"ℚ[i][X]"
Polynomial Module
(with Gaussian Number support)

Rob Hubbard

2006–2007

Contents

1 Introduction **3**
1.1 The Polynomial Module . . . 3
1.2 Scope . . . 3
1.3 Limitations . . . 3
1.4 Notes on this Documentation . . . 3
1.5 Acknowledgements . . . 3
1.6 History . . . 3

2 Polynomials **4**
2.1 Creating Polynomials . . . 4
2.2 Polynomial Deconstruction . . . 5
2.3 Strings . . . 6
2.4 Quoting and Unquoting . . . 9
2.5 Polynomial Tests . . . 9
2.6 Polynomial Arithmetic and Mathematics . . . 11
2.6.1 Basic Arithmetic . . . 11
2.6.2 Division, Divisibility, Factors and Indices . . . 11
2.6.3 Roots and Multiplicity . . . 12
2.6.4 Application and Composition . . . 12
2.6.5 GCD and LCM . . . 13
2.6.6 Conversion to other Forms . . . 13
2.7 DFT . . . 14
2.8 Polynomial Geometrical Transformation . . . 15
2.9 Rebasing . . . 16
2.10 Calculus: Formal Derivation and Anti-derivation . . . 16
2.11 Polynomial Factorisation . . . 17
2.11.1 Galois Theory (is not the whole story) . . . 17
2.11.2 Primes and Factorisation in ℤ and ℚ . . . 17
2.11.3 The Rational Root Theorem . . . 18
2.11.4 Gaussian Primes and Factorisation in ℤ[i] and ℚ[i] . . . 19
2.11.5 'Rational Complex Root Theorem' . . . 20
2.11.6 Root Multiplicities and the Square-free Part . . . 21
2.12 Solution of Polynomials . . . 22
2.12.1 Galois Theory (again) . . . 22
2.12.2 Monic Depression and Polynomials . . . 22

2.12.3 The Linear Case . . . 22
2.12.4 The Quadratic Case . . . 22
2.12.5 The Cubic Case . . . 23
2.12.6 The Quartic Case . . . 24
2.12.7 The Quintic and Higher Degree Cases . . . 25
2.12.8 Direct Solution by Radicals . . . 25
2.12.9 Patterns of Coefficients . . . 25
2.12.10 Root Bounds . . . 27
2.12.11 Root Approximation . . . 29
2.13 Strategy for Finding Roots . . . 29

3 Difference Systems **29**
3.1 Introduction — Difference Tables . . . 29
3.2 Creating Difference Systems . . . 30
3.3 Difference System Deconstruction and Tests . . . 30
3.4 Difference System Arithmetic and Manipulation . . . 30
3.5 Conversion between Polynomials and Difference Systems . . . 30
3.6 Rebasing Difference Systems . . . 31
3.7 Recurrence Relations (Appendix) . . . 31
3.8 Application of Difference Systems (Appendix) . . . 32
3.9 Finite Sums and Finite Differences (of Polynomials) . . . 33
3.9.1 Digression: Finite Sums via Faulhaber's Formula . . . 34
3.10 Operators . . . 36
3.10.1 Digression: Operator Relationships . . . 36
3.10.2 Another Digression: Further Operator Relationships . . . 37

4 Bézier Parameterisations **37**
4.1 Introduction . . . 37
4.2 Creating Béziers . . . 37
4.3 Bézier Functions . . . 38
4.4 Conversion between Polynomials and Béziers . . . 38
4.5 Adjusting the degree of Béziers . . . 38
4.6 Rebasing Béziers . . . 38
4.7 Splitting Béziers . . . 38
4.8 Bézier Calculus . . . 39

This document is version (β)–319 for, and using, `polynomial.q` revision 317, `alg_common.q` revision 0, (`alg_matrix.q` revision 310), and Q 7.7.

It was generated from a pre-processed `polynomial.unpre.tex` revision 319 on June 10, 2007.

1 Introduction

1.1 The Polynomial Module

The module defines types `Polynomial`, `DiffSystem` and `Bezier` for Albert Gräf's equational programming language 'Q'
(`http://q-lang.sourceforge.net/`).
The module is compatible with Q version 7.7 (onwards).

1.2 Scope

This module is mainly, but not entirely, aimed at polynomials over the rational complex numbers $\mathbb{Q}$[i], the integral complex numbers or 'Gaussian integers' $\mathbb{Z}$[i] in addition to the rationals $\mathbb{Q}$ and integers $\mathbb{Z}$.
Some routines are also suitable for polynomials over the complex numbers $\mathbb{C} = \mathbb{R}$[i] and real numbers $\mathbb{R}$.
(There is partial support for non-numeric coefficients.)

1.3 Limitations

This module is aimed only at univariate polynomials, i.e. polynomials in a single indeterminate, rather than multivariate polynomials. Rational functions, i.e. quotients of polynomials, are not supported.
The module is very far from a full CAS (computer algebra system).

1.4 Notes on this Documentation

This document does not describe all the many minor `public` functions in the module.
Some `private` functions such as `view_round` D X, which rounds values to D decimal places (for easy viewing), and `doc_simp` ("simplify for documentation"), which recursively applies such simplifications through a compound object, are used in the examples.
The notation '`function_name` $Params \to (X, Y)$' is sometimes used to highlight that a function returns a **tuple** of values.

1.5 Acknowledgements

Thank you to Albert Gräf, especially for the helpful and thorough answers to my many questions about Q, Q Lex and Q Yacc, which were used, briefly, in version 0.3 (revision 278).

1.6 History

Whilst on holiday in 2006, I was pondering the form of the polynomial factorisation of $\sum_{x=1}^{n} x^k$ for various powers k; specifically whether there is a pattern to the factors. The coefficients of the resulting polynomials are given by 'Faulhaber's formula' [(16) in §3.9.1]. I was curious as to whether these polynomials could more easily be calculated from the factors.
I had no computer available, and was therefore trying to do the calculations by hand.
It occurred to me that I could find the sums in question more easily by using difference tables. (See §3.1.)
I then needed to factorise my results. I was only interested in 'simple' factors — factors with integer or rational coefficients, and probably only linear ones. Equivalently, I needed to find integer or rational roots to the polynomial.
I had, some years ago, studied Galois Theory, and thus had considered that beyond quartic (degree 4) polynomials, in general, little could be done to find the roots except by approximation. The theory also shows that some higher degree polynomials were still susceptible to factorisation 'by radicals' if there were certain patterns in the coefficients.
For example, if the constant coefficient is zero, then zero is a factor and may be immediately factored out. Similarly, if all the (non-zero) terms of the polynomial $p_0 + p_1X^2 + p_2X^4 + \cdots + p_kX^{2k}$, are of an even power of the indeterminate X then the polynomial is a composition of a half-degree polynomial $p_0 + p_1X + p_2X^2 + \cdots + p_kX^k$ with the 'squaring' polynomial X^2.

I began to wonder, though, whether some properties of integral or rational roots could be determined from the coefficients; bounds on them perhaps.

Back at home, some searching and reading exposed the existence of bounds for complex roots. (See §2.12.10.) For rational roots, I would also need to find a bound for the denominator. Then an exhaustive test of all the potential roots in the resulting finite set of possibilities would yield all rational roots.

As luck would have it, I soon found, in the literature, the Rational Root Theorem. That was exactly what I needed. (See §2.11.3.) I generalised the theorem to form the 'Rational Complex Root Theorem'. I don't know whether this is a new result; it is a simple generalisation, but I have not encountered it in the literature. (See §2.11.5.)

Reading around the subject further, I found many other interesting and useful results.

The solutions of the general cubic and quartic polynomials (over $\mathbb{C}$) by radicals that I was previously aware of were, well, complex. The methods are often difficult to apply, difficult to implement as a computer program, difficult to understand in principle, and sometimes contain errors (perhaps just an incorrect sign somewhere).

However, I found very many different solutions in the literature. Amongst them were some real gems: Concise, simple to understand, and easy to apply. (See §2.12.)

I began to write this module, in part, to test some of these ideas and algorithms.

2 Polynomials

The module defines the type `Polynomial`.

2.1 Creating Polynomials

The module allows polynomials to be created in several different ways. The most basic is probably by specifying the coefficients in a list.

`poly_with_coeffs` *CC* — polynomial with given list of coefficients.

This is the 'working form' of the polynomial.

Example 1 *Constructing the polynomial* $\frac{2}{3} + 5\mathrm{i}X + X^3$ *with coefficients.*

```
==> poly_with_coeffs [2%3,5*i,0,1]
poly_lambda '(\X . X^3+(0+:5)*X+2%3)
```

Notice that the coefficients are specified from the constant coefficient to the leading coefficient, and that any intermediate zero coefficients must be given.

The coefficients may be any numbers. However, this module is mainly concerned with polynomials over the rational complex numbers $\mathbb{Q}$[i].

Polynomials may be created in other ways.

`const_poly` *N* — constant polynomial.

`monic_poly_with_root` *R* — monic (linear) polynomial with the given root.

`linear_poly` (*A*, *B*) — linear polynomial $A + BX$; note that the values are passed as a pair.

`poly_term` *N C* — N^{th} term or monomial with coefficient *C*.

`monic_poly_with_roots` *RR* — monic polynomial with the given list or `bag` of roots.

`monic_poly_with_root_mults` *RR* — monic polynomial with the given list of (root, multiplicity) pairs or `dict` from root to multiplicity.

`poly_thru_points` *Points* — polynomial of minimal degree fitting the list of points.

`linear_map_poly` *FromRange ToRange* — linear polynomial *L* mapping *FromRange* = (U, V) to *ToRange* = (X, Y), i.e. s.t. $L(U) = X$ and $L(V) = Y$.

Certain simple polynomials may be constructed: monomials, and monic linear polynomials with a given root. (A "monomial" is the same as a "polynomial term".)

Example 2 *Monomials (terms) and linear polynomials.*

```
==> poly_term 10 4
poly_lambda '(\X . 4*X^10)
==> const_poly 7
poly_lambda '(\X . 7)
==> monic_poly_with_root 10
poly_lambda '(\X . X-10)
==> linear_poly (5,12)
poly_lambda '(\X . 12*X+5)
==> linear_map_poly (-1,1) (0,10)
poly_lambda '(\X . 5*X+5)
```

The roots, either repeated, or given with their multiplicities may be used.

Example 3 *Constructing polynomials with given roots and multiplicities.*

```
==> monic_poly_with_roots [1,2,3]
poly_lambda '(\X . X^3-6*X^2+11*X-6)
==> monic_poly_with_roots [0,0,1,1,1]
poly_lambda '(\X . X^5-3*X^4+3*X^3-X^2)
==> monic_poly_with_root_mults [(0,2),(1,3)]
poly_lambda '(\X . X^5-3*X^4+3*X^3-X^2)
```

A polynomial P may be constructed to fit a given set of points (X_i, Y_i), i.e. $P(X_i) = Yi$. The X_i must all be distinct. The X_i and Y_i may be complex.

Example 4 *Fitting polynomials to points.*

```
==> poly_thru_points [(1,2),(2,4),(3,8),(4,16)]
poly_lambda '(\X . 1%3*X^3-X^2+8%3*X)
==> poly_thru_points [(1,1),(2,4),(3,9),(4,16)]
poly_lambda '(\X . X^2)
==> poly_thru_points [(1,1),(i,i)]
poly_lambda '(\X . X)
```

There are also functions to convert a string to a polynomial (see §2.3), and a quoted expression to a polynomial (see §2.4).

2.2 Polynomial Deconstruction

There are various functions available to extract the coefficients, the degree, and so on.

`degree` *Poly* — degree; `-inf` for the zero polynomial.

`length` *Poly* — the 'length' of the polynomial (degree + 1 for non-zero polynomials).

`coefficients` P — the whole list of coefficients.

`len_coefficients` L P — extract (the lower) L coefficients from the polynomial; L may be any non-negative integer.

`coefficient` N P — the N^{th} coefficient; indexed from 0.

`rev_coefficient` N P — the N^{th} coefficient backwards from the lead coefficient; indexed from 0.

`constant_coefficient` P — the constant coefficient.

`leading_coefficient` P — the leading coefficient.

Example 5 *Finding the coefficients.*

```
==> def P = monic_poly_with_roots [0,0,2,3,5,7];
==> coefficients P; degree P;
[0,0,210,-247,101,-17,1]
6
==> (constant_coefficient P, leading_coefficient P);
(0,1)
==> (coefficient 2 P, rev_coefficient 2 P);
(210,101)
```

There are also functions to convert a polynomial to a string (see §2.3) or a quoted expression (see §2.4).

2.3 Strings

The module includes functions for converting between polynomials and string representations.
These are related to the quoted expression functions described in §2.4.

