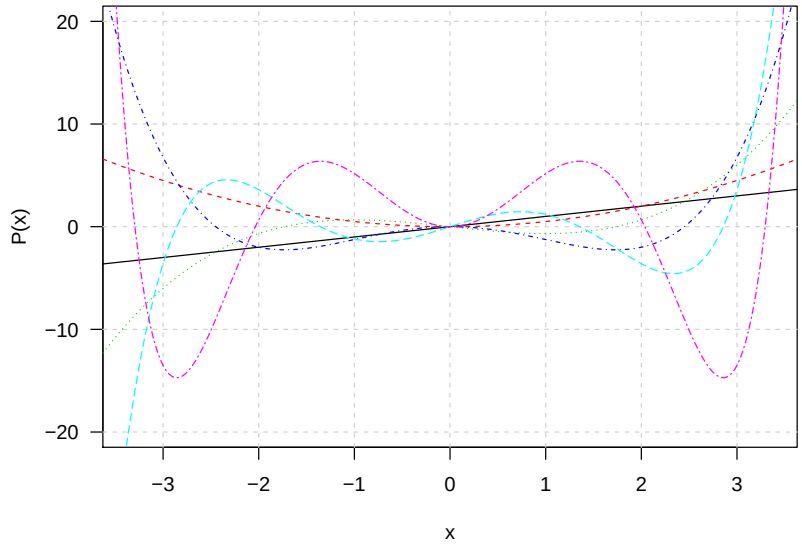
```
He <- polylist(polynom(1), polynom(0:1))
x <- polynom()
for (n in 3:6) {
  He[[n]] <- x * He[[n-1]] - (n-2) * He[[n-2]] ## R indices start from 1, not 0
}
plot(He)
```




```
plot(deriv(He))
```




```
plot(integral(He))
```



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

```
x <- c(0, 1, 2, 3, 5)
(op <- poly.orth(x))
List of polynomials:
[[1]]
0.4472136

[[2]]
-0.5718628 + 0.2599376*x

[[3]]
0.5576469 - 0.8387009*x + 0.1650635*x^2

[[4]]
-0.3747527 + 2.015118*x - 1.178499*x^2 + 0.1594343*x^3

[[5]]
```

```
0.1468446 - 4.506906*x + 5.672485*x^2 - 2.090088*x^3 + 0.2269973*x^4
(fop <- as.function(op))
function (z, ...)
sapply(x, function(p) p(z), ...)
<bytecode: 0x55a2dda61ba8>
<environment: 0x55a2dda61400>
zapsmall(crossprod(fop(x)))  ## Verify orthonormality
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

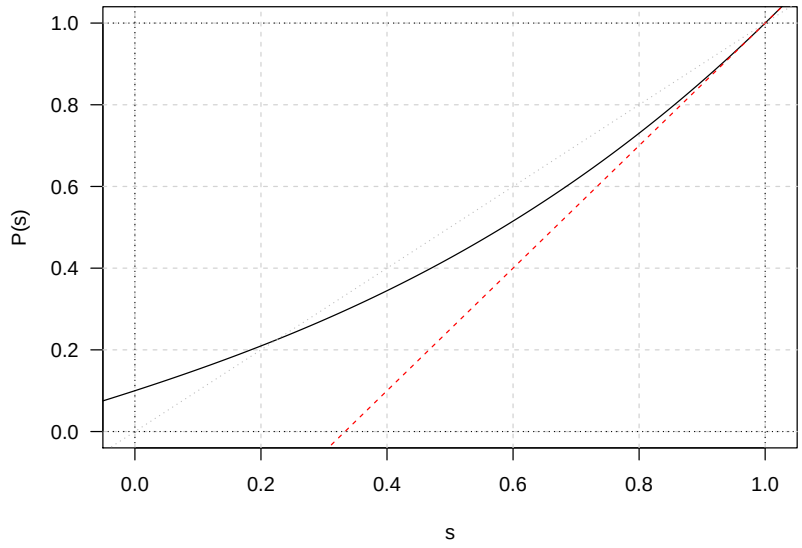
3. Miscellaneous computations using polynomial arithmetic.