`INDETERMINATE` — a reference to the default indeterminate string; initially `"X"`, but may be set to e.g. `"t"` or `"Tau"`.

`poly_str` *Poly* — convert a polynomial to a string containing a closed representation in the default indeterminate.

`poly_str_open` *Poly* — convert a polynomial to a string containing an open representation in the default indeterminate.

`poly_str_with` *Ind Poly* — convert a polynomial to a string containing a closed representation in the given indeterminate.

`poly_str_open_with` *Ind Poly* — convert a polynomial to a string containing an open representation in the given indeterminate.

`poly_val` *Str* — convert a string containing a closed representation of a polynomial to a polynomial; the indeterminate is implicitly determined.

`poly_val_open` *Str* — convert a string containing an open representation of a polynomial in the default indeterminate to a polynomial.

`poly_val_open_with` *Ind Str* — convert a string containing an open representation of a polynomial in the given indeterminate to a polynomial.

Closed string representations of polynomials use a notation very much like Q's lambda expressions. The backslash (\) in a string must, however, be escaped as (\\). If the dot (`.`) is followed by a number, then, in order to avoid this being parsed as a float (as with Q lambdas), a space should be inserted in between; alternatively, parentheses (`( ··· )`) could be used.
In difference to Q, the exact powers of the indeterminate use (^) rather than `pow`.
Polynomials may be encoded as strings, and converted back.

Example 6 *Polynomial ↔ String. (Open and closed forms.)*

```
==> def P = poly_with_coeffs [2,-1,0,3+4*i,12.5,1.0,22%7]; P
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> poly_str P
"\\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2"
==> poly_val _
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> poly_str_with "t" P
"\\t . 22%7*t^6+1.0*t^5+12.5*t^4+(3+:4)*t^3-t+2"
```

```
==> poly_val _
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> poly_str_open P
"22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2"
==> poly_val_open _
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> poly_str_open_with "Tau" P
"22%7*Tau^6+1.0*Tau^5+12.5*Tau^4+(3+:4)*Tau^3-Tau+2"
==> poly_val_open_with "Tau" _
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
```

The `poly_val` family of functions may also be used as a constructor, given user-defined strings:

Example 7 *Use as a constructor. (Open and closed forms.)*

```
==> poly_val "\\X.X^2+1"
poly_lambda '(\X . X^2+1)
==> poly_val "\\X.-X^2" //note precedence
poly_lambda '(\X . -X^2)
==> poly_val_open "X^4+1"
poly_lambda '(\X . X^4+1)
==> poly_val_open_with "t" "4*t^3+1"
poly_lambda '(\X . 4*X^3+1)
```

The output from the `poly_str` family of functions should be reasonably simple. A coefficient is usually omitted if it has size `1` and is `exact` (the exception being the constant term). A term usually entirely is omitted if its coefficient is `0` and `exact` (the exception being the zero polynomial).

Example 8 *Simplified output*

```
==> poly_str $ poly_with_coeffs [] //empty; zero
"\\X . 0"
==> poly_str $ poly_with_coeffs [-5] //constant
"\\X . -5"
==> poly_str $ poly_with_coeffs [0,0,2]
"\\X . 2*X^2"
==> poly_str $ poly_with_coeffs [1,0,1] //note how the ones are presented...
"\\X . X^2+1"
==> poly_str $ poly_with_coeffs [0,0,-1]
"\\X . -X^2"
==> poly_str $ poly_with_coeffs [0.0, 0, 1.0] //inexact
"\\X . 1.0*X^2+0.0"
==> poly_str $ poly_with_coeffs [-5,-2,-1,1] //negatives
"\\X . X^3-X^2-2*X-5"
==> poly_str $ poly_with_coeffs [-4%3,12%5] //rationals
"\\X . 12%5*X-4%3"
==> poly_str $ poly_with_coeffs [-4-3*i,12+5*i] //complex
"\\X . (12+:5)*X+(-4 +: -3)"
```

The input format for the `poly_val` family of functions is reasonably flexible. Whitespace and parenthesis are permitted. The ordering of the terms is not important. The indeterminate used in `poly_val` and `poly_val_open_with` may be almost any valid identifier in upper or lower case. Q keywords and the names of the standard library functions are obviously best avoided.

Example 9 *Varying the input.*

```
==> poly_val "\\Phi. 3*Phi^4+Phi+1" //note the space after the .
poly_lambda '(\X . 3*X^4+X+1)
==> poly_val "\\t.t*t*t+-t-1"
```

```
poly_lambda '(\X . X^3-X-1)
==> poly_val "\\t.(1+t)+t^4*3"
poly_lambda '(\X . 3*X^4+X+1)
```

If the string to be converted back to a polynomial is not algebraically well-formed, then the returned expression simply won't be reduced.

Only with `poly_val` is the exact string you pass parsed. With the other functions in the family, you may see a slightly different string.

Example 10 *Syntax errors.*

```
==> poly_val "\\X.X**3" //error: (**) not a valid operator
parse_ERROR (valq "\\X.X**3")
==> poly_val "X^3" //error: open rather than closed
parse_ERROR '(X^3)
==> poly_val "\\t.1" //error: ".1" is parsed as a float
parse_ERROR (valq "\\t.1")
==> poly_val "\\t. 1"; poly_val "\\t.(1)" //two workarounds
poly_lambda '(\X . 1)
poly_lambda '(\X . 1)
==> //the string in the result will differ...
==> poly_val_open_with "x" "x^^4" //error: (^^) not a valid operator
parse_ERROR (valq "\\x . x^^4")
==> poly_val_open "\\X.X^4" //error: closed rather than open
parse_ERROR '(\X . \X . X^4)
```

If you try to parse something that is algebraic but not polynomial, you may get a garbled or otherwise unexpected reply. There are no guarantees about what happens in such cases.

Example 11 *Nonsense?*

```
==> poly_val "\\X.X^X" //eh??
pow (poly_lambda '(\X . X)) (poly_lambda '(\X . X))
==> poly_val "\\Y.X^3" //eh??
pow (parse_ERROR '(\Y . X)) 3
```

The polynomial parser may be able to parse some unexpanded forms of polynomials. This behaviour is not guaranteed, however.

Example 12 *Not a CAS.*

```
==> poly_val "\\t.(2+t)^4" //hmmm...
poly_lambda '(\X . X^4+8*X^3+24*X^2+32*X+16)
```

Use the standard functions `get` and `put` to manipulate `INDETERMINATE`.

Example 13 *Changing the default indeterminate.*

```
==> def P = poly_with_coeffs [2,-3,1]; P
poly_lambda '(\X . X^2-3*X+2)
==> def MyInd = get INDETERMINATE; MyInd
"X"
==> put INDETERMINATE "tau"
()
==> poly_str P
"\\tau . tau^2-3*tau+2"
==> put INDETERMINATE MyInd
()
==> poly_str P
"\\X . X^2-3*X+2"
```

2.4 Quoting and Unquoting

The module includes functions for converting between polynomials and quoted expressions.
These are related to the string functions described in §2.3.

`lambdaq_from_poly` *Poly* — convert a polynomial to a quoted closed representation in the default indeterminate.

`exprq_from_poly` *Poly* — convert a polynomial to a quoted open representation in the default indeterminate.

`lambdaq_from_poly_with` *Ind Poly* — convert a polynomial to a quoted closed representation in the given indeterminate.

`exprq_from_poly_open_with` *Ind Poly* — convert a polynomial to a quoted open representation in the given indeterminate.

`poly_from_lambdaq` *Quoted* — convert a quoted closed representation of a polynomial to a polynomial; the indeterminate is implicitly determined.

`poly_lambda` *Quoted* — `poly_lambda` is just an alias of `poly_from_lambdaq` .

`poly_from_exprq` *Quoted* — convert a quoted open representation of a polynomial in the default indeterminate to a polynomial.

`poly_from_exprq_with` *Ind Quoted* — convert a quoted open representation of a polynomial in the given indeterminate (given as a string) to a polynomial.

`poly_from_exprq_withq` *QInd Quoted* — convert a quoted open representation of a polynomial in the given indeterminate (given as a quoted expression) to a polynomial.

Example 14 *Polynomial ↔ Quoted Expression. (Open and closed forms.)*

```
==> def P = poly_with_coeffs [2,-1,0,3+4*i,12.5,1.0,22%7]; P
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> lambdaq_from_poly P
'(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> poly_from_lambdaq _
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> lambdaq_from_poly_with "t" P
'(\t . 22%7*t^6+1.0*t^5+12.5*t^4+(3+:4)*t^3-t+2)
==> poly_from_lambdaq _
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> exprq_from_poly P
'(22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> poly_from_exprq _
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> def Q = exprq_from_poly_with "Tau" P; Q
'(22%7*Tau^6+1.0*Tau^5+12.5*Tau^4+(3+:4)*Tau^3-Tau+2)
==> poly_from_exprq_with "Tau" Q
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
==> poly_from_exprq_withq 'Tau Q
poly_lambda '(\X . 22%7*X^6+1.0*X^5+12.5*X^4+(3+:4)*X^3-X+2)
```

2.5 Polynomial Tests

The type `Polynomial` may be used as a type guard.
There are various functions for testing for various types of polynomial, in particular, for polynomials with coefficients of a particular type.
These include:

`ispoly` X — is a polynomial.

`iszeropolyval` X — is the 'zero' polynomial.

`isconstpolyval` X — is a 'constant' polynomial.

`ismonicpolyval` X — is a monic polynomial.

`is_intval_poly` X — polynomial with integer-valued ($\mathbb{Z}$) coefficients.

`is_ratval_poly` X — polynomial with rational-valued ($\mathbb{Q}$) coefficients.

`is_realval_poly` X — polynomial with real-valued ($\mathbb{R}$) coefficients.

`is_intcompval_poly` X — polynomial with integral complex-valued ($\mathbb{Z}$[i]) coefficients.

`is_ratcompval_poly` X — polynomial with rational complex-valued ($\mathbb{Q}$[i]) coefficients.

`is_compval_poly` X — polynomial with complex-valued ($\mathbb{C}$) coefficients.

`is_num_poly` X — polynomial with numeric coefficients.

`isexact_poly` X — polynomial with exact coefficients.

`isexact_intval_poly` X — polynomial with exact integer-valued coefficients.

`isexact_ratval_poly` X — polynomial with exact rational-valued coefficients.

`isexact_intcompval_poly` X — polynomial with exact integral complex-valued coefficients.

`isexact_ratcompval_poly` X — polynomial with exact rational complex-valued coefficients.

$LPoly$ `deg_eq_deg` $RPoly$ — efficient test that the degree of a polynomial is equal to that of another; the value may be `-inf` or a non-negative real value.

$LPoly$ `deg_neq_deg` $RPoly$ — efficient test that the degree of a polynomial is not equal to that of another.

$LPoly$ `deg_lt_deg` $RPoly$ — efficient test that the degree of a polynomial is less than that of another.

$LPoly$ `deg_lte_deg` $RPoly$ — efficient test that the degree of a polynomial is less than or equal to that of another.

$LPoly$ `deg_gt_deg` $RPoly$ — efficient test that the degree of a polynomial is greater than that of another.

$LPoly$ `deg_gte_deg` $RPoly$ — efficient test that the degree of a polynomial is greater than or equal to that of another.

`degs_cmp` $LPoly$ $RPoly$ — efficient comparison of the degrees of two polynomials; returns -1 if `degree` LPoly $<$ `degree` RPoly, 0 if `degree` LPoly $=$ `degree` RPoly, and 1 if `degree` LPoly $>$ `degree` RPoly.

$Poly$ `deg_eq` $Value$ — efficient test that the degree of a polynomial is equal to a value; the value may be `-inf` or a non-negative real value.

$Poly$ `deg_neq` $Value$ — efficient test that the degree of a polynomial is not equal to a value.

$Poly$ `deg_lt` $Value$ — efficient test that the degree of a polynomial is less than a value.

$Poly$ `deg_lte` $Value$ — efficient test that the degree of a polynomial is less than or equal to a value.

$Poly$ `deg_gt` $Value$ — efficient test that the degree of a polynomial is greater than a value.

$Poly$ `deg_gte` $Value$ — efficient test that the degree of a polynomial is greater than or equal to a value.

`deg_cmp` $Poly$ $Value$ — efficient comparison of the degree of a polynomial with a value; returns -1 if `degree` Poly $<$ Value, 0 if `degree` Poly $=$ Value, and 1 if `degree` Poly $>$ Value.

$Poly$ `deg_in` (Min, Max) — efficient test that the degree of a polynomial is in a given range (inclusively).

Poly `deg_btw` (Inf, Sup) — efficient test that the degree of a polynomial is between two values (exclusively).

If you require other properties, then it is possible to map a predicate over the coefficients either 'universally' using the quantifier `poly_all_coeffs` or 'existentially' using the quantifier `poly_some_coeff`.

Given a scalar predicate `Pred`, these functions produce a polynomial predicate, which, for a degree N polynomial P quantifies over its $N+1$ terms, from the constant term to the leading term, and **including** `exact 0` terms.

Beware that, for **any** predicate `Pred` (and in particular its own converse)

`poly_all_coeffs` *Pred* `ZeroPoly` = `true` and

`poly_some_coeff` *Pred* `ZeroPoly` = `false`,

where `ZeroPoly` is the zero or empty polynomial described in §2.6.1.

2.6 Polynomial Arithmetic and Mathematics

2.6.1 Basic Arithmetic

There are some pre-defined polynomials.

`ZeroPoly` — the additive identity.

`OnePoly` — the multiplicative identity.

`IndetPoly` — the compositional or functional identity, consisting of the indeterminate only.

Example 15 *Pre-defined polynomials.*

```
==> ZeroPoly; OnePoly; IndetPoly;
poly_lambda '(\X . 0)
poly_lambda '(\X . 1)
poly_lambda '(\X . X)
```

The operators `(+)`, `(-)` and `(*)` are defined. The operator `pow` is defined for non-negative integral right-hand operands. The operators `(/)` and `(%)` are defined only for division by a scalar. For `(div)` and `(mod)`, see §2.6.2.

Example 16 *Addition, subtraction and multiplication.*

```
==> def A = poly_with_coeffs [4,0,0,0,1]
==> def B = poly_with_coeffs [3,2]
==> A+B; A-B; A*B; pow A 3
poly_lambda '(\X . X^4+2*X+7)
poly_lambda '(\X . X^4-2*X+1)
poly_lambda '(\X . 2*X^5+3*X^4+8*X+12)
poly_lambda '(\X . X^12+12*X^8+48*X^4+64)
==> A/2; A%2
poly_lambda '(\X . 0.5*X^4+0.0*X^3+0.0*X^2+0.0*X+2.0)
poly_lambda '(\X . 1%2*X^4+2)
```

The relational operators `(=)` and `(<>)` are also defined for `Polynomial`. (The other inequality operators are not defined.)

The function `zeroed` adds a constant to a polynomial so that it passes through the origin, i.e. so that $P(0) = 0$.

2.6.2 Division, Divisibility, Factors and Indices

Division is defined too, via the function `divide` and operators `(div)` and `(mod)`.

`divide` N $D \rightarrow (Q, R)$ — Quotient Q and remainder `R`.

Example 17 *Division.*

```
==> def (Q,R) = divide A B; (Q,R)
(poly_lambda '(\X . 1%2*X^3-3%4*X^2+9%8*X-27%16),poly_lambda '(\X . 145%16))
==> A = B * Q + R
true
==> A div B
poly_lambda '(\X . 1%2*X^3-3%4*X^2+9%8*X-27%16)
==> A mod B
poly_lambda '(\X . 145%16)
```

Divisibility of *factors* (or *divisors*) may be tested with the `(divides)` operator.

X `divides` Y — whether $X|Y$.

`index_of_factor` P F — the index of a (potential) factor F in a polynomial P.

`index_factor` P F $\rightarrow (F, I)$ — pair the index I of a factor F in a polynomial P.

`indices_multiplicities` P FF — map of `index_factor` P over a list of factors FF.

The index of a factor F in a polynomial P is given by `index_of_factor` P F; if the index is 0, then F is not a factor of P.

2.6.3 Roots and Multiplicity

Roots may be tested with the `(has_root)` operator.

P `has_root` X — whether $P(X) = 0$.

`multiplicity_of_value` P X — the multiplicity of a value (potential root) X in a polynomial P.

`value_multiplicity` P X $\rightarrow (X, M)$ — pair the multiplicity M of a value X in a polynomial P.

`value_multiplicities` P XX — map of `value_multiplicities` P over a list of values XX.

The multiplicity of a root X in a polynomial P is given by `multiplicity_of_value` P X. In fact, the multiplicity of any value may be tested. The value is a *root* if its multiplicity is ≥ 1; it is a *multiple root* (or *repeated root*) if its multiplicity is > 1.

2.6.4 Application and Composition

Polynomials may be applied with the `apply` function and composed with the `(.)` operator.

`apply` P X — $P(X)$.

Example 18 *Application and composition.*

```
==> def Cube = poly_with_coeffs [0,0,0,1]
==> def Succ = poly_with_coeffs [1,1]
==> apply Cube 10; apply Succ 10
1000
11
==> Cube . Succ; Succ . Cube
poly_lambda '(\X . X^3+3*X^2+3*X+1)
poly_lambda '(\X . X^3+1)
```

2.6.5 GCD and LCM

The greatest common divisor (GCD) and least common multiple (LCM) functions are extended to polynomials. (The "greatest common divisor" is also known as the "highest common factor (HCF)".)

These are not dependent on the GCD and LCM of the coefficient type. Specifically, these functions may even be applied to polynomials with coefficients over $\mathbb{R}$ or $\mathbb{C}$, over which the GCD and LCM are not defined. However, the results tend to be poor due to floating-point rounding errors (as if, for example, one polynomial has a root 5.001 and the other has a root 4.999).

Example 19 *GCD and LCM of polynomials with integral coefficients.*

```
==> def A = 2 * monic_poly_with_roots [0,1,-3]; A
poly_lambda '(\X . 2*X^3+4*X^2-6*X)
==> def B = monic_poly_with_roots [0,1,4]; B
poly_lambda '(\X . X^3-5*X^2+4*X)
==> gcd A B; lcm A B
poly_lambda '(\X . X^2-X)
poly_lambda '(\X . 2*X^4-4*X^3-22*X^2+24*X)
```

Example 20 *GCD and LCM of polynomials with rational coefficients.*

```
==> def A = 2 * monic_poly_with_roots [0,1,-1%3]; A
poly_lambda '(\X . 2*X^3-4%3*X^2-2%3*X)
==> def B = monic_poly_with_roots [0,1,1%4]; B
poly_lambda '(\X . X^3-5%4*X^2+1%4*X)
==> gcd A B; lcm A B
poly_lambda '(\X . X^2-X)
poly_lambda '(\X . 2*X^4-11%6*X^3-1%3*X^2+1%6*X)
```

Example 21 *GCD and LCM of polynomials with Gaussian coefficients.*

```
==> def A = 2 * monic_poly_with_roots [0, 1*i, -3]; A
poly_lambda '(\X . 2*X^3+(6 +: -2)*X^2+(0 +: -6)*X)
==> def B = monic_poly_with_roots [0, 1*i, 4]; B
poly_lambda '(\X . X^3+(-4 +: -1)*X^2+(0+:4)*X)
==> gcd A B; lcm A B
poly_lambda '(\X . X^2+(0 +: -1)*X)
poly_lambda '(\X . 2*X^4+(-2 +: -2)*X^3+(-24+:2)*X^2+(0+:24)*X)
```

2.6.6 Conversion to other Forms

`monic_parts` $P \to (C, M)$ — decomposition into scalar value C and monic polynomial M.

`monic_part` P — the equivalent monic polynomial.

`over_int_parts` $P \to (Q, Z)$ — decomposition of rational polynomial P into rational value Q and minimal integral polynomial Z with positive leading coefficient; defined only for polynomials with `exact` rational coefficients.

`over_int_part` P — the equivalent integral polynomial; defined only for polynomials with `exact` rational coefficients.

`over_intcomp_parts` $P \to (Q, Z)$ — decomposition of complex rational polynomial P into complex rational value Q and minimal integral complex polynomial Z with 'positive' leading coefficient; defined only for polynomials with `exact` rational complex coefficients.

`over_intcomp_part` P — the equivalent integral complex polynomial; defined only for polynomials with `exact` rational complex coefficients.

See also §2.11.6 regarding the square-free part.

2.7 DFT

The Discrete Fourier Transform (DFT) for polynomials is available (`dft` N P), as is its inverse (`inv_dft` N P).
The order-N DFT is essentially the value of the polynomial at the N N^{th} roots of unity, which lie on the 'unit circle', the circle of radius 1 centred at 0 in the complex plane.
The DFT of a polynomial is encoded as another polynomial.
The order-1, order-2 and order-4 DTFs use exact arithmetic where possible.

`dft` N P — order-N DFT.

`inv_dft` N P — order-N inverse DFT.

Example 22 *DFT.*

```
==> def P = poly_with_coeffs [0,2,0,1]; degree P
3

==> dft 4 P
poly_lambda '(\X . (0+:1)*X^3-3*X^2+(0 +: -1)*X+3)
==> dft 4 _ // it can be applied again
poly_lambda '(\X . 8*X^3+4*X)
==> inv_dft 4 _
poly_lambda '(\X . (0+:1)*X^3-3*X^2+(0 +: -1)*X+3)
==> inv_dft 4 _ // should equal P
poly_lambda '(\X . X^3+2*X)

==> def D = dft 5 P; doc_simp D
poly_lambda '(\X . (-0.191+:1.314)*X^4+(-1.309+:2.127)*X^3+(-1.309 +: -2.127)*X^
2+(-0.191 +: -1.314)*X+3)
==> def I = inv_dft 5 D; doc_simp I; degree I
poly_lambda '(\X . 1.0*X^3+0.0*X^2+2.0*X+0.0)
4
```

Example 23 *The order of the DFT should be at least the 'length' of the polynomial.*

```
==> def P = poly_with_coeffs [3,0,2,0,1]; length P; P
5
poly_lambda '(\X . X^4+2*X^2+3)
==> dft 4 _
poly_lambda '(\X . 2*X^3+6*X^2+2*X+6)
==> inv_dft 4 _ // doesn't equal P
poly_lambda '(\X . 2*X^2+4)
```

An alternative DFT is also available (`dft_alt` N P and `inv_dft_alt` N P). The determinant of the matrix corresponding to `dft` is N, to `inv_dft` is $1/N$, to `dft_alt` and `inv_dft_alt` is 1.
The DFT is a little like a 'logarithm for polynomials' mapping the ordinary 'convolving' polynomial product `(*)` to a component-wise 'dot product' `dot_prod`.

Example 24 *Polynomial product via DFT. Order 8 used. (Order 6 would have been sufficient.)*

```
==> def A = poly_with_coeffs [3,0,1]; A; degree A
poly_lambda '(\X . X^2+3)
2
==> def B = poly_with_coeffs [2,0,0,1]; B; degree B
poly_lambda '(\X . X^3+2)
3
==> def DA = dft 8 A; doc_simp DA
poly_lambda '(\X . (3.0+:1.0)*X^7+2.0*X^6+(3.0 +: -1.0)*X^5+4.0*X^4+(3.0+:1.0)*X
^3+2.0*X^2+(3.0 +: -1.0)*X+4)
```

```
==> def DB = dft 8 B; doc_simp DB
poly_lambda '(\X . (1.293+:0.707)*X^7+(2.0 +: -1.0)*X^6+(2.707+:0.707)*X^5+1.0*X
^4+(2.707 +: -0.707)*X^3+(2.0+:1.0)*X^2+(1.293 +: -0.707)*X+3)
==> def D = dot_prod DA DB; doc_simp D
poly_lambda '(\X . (3.172+:3.414)*X^7+(4.0 +: -2.0)*X^6+(8.828 +: -0.586)*X^5+4.
0*X^4+(8.828+:0.586)*X^3+(4.0+:2.0)*X^2+(3.172 +: -3.414)*X+12)
==> def I = inv_dft 8 D; doc_simp I; degree I
poly_lambda '(\X . 1.0*X^5+0.0*X^4+3.0*X^3+2.0*X^2+0.0*X+6.0)
7
==> A * B; degree _ // for comparison
poly_lambda '(\X . X^5+3*X^3+2*X^2+6)
5
```

(Any slight inaccuracies are due to the usual rounding errors in the floating point arithmetic.)

2.8 Polynomial Geometrical Transformation

There are some functions for applying geometrical linear transformations to polynomials.

`poly_translate` $XT\ YT\ P$ — translation.

`poly_scale` $XS\ YS\ P$ — scaling and/or reflection.

These functions transform the curve along the 'input'- or 'X'-axis and/or along the 'output'- or 'Y'-axis. They are related to the 'rebasing' functions of §2.9.