```
(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
```

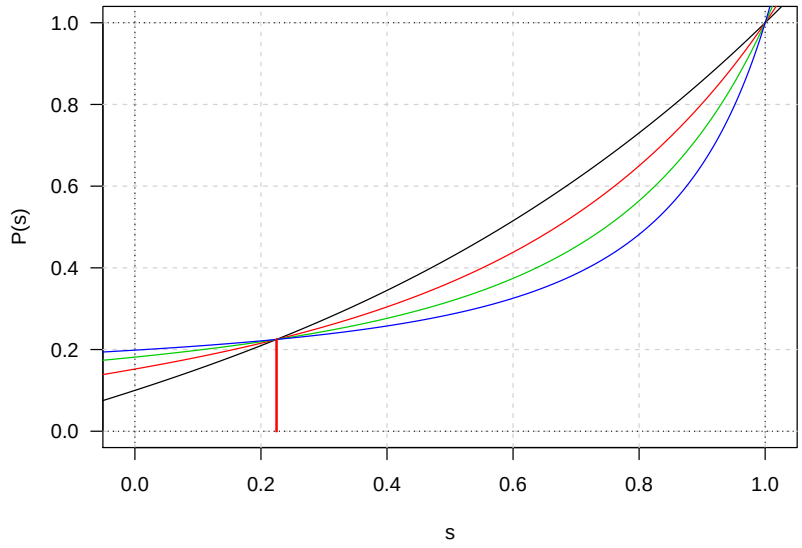
4. A simple branching process. If an organism can have $0, 1, 2, \ldots$ offspring with probabilities $p_0, p_1, p_2, \ldots$ the generating function for this discrete distribution is defined as $P(s) = p_0 + p_1 s + p_2 s^2 + \ldots$. If only a (known) finite number of offspring is possible, the generating 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 zero of $s = 1$, 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 3 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 <- polynom(c(1, 5, 2, 2)/10)
s <- polynom(c(0, 1))
(mean_offspring <- deriv(P)(1))
[1] 1.5
plot(P, xlim = c(0,1), ylim = c(0,1), xlab = "s", ylab = "P(s)")
abline(h=0:1, v=0:1, lty = "dotted")
lines(tangent(P, 1), lty = "dashed", col = "red")
lines(s, lty = "dotted", col = "grey")
```




```
## higher generations
plot(P, xlim = c(0,1), ylim = c(0,1), xlab = "s", ylab = "P(s)")
lines(P(P), col = 2)
lines(P(P(P)), col = 3)
lines(P(P(P(P))), col = 4)
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 <- ep[2]                 ## extract the appropriate value
segments(ex, 0, ex, P(ex), col = "red", lwd = 2)
abline(h=0:1, v=0:1, lty = "dotted")
```



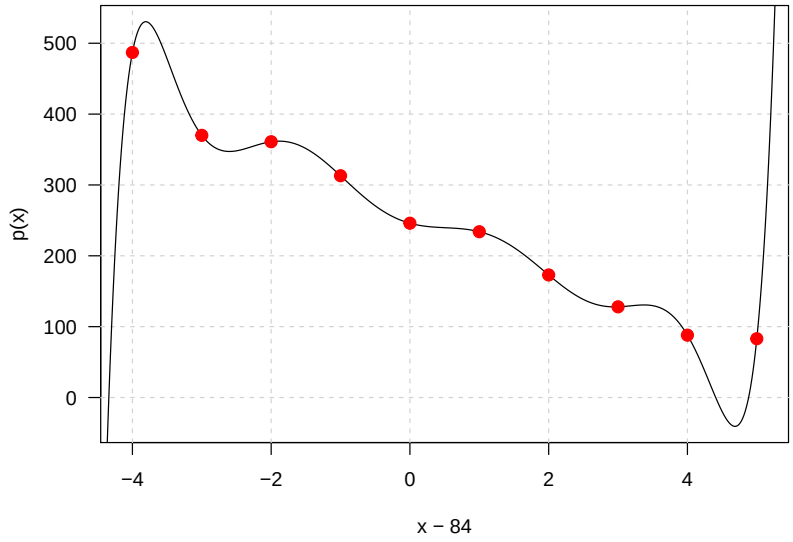
5. Polynomials can be numerically fragile. This can easily lead to surprising numerical problems.

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

p <- poly.calc(x, y)        ## leads to catastropic numerical failure!
p(x) - y                    ## 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(x - 84, y)  ## changing origin fixes the problem
p1(x - 84) - y              ## 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 = "x - 84")
points(x - 84, y, col = "red", cex = 2)
```




```
#### Can we now write the polynomial in "raw" form?
z <- polynom()
p0 <- p1(z - 84) ## attempting to change the origin back to zero
                 ## leads to severe numerical problems again
plot(p0, xlim = c(80, 89))
points(x, y, col = "red", cex = 2) ## major numerical errors due to finite precision
```