Example 25 *Translation.*

```
==> def P = poly_with_coeffs [1,1,2]
==> map (rat_simplify . (apply P)) (nums 0 4)
[1,4,11,22,37]
==> poly_translate 2 0 P
poly_lambda '(\X . 2*X^2-7*X+7)
==> map (rat_simplify . (apply _)) (nums 0 4)
[7,2,1,4,11]
==> poly_translate 0 10 P
poly_lambda '(\X . 2*X^2+X+11)
==> map (rat_simplify . (apply _)) (nums 0 4)
[11,14,21,32,47]
```

Example 26 *Reflection and scaling.*

```
==> poly_scale (-1) 1 P
poly_lambda '(\X . 2*X^2-X+1)
==> map (rat_simplify . (apply _)) (nums 0 4)
[1,2,7,16,29]
==> poly_scale 1 (-1) P
poly_lambda '(\X . -2*X^2-X-1)
==> map (rat_simplify . (apply _)) (nums 0 4)
[-1,-4,-11,-22,-37]
==> poly_scale 2 1 P
poly_lambda '(\X . 1%2*X^2+1%2*X+1)
==> map (rat_simplify . (apply _)) (nums 0 4)
[1,2,4,7,11]
==> poly_scale 1 2 P
poly_lambda '(\X . 4*X^2+2*X+2)
==> map (rat_simplify . (apply _)) (nums 0 4)
[2,8,22,44,74]
```

The 'axes' may be complex, thus the X-/Y-axes should not be confused with the real and imaginary parts of each potentially complex axis.

Example 27 *Imaginary translation.*

```
==> poly_translate i 0 P
poly_lambda '(\X . 2*X^2+(1 +: -4)*X+(-1 +: -1))
==> map (ratcomp_simplify . (apply _)) (nums 0 4)
[-1 +: -1,2 +: -5,9 +: -9,20 +: -13,35 +: -17]
==> poly_translate 0 i P
poly_lambda '(\X . 2*X^2+X+(1+:1))
==> map (ratcomp_simplify . (apply _)) (nums 0 4)
[1+:1,4+:1,11+:1,22+:1,37+:1]
```

2.9 Rebasing

There are functions to `rebase` and `rescale` a `Polynomial`. These adjust the input and output axes between two intervals or between an interval and the unit interval.

`rebase` $FromRange$ $ToRange$ P — Find 'rebased' polynomial Q such that $P(F) = Q(T)$ for ranges $FromRange = (U, V)$, $ToRange = (X, Y)$; note that $FromRange$ and $ToRange$ are intervals expressed as pairs.

`debase` $FromRange$ P — Rebase from the given interval to the unit interval $(0, 1)$.

`enbase` $ToRange$ P — Rebase from the unit interval $(0, 1)$ to the given interval.

`rescale` $FromRange$ $ToRange$ P — Find 'rescaled' polynomial Q such that $P(X) = F \rightarrow Q(X) = T$ for (endpoints of) ranges $FromRange = (U, V)$, $ToRange = (X, Y)$.

`descale` $FromRange$ P — Rescale from the given interval to the unit interval $(0, 1)$.

`enscale` $ToRange$ P — Rescale from the unit interval $(0, 1)$ to the given interval.

Example 28 *Rebase and rescale.*

```
==> def P = poly_thru_points [(0,0), (5,10), (10,5)]; P
poly_lambda '(\X . -3%10*X^2+7%2*X)
==> def R = rebase (0,10) (-100,100) P; R
poly_lambda '(\X . -3%4000*X^2+1%40*X+10)
==> map (apply R) [-100, 0, 100]
[0%1,10%1,5%1]
==> def S = rescale (0,10) (-100,100) P; S
poly_lambda '(\X . -6*X^2+70*X-100)
==> map (apply S) [0, 5, 10]
[-100,100,0]
```

These functions are related to the geometrical transformation functions of §2.8.

2.10 Calculus: Formal Derivation and Anti-derivation

The formal derivative and anti-derivative may be calculated. (The terms "differentiate" and "differentiation" are sometimes incorrectly used in place of "derive" and "derivation". Another term for "anti-derivative" is "integral".)

`derive` P — Polynomial formal derivative of P.

`derive_pow` N P — N^{th} formal derivative of P.

`derivations` P — List of successive derivatives of P.

`antiderive` K P — Indefinite polynomial formal anti-derivative with constant K.

`zeroed_antiderive` P — Indefinite polynomial formal anti-derivative Q s.t. $Q(0) = 0$.

`definite_antiderive` A B P — Definite polynomial formal anti-derivative; this is a scalar rather than a polynomial.

`ldefinite_antiderive` A P — Semi-definite polynomial formal anti-derivative with the left bound fixed; this is a polynomial.

`rdefinite_antiderive` B P — Semi-definite polynomial formal anti-derivative with the right bound fixed; this is a polynomial.

Example 29 *Formal calculus.*

```
==> def P = poly_with_coeffs [1,0,3,0,1]
==> derive P
poly_lambda '(\X . 4*X^3+6*X)
==> antiderive 7 P
poly_lambda '(\X . 1%5*X^5+X^3+X+7)
==> zeroed_antiderive P
poly_lambda '(\X . 1%5*X^5+X^3+X)
==> definite_antiderive 10 20 P;
627010%1
==> apply (ldefinite_antiderive 10 P) 20
627010%1
==> apply (rdefinite_antiderive 20 P) 10
627010%1
```

2.11 Polynomial Factorisation

The polynomial module has some functions that try quite hard to factorise polynomials.

2.11.1 Galois Theory (is not the whole story)

If you have studied 'Galois Theory' in algebra, then you might consider that it is only practical to factorise linear, quadratic, cubic and quartic (biquadratic) polynomials. However, in some cases we can do somewhat better.

For notes on the solution by radicals of polynomials up to quartic, see §2.12.

2.11.2 Primes and Factorisation in $\mathbb{Z}$ and $\mathbb{Q}$

Integers ($\in \mathbb{Z}$) may be uniquely factorised into a unit (sign) and powers of primes.

The units in $\mathbb{Z}$ are $+1$ and -1.

Rational numbers ($\in \mathbb{Q}$) may be similarly factorised if we allow negative powers of primes.

`factorise` $X \to (U, PP)$ — factorisation of an integer or rational number X into scalar unit U and positive integer factors PP with indices.

Example 30 *Factorisation for $\mathbb{Z}$ and $\mathbb{Q}$.*

```
==> factorise 60
(1,[(5,1),(3,1),(2,2)])
==> factorise (-26%7)
(-1,[(13,1),(7,-1),(2,1)])
==> factorise 1 // a unit
(1,[])
```

The right-hand elements of the pairs are the indices of the factors.

2.11.3 The Rational Root Theorem

The 'Rational Root Theorem' states that for a polynomial $P(X) = P_0 + P_1X + P_2X^2 + \cdots + P_KX^K$ with the $P_I \in \mathbb{Z}$, i.e. the coefficients are integers, then all rational roots $\pi_J \in \mathbb{Q}$ must be of the form $\pi_J = N_J/D_J$ with $N_J, D_J \in \mathbb{Z}$ where each $N_J|P_0$ and $D_J|P_K$. The theorem does not say anything about how many of the roots will be rational. If $P_0 = 0$, then we gain no useful information about the N_J; however, then we know that 0 is a root, thus may factor that out (possibly more than once) to obtain another polynomial for which $P_0 \neq 0$. Given a polynomial with rational coefficients ($\in \mathbb{Q}$), then we can multiply by a constant to remove the coefficients' denominators. The theorem gives the best information if the integer coefficients are jointly coprime (i.e. do not all share a common factor).

`poly_rat_root_subset` P — the subset of roots that are rational.

`monic_rat_factor_subset` P — the subset of monic factors for the rational roots.

`over_int_factor_subset` P — the subset of integral factors for the rational roots.

`poly_rat_roots_rem` $Q \to (RR, P)$ — rational roots RR with multiplicities and remainder polynomial P.

`monic_rat_factorise` $Q \to (C, MM)$ — factorisation into a scalar value C and monic factors MM.

`over_int_factorise` $P \to (Q, ZZ)$ — factorisation into a scalar value Q and integral polynomial factors ZZ (only for rational polynomials P).

`multiplicity_product` LL — takes a list of polynomials and indices and forms the product polynomial.

Example 31 *Finding rational roots of a rational polynomial.*

```
==> def P = (monic_poly_with_roots [1,4,7%12]) * (poly_with_coeffs [4,0,2]);
==> P; degree P; monic_part P; over_int_part P;
poly_lambda '(\X . 2*X^5-67%6*X^4+107%6*X^3-27*X^2+83%3*X-28%3)
5
poly_lambda '(\X . X^5-67%12*X^4+107%12*X^3-27%2*X^2+83%6*X-14%3)
poly_lambda '(\X . 12*X^5-67*X^4+107*X^3-162*X^2+166*X-56)

==> poly_rat_root_subset P;
[7%12,1,4]
==> monic_rat_factor_subset P;
[poly_lambda '(\X . X^2+2),poly_lambda '(\X . X-4),poly_lambda '(\X . X-1),poly_
lambda '(\X . X-7%12)]
==> over_int_factor_subset P;
[poly_lambda '(\X . X^2+2),poly_lambda '(\X . X-4),poly_lambda '(\X . X-1),poly_
lambda '(\X . 12*X-7)]

==> poly_rat_roots_rem P;
([(4,1),(1,1),(7%12,1)],poly_lambda '(\X . 2*X^2+4))
==> monic_rat_factorise P;
(2,[(poly_lambda '(\X . X-7%12),1),(poly_lambda '(\X . X-1),1),(poly_lambda '(\X
 . X-4),1),(poly_lambda '(\X . X^2+2),1)])
==> over_int_factorise P;
(1%6,[(poly_lambda '(\X . X^2+2),1),(poly_lambda '(\X . X-4),1),(poly_lambda '(\
X . X-1),1),(poly_lambda '(\X . 12*X-7),1)])
```

Example 32 *Finding rational roots of a complex rational polynomial.*

```
==> def P = (monic_poly_with_roots [1,1,4,7%12,i,4+3*i]);
==> P; degree P;
poly_lambda '(\X . X^6+(-127%12 +: -4)*X^5+(215%6+:91%3)*X^4+(-79%2 +: -229%3)*X
^3+(11%6+:87)*X^2+(221%12 +: -139%3)*X+(-7+:28%3))
6
```

```
==> poly_rat_root_subset P;
[7%12,1,4]
==> monic_rat_factor_subset P;
[poly_lambda '(\X . X^3+(-5 +: -4)*X^2+(1+:8)*X+(3 +: -4)),poly_lambda '(\X . X-
4),poly_lambda '(\X . X-1),poly_lambda '(\X . X-7%12)]
==> over_int_factor_subset P;
[poly_lambda '(\X . X-4),poly_lambda '(\X . X-1),poly_lambda '(\X . 12*X-7)]

==> poly_rat_roots_rem P;
([(4,1),(1,2),(7%12,1)],poly_lambda '(\X . X^2+(-4 +: -4)*X+(-3+:4)))
==> monic_rat_factorise P;
(1,[(poly_lambda '(\X . X-7%12),1),(poly_lambda '(\X . X-1),2),(poly_lambda '(\X
 . X-4),1),(poly_lambda '(\X . X^2+(-4 +: -4)*X+(-3+:4)),1)])
```

(The second numbers in the pairs are the multiplicities.)

The proof of the Rational Root Theorem is so simple that it is worth giving here.
Suppose $P(X) = P_0 + P_1X + P_2X^2 + \cdots + P_KX^K \in \mathbb{Z}[X]$, and that $Q = N/D \in \mathbb{Q}$ is a root where $N, D \in \mathbb{Z}$ and N and D are coprime, that is, the GCD $(N, D) = 1$. $D \neq 0$.
WLOG, suppose that $P_0 \neq 0$. (Otherwise $N|0 = P_0$ for any N.) Then, $Q \neq 0$ and $N \neq 0$.
Similarly WLOG, suppose that $P_K \neq 0$. (Otherwise $D|0 = P_K$ for any D.)
Then $P(Q) = P_0 + P_1(N/D) + P_2(N/D)^2 + \cdots + P_K(N/D)^K = 0$.
(Multiply through by D^K) $P_0D^K + P_1ND^{K-1} + P_2N^2D^{K-2} + \cdots + P_KN^K = 0$
(Rearrange) $P_0D^K = N(-P_1D^{K-1} - P_2ND^{K-2} - \cdots - P_KN^{K-1})$.
Thus, $N|P_0D^K$, and since $(N, D) = 1$, we have $N|P_0$.
(Rearrange again) $P_KN^K = D(-P_0D^{K-1} - P_1ND^{K-2} - \cdots - P_{K-1}N^{K-1})$
Thus $D|P_KN^K$, and since $(N, D) = 1$, we have $D|P_K$.

2.11.4 Gaussian Primes and Factorisation in $\mathbb{Z}$[i] and $\mathbb{Q}$[i]

A Gaussian integer is an integral complex number ($\in \mathbb{Z}$[i]), i.e. complex number with integer parts.

A Gaussian integer is said to be 'positive' if its real part is positive, and either the absolute value of the imaginary part is less than the real part, or the imaginary part itself is equal to the real part (and therefore also positive). Thus, essentially they are those Gaussian integers within the 'cone' extending to the right of 0 (the origin), bounded above by a 45° line (inclusively) and below by the −45° line (exclusively).

Like the integers, the Gaussian integers form a UFD — unique factorisation domain.

The units of the UFD $\mathbb{Z}$[i] are $+1, -1, +\mathrm{i}$ and $-\mathrm{i}$.

If an integer is a Gaussian prime, then it is an ordinary prime. However, an integer that is an ordinary prime may not be a Gaussian prime. Gaussian factorisation is 'finer' than ordinary factorisation.

(Really, we should speak of irreducibility rather than primality. A non-zero non-unit N is *prime* if $[N|AB \Rightarrow N|A$ or $N|B]$; *composite* otherwise. A non-zero non-unit N is *irreducible* if $[N = AB \Rightarrow A$ is a unit or B is a unit]; *reducible* otherwise. In an ID (integral domain), every prime element is irreducible. In a UFD (which is also an ID), every irreducible is prime.)

For example, 3 is a Gaussian prime. However, 5 is not a Gaussian prime, since $5 = (2 + \mathrm{i})(2 - \mathrm{i})$

`gaussian_factorise` $N \rightarrow (U, PP)$ — factorisation of an integer or rational complex number N into scalar unit U and 'positive' integer complex factors PP with indices.

Example 33 *Primes and Gaussian primes.*

```
==> factorise 3; factorise 5; factorise (-13);
(1,[(3,1)])
(1,[(5,1)])
(-1,[(13,1)])
```

```
==> factorise 60;
(1,[(5,1),(3,1),(2,2)])

==> gaussian_factorise 3; gaussian_factorise 5; gaussian_factorise (-13);
(1,[(3,1)])
(1,[(2 +: -1,1),(2+:1,1)])
(-1,[(3 +: -2,1),(3+:2,1)])
==> gaussian_factorise 60;
(-1,[(3,1),(2 +: -1,1),(2+:1,1),(1+:1,4)])

==> gaussian_factorise (4+3*i); //complex
(0+:1,[(2 +: -1,2)])
==> gaussian_factorise ((4+3*i)%13); //rational complex
(0+:1,[(3 +: -2,-1),(3+:2,-1),(2 +: -1,2)])

==> gaussian_factorise 1; //a unit
(1,[])
```

The factorisation is into 'positive' Gaussian primes, and possibly a unit.

2.11.5 'Rational Complex Root Theorem'

The Rational Root Theorem may be generalised. (I have not seen the generalisation in the literature.)
Thus the 'Rational Complex Root Theorem' states that for a polynomial $P(X) = P_0 + P_1X + P_2X^2 + \cdots + P_KX^K$ with the $P_I \in \mathbb{Z}$[i], i.e. the coefficients are Gaussian integers, then all rational roots $\pi_J \in \mathbb{Q}$[i] must be of the form $\pi_J = N_J/D_J$ with $N_J, D_J \in \mathbb{Z}$[i] where each $N_J|_{\text{Gauss}}P_0$ and $D_J|_{\text{Gauss}}P_K$.
The proof is essentially the same as before, but with $\mathbb{Z}$ replaced by $\mathbb{Z}$[i], with $\mathbb{Q}$ replaced by $\mathbb{Q}$[i], and with integer divisibility and GCD replaced by Gaussian integer divisibility and GCD.
Thus, rational complex roots of rational complex polynomials may be found.

`poly_ratcomp_root_subset` P — the subset of roots that are rational complex numbers; [it finds any (real) rational roots first].

`monic_ratcomp_factor_subset` P — the subset of monic factors for the rational complex roots.

`over_intcomp_factor_subset` P — the subset of complex integral factors for the rational complex roots.

`poly_ratcomp_roots_rem` $Q \to (RR, P)$ — rational complex roots RR with multiplicities and remainder polynomial P.

`sep_poly_ratcomp_roots_rem` $Q \to (QQ, CC, P)$ — the subset of roots that are rational complex numbers, providing the real rational roots QQ with multiplicities and non-real rational roots CC with multiplicities separately.

`monic_ratcomp_factorise` $Q \to (C, MM)$ — factorisation into scalar value C and monic factors MM.

`over_intcomp_factorise` $P \to (Q, ZZ)$ — factorisation into scalar value Q and integral complex polynomial factors ZZ.

Example 34 *Finding rational complex roots.*

```
==> def P = (monic_poly_with_roots [i,4+3*i,7%12]) * (poly_with_coeffs [4,0,2])
    ;
==> P ; degree P; monic_part P;
poly_lambda '(\X . 2*X^5+(-55%6 +: -8)*X^4+(8%3+:38%3)*X^3+(-89%6 +: -62%3)*X^2+
(-8%3+:76%3)*X+(7 +: -28%3))
5
poly_lambda '(\X . X^5+(-55%12 +: -4)*X^4+(4%3+:19%3)*X^3+(-89%12 +: -31%3)*X^2+
(-4%3+:38%3)*X+(7%2 +: -14%3))
```

```
==> poly_ratcomp_root_subset P;
[0+:1,7%12,4+:3]
==> monic_ratcomp_factor_subset P;
[poly_lambda '(\X . X^2+2),poly_lambda '(\X . X+(-4 +: -3)),poly_lambda '(\X . X
-7%12),poly_lambda '(\X . X+(0 +: -1))]
==> over_intcomp_factor_subset P;
[poly_lambda '(\X . X^2+2),poly_lambda '(\X . X+(-4 +: -3)),poly_lambda '(\X . 1
2*X-7),poly_lambda '(\X . X+(0 +: -1))]

==> poly_ratcomp_roots_rem P;
([(7%12,1),(4+:3,1),(0+:1,1)],poly_lambda '(\X . 2*X^2+4))
==> sep_poly_ratcomp_roots_rem P;
([(7%12,1)],[(4+:3,1),(0+:1,1)],poly_lambda '(\X . 2*X^2+4))
==> monic_ratcomp_factorise P;
(2,[(poly_lambda '(\X . X+(0 +: -1)),1),(poly_lambda '(\X . X-7%12),1),(poly_lam
bda '(\X . X+(-4 +: -3)),1),(poly_lambda '(\X . X^2+2),1)])
==> over_intcomp_factorise P;
(1%6,[(poly_lambda '(\X . X^2+2),1),(poly_lambda '(\X . X+(-4 +: -3)),1),(poly_l
ambda '(\X . 12*X-7),1),(poly_lambda '(\X . X+(0 +: -1)),1)])
```

The Gaussian coprime numerator and denominator of a rational complex number are given by the function `num_den_gauss`.

Example 35 *Gaussian coprime numerator and denominator.*

```
==> (6-3*i)%5
6%5 +: -3%5
==> num_den_gauss _
(3+:0,2+:1)
==> gcd_gauss (6-3*i) 5
2 +: -1
```

2.11.6 Root Multiplicities and the Square-free Part

We can do better still. Suppose that R is a root of the polynomial P with multiplicity N. Then R has multiplicity $N-1$ in the derivative P'. So, R will have multiplicity 1 in the 'square-free part' of P, which may be obtained by dividing P by the GCD of P and P'.

The square-free part may be calculated exactly if the **coefficients** are `exact`, even if the **roots** are inexact.

`square_free_part` P — the square-free part.

`monic_square_free_part` P — the monic square-free part.

Example 36 *Repeated roots, and the square-free part.*

```
==> def A = monic_poly_with_roots [0,0,1,1,1,2,2,3,3];
==> def B = pow (poly_with_coeffs [2,0,1]) 3;
==> def P = A * B; P; degree P;
poly_lambda '(\X . X^15-13*X^14+76*X^13-280*X^12+769*X^11-1693*X^10+3038*X^9-451
4*X^8+5612*X^7-5732*X^6+4712*X^5-3032*X^4+1344*X^3-288*X^2)
15
==> def DP = derive P; DP; degree _;
poly_lambda '(\X . 15*X^14-182*X^13+988*X^12-3360*X^11+8459*X^10-16930*X^9+27342
*X^8-36112*X^7+39284*X^6-34392*X^5+23560*X^4-12128*X^3+4032*X^2-576*X)
14
==> gcd P DP; degree _;
poly_lambda '(\X . X^9-7*X^8+21*X^7-45*X^6+78*X^5-96*X^4+92*X^3-68*X^2+24*X)
9
```

```
==> def S = monic_square_free_part P; S; degree S;
poly_lambda '(\X . X^6-6*X^5+13*X^4-18*X^3+22*X^2-12*X)
6
==> poly_rat_roots_rem S;
([(3,1),(2,1),(1,1),(0,1)],poly_lambda '(\X . X^2+2))
```

2.12 Solution of Polynomials

2.12.1 Galois Theory (again)

Galois theory shows that **general** polynomial may only be solved by radicals, that is, using the coefficients, integers, ordinary arithmetic and root extraction, up to degree 4, or quartic.

As mentioned in the introduction (§1.6), the solutions of the general cubic and quartic polynomials (over $\mathbb{C}$) by radicals mentioned in the literature are often difficult to apply, difficult to implement as a computer program (for example, due to requiring the finding of a real number with given properties, calculated from complex numbers, where the rounding errors in the floating point arithmetic can confound matters), and difficult to understand in principle (either in terms of length, or in the use of some formula seemingly pulled out of nowhere).

The simple-to-follow 'gems' that I found in the literature are described here.

2.12.2 Monic Depression and Polynomials

Any (non-zero) polynomial has the same roots as some monic polynomial. Given a polynomial

$$P(X) = P_0 + P_1X + P_2X^2 + \cdots + P_KX^K$$

with $P_K \neq 0$, the equivalent **monic** polynomial is

$$M(X) = P(X)/P_K$$

Then, given any monic polynomial

$$P(X) = P_0 + P_1X + P_2X^2 + \cdots + P_{K-1}X^{K-1} + X^K$$

there is a related **depressed** polynomial

$$Q(Y) = Q_0 + Q_1Y + Q_2Y^2 + \cdots + Q_{K-2}Y^{K-2} + 0 + Y^K$$

(with no Y^{K-1} term), which is found using the substitution

$$X = Y - P_{K-1}/K$$

2.12.3 The Linear Case

The linear case is essentially already solved.

2.12.4 The Quadratic Case

The monic depressed quadratic equation

$$C + X^2 = 0 \tag{1}$$

is immediately soluble.

However, there is more to say about the quadratic case than you might expect.

Given a general quadratic equation

$$C + BX + AX^2 = 0 \tag{2}$$

you are probably familiar with the 'usual' equation

$$X = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \tag{3}$$

This equation arises directly from the solution of the depressed quadratic.
Less familiar is the equivalent equation

$$X = \frac{2C}{-B \pm \sqrt{B^2 - 4AC}} \tag{4}$$

Let $D = \sqrt{B^2 - 4AC}$.
Notice that

$$\begin{aligned} X &= \frac{-B \pm D}{2A} = \frac{(-B \pm D)(-B \mp D)}{2A(-B \mp D)} \\ &= \frac{B^2 - D^2}{2A(-B \mp D)} = \frac{B^2 - (B^2 - 4AC)}{2A(-B \mp D)} = \frac{4AC}{2A(-B \mp D)} \\ &= \frac{2C}{-B \mp D} \end{aligned}$$

However, both equations (3) and (4) are numerically unstable when, for example, $A, B, C \in \mathbb{R}$ and $|AC| \ll |B|$. Therefore the module uses a more stable form.
The stable form, for real values only, is

$$X_1 = -\frac{B + \text{sgn}(B)\sqrt{B^2 - 4AC}}{2A}$$

The module uses a variation of this.
There are relationships between the roots $\{X_1, X_2\}$ and the coefficients:

$$X_1 X_2 = \frac{C}{A} \tag{5}$$

$$X_1 + X_2 = -\frac{B}{A} \tag{6}$$

Equation (6) is not stable, as it is closely related to equation (3).
Equation (5) is stable however. Thus

$$X_2 = \frac{C}{AX_1}$$

Example 37 *Stability for the quadratic equation. Note that some of these routines are* `private`.

```
==> def P = monic_poly_with_roots [10000,1/10000]
==> def [C, B, _] = coefficients P; C; B
1.0
-10000.0001

==> monic_quadratic_rad_solve_usual C B //nb: private
[0.00010000000000202272,10000.0]
==> monic_quadratic_rad_solve_alt C B //nb: private
[9999.99997977284,0.0001]

==> monic_quadratic_rad_solve_stable C B //nb: private
[10000.0,0.0001]
```

2.12.5 The Cubic Case

The solution of the monic depressed cubic equation

$$Q + PX + X^3 = 0 \tag{7}$$

may be reduced to that of a quadratic.
Substitute $X = W + \frac{P}{3W}$: $Q + P\left(W + \frac{P}{3W}\right) + \left(W + \frac{P}{3W}\right)^3 = 0$

Expand : $-\frac{P^3}{27W^3} + Q + W^3 = 0$

Multiply through by W^3 : $-\frac{P^3}{27} + QW^3 + (W^3)^2 = 0$

Let $K = -\frac{P^3}{27}$ and $Y = W^3$:

$$K + QY + Y^2 = 0 \tag{8}$$

This is quadratic, and yields 2 roots for Y: $\{Y_1, Y_2\}$. Back-substituting, each Y_i yields 3 values for W: $\{W_i, \omega W_i, \omega^2 W_i\}$ where $\omega = (-1 + \mathrm{i}\sqrt{3})/2$ is a primitive cube root of 1.

Although $\{W_1, \omega W_1, \omega^2 W_1\} \not\equiv \{W_2, \omega W_2, \omega^2 W_2\}$, back-substituting either set yields the same set $\{X_1, X_2, X_3\}$ (perhaps permuted) of roots of the depressed cubic equation (7).

The special case to watch for is when $W_i = 0$ (and hence $Y_i = 0$). If only one of W_1, W_2 is zero, then select the other. (In this case, $K = 0$ because it is the constant coefficient of the quadratic equation (8), and so $P = 0$ too.) If both $W_1 = W_2 = 0$, then $Q = 0$ due to the quadratic equation (8). Hence, in this degenerate case, the roots of equation (7) are $\{0, 0, 0\}$.

2.12.6 The Quartic Case

The solution of the monic depressed quartic equation

$$T(X) = E + DX + CX^2 + X^4 = 0 \tag{9}$$

may be reduced to that of a cubic and two quadratics.

Try to factorise into a pair of quadratics:

$$E + DX + CX^2 + X^4 = (Q + PX + X^2)(S + RX + X^2) = 0 \tag{10}$$

WLOG, assume that $D \neq 0$, so that $P, R \neq 0$. (Otherwise $T(X)$ is quadratic in X^2. This special case may be handled separately.)

So,

$$\begin{aligned} 0 &= P + R \\ C &= Q + PR + S \\ D &= PS + QR \\ E &= QS \end{aligned}$$

Therefore,

$$S + Q = C - PR = C + P^2 \tag{11}$$

$$S - Q = S + \frac{QR}{P} = \frac{D}{P} \tag{12}$$

$$SQ = E \tag{13}$$

Notice that

$$(C + P^2)^2 - \left(\frac{D}{P}\right)^2 = (S + Q)^2 - (S - Q)^2 = 4SQ = 4E$$

So, $(C + P^2)^2 - \frac{D^2}{P^2} - 4E = 0$

Let $Y = P^2$. So, $(C^2 + 2CY + Y^2) - \frac{D^2}{Y} - 4E = 0$

Multiplying by Y: $-(D^2) + (C^2 - 4E)Y + 2CY^2 + Y^3 = 0$

The 3 solutions for this cubic in Y result in 6 values for $P = \pm\sqrt{Y}$, corresponding to the 6 possible factorisations into the quadratics, with the 6 possible arrangements of two of the roots in the left-hand quadratic of the biquadratic equation (10): $\{X_1, X_2\}$, $\{X_1, X_3\}$, $\{X_1, X_4\}$, $\{X_2, X_3\}$, $\{X_2, X_4\}$, $\{X_3, X_4\}$.

We may choose any of the 6 values for P (from any of the 3 values for Y and either sign of the square root). Back substituting [and combining the equations (11–12)]

$$\begin{aligned} 2Q &= (C + P^2) - D/P \\ R &= -P \\ 2S &= (C + P^2) + D/P \end{aligned}$$

the solutions for X in the depressed quartic equation (9) may be then found by solving the resulting quadratics in the biquadratic equation (10).

2.12.7 The Quintic and Higher Degree Cases

Although the **general** quintic (or higher degree) polynomials are not soluble by radicals, certain ones are.

See §2.13 for notes on how you might approach these higher-degree cases.

2.12.8 Direct Solution by Radicals

`radical_solve` $P \to (All, PP)$ — solves linear, quadratic, cubic and quartic polynomials directly 'by radicals'; returns whether all roots were found with a subset of the roots.

The values produced are not generally of `exact` type due to the use of root extraction and floating-point arithmetic.

Example 38 *Solution by radicals.*

```
==> def P = monic_poly_with_roots [1,2,-4,i]; P; degree P
poly_lambda '(\X . X^4+(1 +: -1)*X^3+(-10 +: -1)*X^2+(8+:10)*X+(0 +: -8))
4
==> doc_simp (radical_solve P)
(true,[-4.0,0.0+:1.0,2.0,1.0])
```

2.12.9 Patterns of Coefficients

Sometimes it is possible to 'spot' certain patterns in the coefficients of a polynomial, and to transform the polynomial into a simpler one whose roots may be transformed back.

For example, 'The Fundamental Theorem for Palindromic Polynomials' says that a polynomial $P(X) = P_0 + P_1X + P_2X^2 + \cdots + P_{2K}X^{2K}$ of even degree $2K$ whose coefficients are palindromic, i.e. $\forall I \bullet P_I = P_{2K-I}$, may be transformed into a polynomial T of degree K, and if ρ is a root of the original polynomial P, then ρ is a root of the quadratic $1 - \tau X + X^2$ where τ is a root of T.

Essentially, this is done through the substitution $X = W + \dfrac{1}{W}$.

Finding the coefficients of T requires some work.

These may be found by calculating the (infinite) matrix inverse

$$\begin{bmatrix} 1 & & & & & & & & & \\ 0 & 1 & & & & & & & & \\ 2 & 0 & 1 & & & & & & & \\ 0 & 3 & 0 & 1 & & & & & & \\ 6 & 0 & 4 & 0 & 1 & & & & & \\ 0 & 10 & 0 & 5 & 0 & 1 & & & & \\ 20 & 0 & 15 & 0 & 6 & 0 & 1 & & & \\ 0 & 35 & 0 & 21 & 0 & 7 & 0 & 1 & & \\ 70 & 0 & 56 & 0 & 28 & 0 & 8 & 0 & 1 & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}^{-1} = \begin{bmatrix} 1 & & & & & & & & & \\ 0 & 1 & & & & & & & & \\ -2 & 0 & 1 & & & & & & & \\ 0 & -3 & 0 & 1 & & & & & & \\ 2 & 0 & -4 & 0 & 1 & & & & & \\ 0 & 5 & 0 & -5 & 0 & 1 & & & & \\ -2 & 0 & 9 & 0 & -6 & 0 & 1 & & & \\ 0 & -7 & 0 & 14 & 0 & -7 & 0 & 1 & & \\ 2 & 0 & -16 & 0 & 20 & 0 & -8 & 0 & 1 & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

The non-zero coefficients in the left matrix can be seen to be the binomial coefficients. Those of the right matrix were harder to identify. However, they turned out to be related to coefficients of Lucas/Cardan polynomials. (Found as sequence A034807 in the On-line Encyclopedia on Integer Sequences.)

There are other patterns of coefficients. This module includes the following patterns:

`pattern_zeroes` P — For polynomials with zero constant and neighbouring coefficients.

`pattern_power` P — For polynomials with terms only for powers of the indeterminate.

`pattern_anti_palindromic` P — For anti-palindromic polynomials.

`pattern_odd_palindromic` P — For palindromic polynomials of odd degree.

`pattern_even_palindromic` P — For palindromic polynomials of even degree.

`pattern_mixed` P — A combination of the others.

These functions take a `Polynomial` and return a pair consisting of a `Polynomial` and a corresponding function of roots. Such a returned function operates on lists of roots; that is, given a list of roots of the output polynomial, it produces a list of roots of the input polynomial.

Example 39 *A small example to demonstrate the point.*

```
==> def P = poly_with_coeffs [2,4,6,8,6,4,2]
==> P; degree P
poly_lambda '(\X . 2*X^6+4*X^5+6*X^4+8*X^3+6*X^2+4*X+2)
6
==> def (Q, F) = pattern_even_palindromic P
==> Q
poly_lambda '(\X . 2*X^3+4*X^2)
==> def (R, G) = pattern_zeroes Q
==> R; degree R
poly_lambda '(\X . 2*X+4)
1
==> def (All, R_roots) = radical_solve R; (All, # R_roots); doc_simp R_roots
(true,1)
[-2]
==> def P_roots = (F . G) R_roots; # P_roots; doc_simp P_roots
6
[0%1+:1%1,0%1 +: -1%1,0%1+:1%1,0%1 +: -1%1,-1%1,-1%1]
==> doc_simp (2 * monic_poly_with_roots P_roots)
poly_lambda '(\X . 2*X^6+4*X^5+6*X^4+8*X^3+6*X^2+4*X+2)
```

Example 40 *A rather contrived example.*

```
==> def P = poly_with_coeffs [0,0,2,2,0,0,0,0,15,15,0,1,1,0,42,42,0,3,3,0,58,58
    ,0,3,3,0,42,42,0,1,1,0,15,15,0,0,0,0,2,2]
==> degree P
39
==> def (Q, F) = pattern_mixed P
==> degree Q
3
==> def (All, Q_roots) = radical_solve Q; (All, # Q_roots); doc_simp Q_roots
(true,3)
[0.156+:1.254,-0.313,0.156 +: -1.254]
==> def P_roots = F Q_roots; # P_roots; doc_simp P_roots
39
[0,0,-1,0.866+:0.5,-0.866+:0.5,0.0 +: -1.0,0.866 +: -0.5,0.0+:1.0,-0.866 +: -0.5
,0.866+:0.5,-0.866+:0.5,0.0 +: -1.0,0.866 +: -0.5,0.0+:1.0,-0.866 +: -0.5,0.866+
:0.5,-0.866+:0.5,0.0 +: -1.0,0.866 +: -0.5,0.0+:1.0,-0.866 +: -0.5,1.069+:0.586,
-1.042+:0.632,-0.027 +: -1.218,0.72 +: -0.395,-0.018+:0.82,-0.701 +: -0.426,0.83
9 +: -0.545,0.052+:0.999,-0.891 +: -0.454,0.839+:0.545,-0.891+:0.454,0.052 +: -0
.999,1.069 +: -0.586,-0.027+:1.218,-1.042 +: -0.632,0.72+:0.395,-0.701+:0.426,-0
.018 +: -0.82]
```

There are functions to combine and compose coefficient patterns and 'solvers':

`comp_pat_2` $C2$ $C1$ — The pattern consisting of pattern $C1$ followed by pattern $C2$.

`co_comp_pat_2` $C1$ $C2$ — Again, the pattern consisting of pattern $C1$ followed by pattern $C2$.

`comp_pat_list` CC — The pattern consisting of the patterns in the list CC applied head first.

`pattern_rad_solve` C S — The solver consisting of the pattern C combined with the solver S.

There is the particular solver:

`pattern_rad_solve_mixed` P — `pattern_mixed` combined with `radical_solve`.

2.12.10 Root Bounds

Routines are available for finding bounds on the absolute values of the roots of a polynomial.
Given a polynomial

$$A(X) = \sum_{i=0}^{n} a_i X^i = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$$

where $a_n \neq 0$, the roots α satisfy many bounds.
The bounds in terms of the coefficients include:

$$|\alpha| \leq \frac{||A||_2}{|a_n|} \tag{Landau}$$

$$|\alpha| < 1 + \frac{\max\{\ |a_i| \bullet 0 \leq i < n\}}{|a_n|} \tag{Cauchy1}$$

$$|\alpha| \leq \max\left\{ \sqrt[(n-i)]{\frac{n|a_i|}{|a_n|}} \ \bullet\ 0 \leq i < n \right\} \tag{Cauchy2}$$

$$|\alpha| < 2\max\left\{ \sqrt[(n-i)]{\frac{|a_i|}{|a_n|}} \ \bullet\ 0 \leq i < n \right\} \tag{Zassenhaus}$$

$$|\alpha| \leq \frac{1}{\sqrt[n]{2}-1} \max\left\{ \sqrt[(n-i)]{\frac{|a_i|}{\binom{n}{i}|a_n|}} \ \bullet\ 0 \leq i < n \right\} \tag{Yap}$$

The k-norm, for real $k > 0$ is defined as follows.

$$||A||_k = \sqrt[k]{\sum_{i=0}^{n} |a_i|^k} \tag{14}$$

$$||A||_\infty = \lim_{k\to\infty} ||A||_k = \max\{|a_i| \bullet 0 \leq i \leq n\} \tag{15}$$

The 1-norm is known as the 'weight', and is the sum of absolute values; the 2-norm, the 'length', is the r.m.s. (of absolute values, in case of complex values); and the ∞-norm, the 'height', is the maximum of absolute values of the coefficiants of the polynomial.

`poly_norm` K P — The K-norm of polynomial P; K should be real and > 1, or `inf`.

`all_root_sup_landau` P — A non-strict upper bound [see (Landau)] for all roots of polynomial P due to Landau.

`all_root_sup_cauchy1` P — A strict upper bound [see (Cauchy1)] for all roots of polynomial P due to Cauchy.

`all_root_sup_cauchy2` P — A non-strict upper bound [see (Cauchy2)] for all roots of polynomial P also due to Cauchy.

`all_root_sup_zassenhaus` P — A strict upper bound [see (Zassenhaus)] for all roots of polynomial P due to Zassenhaus.

`all_root_sup_yap` P — A non-strict upper bound [see (Yap)] for all roots of polynomial P due to Yap.

`all_root_sup` P — The combined upper bound for all roots polynomial P.

`all_root_sups` P — A list of upper bounds for all roots polynomial P.

`some_root_sup_ends` P — An upper bound for some root of polynomial P.

In the function names, '`sup`' is short for 'supremum' (or 'maximum').
(Beware that if L and U are a lower bound for some root and an upper bound for some root, then those roots may not be the same; there may not even be a root between L and U.)

Example 41 *Root bounds.*

```
==> def P = monic_poly_with_roots [1,2,-4]; P
poly_lambda '(\X . X^3+X^2-10*X+8)
==> all_root_sups P
[12.8840987267251,11%1,5.47722557505166,6.32455532033676,7.69464420372614]
==> all_root_sup P
5.47722557505166
```

There is a curious fact: if you reverse the coefficients of a polynomial, its roots are reciprocated (except for the zeros, which are lost). This fact may be used to obtain a lower bound on the absolute values of the roots based on the upper bounds. If B is an upper bound for P reversed, then $1/B$ is a lower bound for P.

`reverse_poly` P — The 'reverse' polynomial.

`all_root_inf_landau` P — A non-strict lower bound for all roots of polynomial P due to Landau.

`all_root_inf_cauchy1` P — A strict lower bound for all roots of polynomial P due to Cauchy.

`all_root_inf_cauchy2` P — Another, non-strict, lower bound for all roots of polynomial P due to Cauchy.

`all_root_inf_zassenhaus` P — A strict lower bound for all roots of polynomial P due to Zassenhaus.

`all_root_inf_yap` P — A non-strict lower bound for all roots of polynomial P due to Yap.

`all_root_inf` P — The combined lower bound for all roots of polynomial P.

`all_root_infs` P — A list of lower bounds for all roots polynomial P.

In the function names, '`inf`' is short for 'infimum' (or 'minimum').

Example 42 *Reverse polynomial and reciprocal roots.*

```
==> def P = monic_poly_with_roots [0,1,-4,i,2-i]; P
poly_lambda '(\X . X^5+X^4+(-9+:2)*X^3+(11+:6)*X^2+(-4 +: -8)*X)
==> reverse_poly P
poly_lambda '(\X . (-4 +: -8)*X^4+(11+:6)*X^3+(-9+:2)*X^2+X+1)
==> def K = poly_ratcomp_roots_rem _; K
([(1,1),(-1%4,1),(2%5+:1%5,1),(0 +: -1,1)],poly_lambda '(\X . -4 +: -8))
==> map (fst) (fst K)
[1,-1%4,2%5+:1%5,0 +: -1]
==> map (ratcomp_simplify.(1%)) _
[1,-4,2 +: -1,0+:1]
```

2.12.11 Root Approximation

`root_approx_eps` *MaxNondecJumps* ε *P Seed* — Using Newton's method, seek a root X of P, perhaps near the seed value *Seed*, such that $|P(X)| < \varepsilon$, and allowing up to *MaxNondecJumps* iterations that don't cause the value of $P(X)$ to strictly decrease; returns `()` upon failure; may not terminate if *MaxNondecJumps* = `inf`.

`root_approx_seq` *From To P Seed* — Using Newton's method, show the iterations from *From* to *To*; this is useful if `root_approx_eps` returns `()`.

These routines are very slow near multiple roots. They are therefore best applied to the square-free part (see §2.11.6).

These routines are also very slow with exact rational values for the *Seed*, due to an exponential increase in the length (log) of the numerator and denominators per iteration. Therefore, only inexact (floating-point) values of *Seed* are recommended.

2.13 Strategy for Finding Roots

- Find the square-free part (§2.11.6).
- Seek patterns in the coefficients (§2.12.9).
- Factor out rational roots (§2.11.3).
- Factor out rational complex roots (§2.11.5).
- Solve any remainder by radicals if possible (§2.12).
- Seek roots by approximation (§2.12.11).

Some of these methods are combined into `solve`:

`solve` $P \to$ (*All*, *QiRM*, *CR*, *Rem*) — try to find all the roots of P; *All* = whether all roots were found, *QiRM* = exact roots with their multiplicities, *CR* = approximate inexact (irrational) roots found, *Rem* = remainder assuming the CR all have multiplicity 1 (which may not be true); if *All* = true **and** *Rem* is a constant polynomial then all roots were found and the CR do all have multiplicity 1; if *All* = true but *Rem* is a non-constant polynomial then all roots were found and some of the CR have multiplicity > 1.

3 Difference Systems

The module defines the type `DiffSystem`.

3.1 Introduction — Difference Tables

Consider the polynomial $P(N) = 1 + N^2$, and its values when $N = 0, 1, 2, \ldots$, and their successive differences. This produces a difference table (Figure 1).

1	2	5	10	17	26	⋯
1	3	5	7	9	⋯	
2	2	2	2	⋯		
0	0	0	⋯			

Figure 1: A (forwards) difference table.

The leading diagonal figures in such a table may be considered to be the coefficients of a difference system.

Newton's formula states that the difference system with just a 1 in the n^{th} position (counting from zero) generates the values $\binom{i}{n}$.

Example 43 *Newton's observation.*

```
==> def D = fwd_diffsys_with_coeffs [0,0,0,1]
==> def P = poly_from_diffsys D; P
poly_lambda '(\X . 1%6*X^3-1%2*X^2+1%3*X)
==> map (rat_simplify . (apply P)) (nums 0 10)
[0,0,0,1,4,10,20,35,56,84,120]
```

There is an alternative 'backwards' difference table (Figure 2).

			1		2		5		10		17		26		$\cdots$
		-1		1		3		5		7		9		$\cdots$	
	2		2		2		2		2		2		$\cdots$		
0		0		0		0		0		$\cdots$					

Figure 2: A backwards difference table.

The leading diagonal figures in this sort of table may also be considered to be the coefficients of a difference system.

3.2 Creating Difference Systems

`fwd_diffsys_with_coeffs` CC — forwards `DiffSystem` with the given list of coefficients.

`bkwd_diffsys_with_coeffs` CC — backwards `DiffSystem` with the given list of coefficients.

3.3 Difference System Deconstruction and Tests

`coefficients` D — the whole list of coefficients of D.

`isdiffsys` D — whether D is a difference system.

`is_fds` D — whether D is a forwards difference system.

`is_bds` D — whether D is a backwards difference system.

3.4 Difference System Arithmetic and Manipulation

The following operators are defined for `DiffSystem`: `(+)` and `(-)`, unary `-`, `(*)` for multiplication by a scalar only, and `(/)` and `(%)` for division by a scalar only.
The following low-level operations on difference systems are defined:

`succ_ds` D — form the 'successor' system: advance the `DiffSystem` D forwards.

`pred_ds` D — form the 'predecessor' system: step the `DiffSystem` D backwards.

`rsh_ds` N D — right-shift N into the coefficients of D.

`lsh_ds` D — left-shift the coefficients of D.

3.5 Conversion between Polynomials and Difference Systems

There are functions to convert between `Polynomial` and `DiffSystem`.

`fwd_diffsys_from_poly` P — forwards `DiffSystem` equivalent to `Polynomial` P.

`bkwd_diffsys_from_poly` P — backwards `DiffSystem` equivalent to `Polynomial` P.

`poly_from_diffsys` D — `Polynomial` equivalent to (forwards or backwards) `DiffSystem` D.

Example 44 *The (forwards) difference system for* $1 + N^2$.

```
==> def P = poly_with_coeffs [1,0,1]; P
poly_lambda '(\X . X^2+1)
==> def D = fwd_diffsys_from_poly P; D
fwd_diffsys_with_coeffs [1,1,2]
==> poly_from_diffsys D
poly_lambda '(\X . X^2+1)
```

Example 45 *The backwards difference system for* $1 + N^2$.

```
==> def P = poly_with_coeffs [1,0,1]; P
poly_lambda '(\X . X^2+1)
==> def D = bkwd_diffsys_from_poly P; D
bkwd_diffsys_with_coeffs [1,-1,2]
==> poly_from_diffsys D
poly_lambda '(\X . X^2+1)
```

A backwards difference system with just a 1 in the n^{th} position (counting from zero) generates the values $\Delta_i^n = \binom{n+i-1}{n}$, which are the n-dimensional polytope numbers.

Example 46 *A variant of Newton's observation. The 2- and 3-dimensional polytope numbers are the triangular and tetrahedral numbers.*

```
==> // 3-D: Tetrahedral numbers
==> def D = bkwd_diffsys_with_coeffs [0,0,0,1]
==> def P = poly_from_diffsys D; P
poly_lambda '(\X . 1%6*X^3+1%2*X^2+1%3*X)
==> map (rat_simplify . (apply P)) (nums 0 10)
[0,1,4,10,20,35,56,84,120,165,220]

==> // 2-D: Triangular numbers
==> def D = bkwd_diffsys_with_coeffs [0,0,1]
==> def P = poly_from_diffsys D; P
poly_lambda '(\X . 1%2*X^2+1%2*X)
==> map (rat_simplify . (apply P)) (nums 0 10)
[0,1,3,6,10,15,21,28,36,45,55]
```

3.6 Rebasing Difference Systems

The `rebase` family of functions of §2.9 may also be applied to `DiffSystem`.

3.7 Recurrence Relations (Appendix)

Difference systems make use of recurrence relations to convert between polynomial coefficients and difference system coefficients.

Example 47 *Some well-known special 'triangles' of numbers*

```
==> // C(n,k) : Binomial Coefficients
==> recurrence_triangle rule_binomial 6;
[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1],[1,5,10,10,5,1]]

==> // s(n;k) : Stirling Numbers of the 1st Kind
==> recurrence_triangle rule_stirling_1st 6;
[[1],[0,1],[0,-1,1],[0,2,-3,1],[0,-6,11,-6,1],[0,24,-50,35,-10,1]]
==> // S(n;k) : Stirling Numbers of the 2nd Kind = Stirling Set Numbers
==> recurrence_triangle rule_stirling_2nd 6;
[[1],[0,1],[0,1,1],[0,1,3,1],[0,1,7,6,1],[0,1,15,25,10,1]]

==> // Stirling Cycle Numbers = unsigned Stirling Numbers of the 1st Kind
```

```
==> recurrence_triangle rule_stirling_cycle 6;
[[1],[0,1],[0,1,1],[0,2,3,1],[0,6,11,6,1],[0,24,50,35,10,1]]
==> // signed Stirling Numbers of the 2nd Kind
==> recurrence_triangle rule_stirling_2nd_signed 6;
[[1],[0,1],[0,-1,1],[0,1,-3,1],[0,-1,7,-6,1],[0,1,-15,25,-10,1]]
```

Example 48 *Some other triangles of numbers*

```
==> // Conversion coefficients from polynomials to difference systems
==> recurrence_triangle rule_bds_from_poly 6;
[[1],[0,1],[0,-1,2],[0,1,-6,6],[0,-1,14,-36,24],[0,1,-30,150,-240,120]]
==> recurrence_triangle rule_fds_from_poly 6;
[[1],[0,1],[0,1,2],[0,1,6,6],[0,1,14,36,24],[0,1,30,150,240,120]]
==> // Conversion coefficients from difference systems to polynomials
==> recurrence_triangle rule_poly_from_bds 5;
[[1],[0,1],[0,1%2,1%2],[0,1%3,1%2,1%6],[0,1%4,11%24,1%4,1%24]]
==> recurrence_triangle rule_poly_from_fds 5;
[[1],[0,1],[0,-1%2,1%2],[0,1%3,-1%2,1%6],[0,-1%4,11%24,-1%4,1%24]]
```

3.8 Application of Difference Systems (Appendix)

Either of these types of difference system may be used to find, for example, $\sum_{k=1}^{n} k^3$. (See §3.9 where routines that encapsulate these methods are described.)

Example 49 *Finding* $\sum_{k=1}^{n} k^3$ *using a forwards difference system.*

```
==> def P = poly_term 3 1; P
poly_lambda '(\X . X^3)
==> map (rat_simplify . (apply P)) (nums 0 10)
[0,1,8,27,64,125,216,343,512,729,1000]
==> fwd_diffsys_from_poly P
fwd_diffsys_with_coeffs [0,1,6,6]
==> succ_ds _ // advance the ds forwards
fwd_diffsys_with_coeffs [1,7,12,6]
==> rsh_ds 0 _ // right shift the coefficients
fwd_diffsys_with_coeffs [0,1,7,12,6]
==> def S = poly_from_diffsys _; S
poly_lambda '(\X . 1%4*X^4+1%2*X^3+1%4*X^2)
==> map (rat_simplify . (apply S)) (nums 0 10)
[0,1,9,36,100,225,441,784,1296,2025,3025]
```

Example 50 *Finding* $\sum_{k=1}^{n} k^3$ *using a backwards difference system.*

```
==> bkwd_diffsys_from_poly P
bkwd_diffsys_with_coeffs [0,1,-6,6]
==> rsh_ds 0 _ // just right shift the coefficients
bkwd_diffsys_with_coeffs [0,0,1,-6,6]
==> def S = poly_from_diffsys _; S
poly_lambda '(\X . 1%4*X^4+1%2*X^3+1%4*X^2)
==> over_int_parts S
(1%4,poly_lambda '(\X . X^4+2*X^3+X^2))
==> over_int_factorise S
(1%4,[(poly_lambda '(\X . X),2),(poly_lambda '(\X . X+1),2)])
```

The penultimate line here shows that $\sum_{k=1}^{n} k^3 = \dfrac{n^2 + 2n^3 + n^4}{4}$.
The final line shows that $\sum_{k=1}^{n} k^3 = \frac{1}{4}n^2(1+n)^2$
By the way:

Example 51 *Q: Why from $k = 1$?*
A: Because that remains consistent for degree-0 polynomials.

```
==> def P = poly_term 0 1; P
poly_lambda '(\X . 1)
==> map (rat_simplify . (apply P)) (nums 0 10)
[1,1,1,1,1,1,1,1,1,1,1]
==> bkwd_diffsys_from_poly P
bkwd_diffsys_with_coeffs [1]
==> rsh_ds 0 _
bkwd_diffsys_with_coeffs [0,1]
==> def S = poly_from_diffsys _; S
poly_lambda '(\X . X)
==> map (rat_simplify . (apply S)) (nums 0 10)
[0,1,2,3,4,5,6,7,8,9,10]
```

3.9 Finite Sums and Finite Differences (of Polynomials)

Finite sums and finite differences are specifically provided for. [The standard notation is ∇F ("nabla" or "del") for the backwards difference of F, and ΔF ("delta") for the forwards difference of F.]

`ds_incl_sum` N D — given the `DiffSystem` D, form its 'inclusive sum' `DiffSystem`; N is the initial value, usually 0.

`ds_excl_sum` N D — given the `DiffSystem` D, form its 'exclusive sum' `DiffSystem`; N is the initial value, usually 0.

`ds_bkwd_diff` D — given the `DiffSystem` D, form its 'backwards difference' `DiffSystem`.

`ds_fwd_diff` D — given the `DiffSystem` D, form its 'forwards difference' `DiffSystem`.

`poly_incl_sum` P — given the `Polynomial` P, form its 'inclusive sum' `Polynomial`; no initial value is specified: 0 is assumed.

`poly_excl_sum` P — given the `Polynomial` P, form its 'exclusive sum' `Polynomial`; no initial value is specified: 0 is assumed.

`poly_bkwd_diff` P — given the `Polynomial` P, form its 'backwards difference' `Polynomial`.

`poly_fwd_diff` P — given the `Polynomial` P, form its 'forwards difference' `Polynomial`.

Example 52 *Finding sums and differences of k^4.*

```
==> def P = poly_term 4 1; P
poly_lambda '(\X . X^4)
==> map (rat_simplify . (apply P)) (nums 0 10)
[0,1,16,81,256,625,1296,2401,4096,6561,10000]

==> def S = poly_incl_sum P; S
poly_lambda '(\X . 1%5*X^5+1%2*X^4+1%3*X^3-1%30*X)
==> map (rat_simplify . (apply S)) (nums 0 10)
[0,1,17,98,354,979,2275,4676,8772,15333,25333]

==> def S = poly_excl_sum P; S
poly_lambda '(\X . 1%5*X^5-1%2*X^4+1%3*X^3-1%30*X)
==> map (rat_simplify . (apply S)) (nums 0 10)
[0,0,1,17,98,354,979,2275,4676,8772,15333]

==> def D = poly_bkwd_diff P; D
poly_lambda '(\X . 4*X^3-6*X^2+4*X-1)
==> map (rat_simplify . (apply D)) (nums 0 10)
```

```
[-1,1,15,65,175,369,671,1105,1695,2465,3439]

==> def D = poly_fwd_diff P; D
poly_lambda '(\X . 4*X^3+6*X^2+4*X+1)
==> map (rat_simplify . (apply D)) (nums 0 10)
[1,15,65,175,369,671,1105,1695,2465,3439,4641]
```

It's not difficult to obtain forward and backward differences in another way:

Example 53 *Alternative calculation of differences of* k^4.

```
==> def P = poly_term 4 1; P
poly_lambda '(\X . X^4)
==> P - (P . (poly_with_coeffs [-1,1]))
poly_lambda '(\X . 4*X^3-6*X^2+4*X-1)
==> (P . (poly_with_coeffs [1,1])) - P
poly_lambda '(\X . 4*X^3+6*X^2+4*X+1)
```

Finite sums are more difficult to obtain so directly.

3.9.1 Digression: Finite Sums via Faulhaber's Formula

Finite sums may be obtained using Faulhaber's formula (16).

$$\sum_{k=1}^{n} k^p = \frac{1}{p+1} \sum_{i=1}^{p+1} \left[(-1)^{\delta_{ip}} \binom{p+1}{i} B_{p+1-i} \right] n^i$$

or equivalently (since the B_j are 0 where it would otherwise make a difference):

$$\sum_{k=1}^{n} k^p = \frac{1}{p+1} \sum_{i=1}^{p+1} \left[(-1)^{p+1-i} \binom{p+1}{i} B_{p+1-i} \right] n^i \tag{16}$$

or equivalently (setting $q = p+1$):

$$\sum_{k=1}^{n} k^{q-1} = \frac{1}{q} \sum_{i=1}^{q} \left[(-1)^{q-i} \binom{q}{i} B_{q-i} \right] n^i$$

or (setting $j = q - i$):

$$\sum_{k=1}^{n} k^{q-1} = \frac{1}{q} \sum_{j=0}^{q-1} \left[(-1)^{j} \binom{q}{j} B_{j} \right] n^{q-j}$$

where the B_j are the Bernoulli numbers.
Note that the formula is of the forms:

$$\sum_{k=1}^{n} k^{q-1} = \sum_{i=1}^{q} a_{q,i}\ n^i$$
$$\text{or} \quad \sum_{k=1}^{n} k^{p} = \sum_{i=1}^{p+1} f_{p,i}\ n^i$$

where

$$a_{q,i} = \frac{(-1)^{q-i}\binom{q}{i} B_{q-i}}{q}$$
$$f_{p,i} = \frac{(-1)^{\delta_{pi}}\binom{p+1}{i} B_{p+1-i}}{p+1}$$

The Bernoulli numbers are given by

$$B_0 = 1$$
$$B_n = -\frac{1}{n+1}\sum_{k=0}^{n-1}\binom{n+1}{k}B_k \qquad \text{if } n > 0$$

It happens that $B_k = 0$ for odd $k \geq 3$.

k	B_k
0	1
1	$-1/2$
2	$1/6$
3	0
4	$-1/30$
6	$1/42$
8	$-1/30$
10	$5/66$
12	$-691/2730$
$\vdots$	$\vdots$

The first few expansions of Faulhaber's formula are:

$$\sum_{k=1}^{n} k^0 = n$$
$$\sum_{k=1}^{n} k^1 = \tfrac{1}{2}n^2 + \tfrac{1}{2}n$$
$$\sum_{k=1}^{n} k^2 = \tfrac{1}{3}n^3 + \tfrac{1}{2}n^2 + \tfrac{1}{6}n$$
$$\sum_{k=1}^{n} k^3 = \tfrac{1}{4}n^4 + \tfrac{1}{2}n^3 + \tfrac{1}{4}n^2$$
$$\sum_{k=1}^{n} k^4 = \tfrac{1}{5}n^5 + \tfrac{1}{2}n^4 + \tfrac{1}{3}n^3 - \tfrac{1}{30}n$$
$$\sum_{k=1}^{n} k^5 = \tfrac{1}{6}n^6 + \tfrac{1}{2}n^5 + \tfrac{5}{12}n^4 - \tfrac{1}{12}n^2$$
$$\sum_{k=1}^{n} k^6 = \tfrac{1}{7}n^7 + \tfrac{1}{2}n^6 + \tfrac{1}{2}n^5 - \tfrac{1}{6}n^3 + \tfrac{1}{42}n$$
$$\sum_{k=1}^{n} k^7 = \tfrac{1}{8}n^8 + \tfrac{1}{2}n^7 + \tfrac{7}{12}n^6 - \tfrac{7}{24}n^4 + \tfrac{1}{12}n^2$$
$$\sum_{k=1}^{n} k^8 = \tfrac{1}{9}n^9 + \tfrac{1}{2}n^8 + \tfrac{2}{3}n^7 - \tfrac{7}{15}n^5 + \tfrac{2}{9}n^3 - \tfrac{1}{30}n$$
$$\sum_{k=1}^{n} k^9 = \tfrac{1}{10}n^{10} + \tfrac{1}{2}n^9 + \tfrac{3}{4}n^8 - \tfrac{7}{10}n^6 + \tfrac{1}{2}n^4 - \tfrac{3}{20}n^2$$
$$\sum_{k=1}^{n} k^{10} = \tfrac{1}{11}n^{11} + \tfrac{1}{2}n^{10} + \tfrac{5}{6}n^9 - n^7 + n^5 - \tfrac{1}{2}n^3 + \tfrac{5}{66}n$$

The module provides a few utility functions in order to do calculations based on Faulhaber's formula:

`fhash` *H F X* — given a unary function *F* and an empty `ref` *H* to an *F*-specific `dict` or `hdict`, `fhash` *H F* is a hashed version of *F*; i.e. *H* must be used only for this purpose.

`cfhash` F X — given a unary function F, `cfhash` F is a hashed version of F; this version uses a common `hdict ref`.

`factorial` N — $N! = \prod_{i=0}^{N} i$

`falling_factorial` X K — notations: $(x)_k$, $x^{\underline{k}}$.

`rising_factorial` X K — notations: $x^{(k)}$, $x^{\overline{k}}$.

`binomial` N K — $\binom{N}{K}$.

`bernoulli` N — B_N.

`faulhaber` P I — Faulhaber's (P, I)-coefficient.

`faulhaber_series` P — the list of Faulhaber's (P, I)-coefficients for $\sum x^P$.

Example 54 *Faulhaber's formula.*

```
==> poly_incl_sum (poly_term 4 1)
poly_lambda '(\X . 1%5*X^5+1%2*X^4+1%3*X^3-1%30*X)
==> poly_with_coeffs (faulhaber_series 4)
poly_lambda '(\X . 1%5*X^5+1%2*X^4+1%3*X^3-1%30*X)
```

3.10 Operators

Some discrete operators have already been described (see §3.9):
∇ is `poly_bkwd_diff`.
Δ is `poly_fwd_diff`.
Some continuous operators have also been described (see §2.10):
D is `derive`.
D^{-1} is `antiderive`.
We now introduce the further discrete operators:
E is `shift`, which is equivalent to `poly_translate` (-1) 0 (see §2.8), is such that $(EP)X = P(X + 1)$.
$E^{\frac{1}{2}}$ is `semishift`.
I is `identity`.
$E^{-\frac{1}{2}}$ is `unsemishift`.
E^{-1} is `unshift`.
There is a correspondence between certain discrete and continuous operators. For example, finite differences and sums correspond to derivations and anti-derivations, respectively.

3.10.1 Digression: Operator Relationships

Note that

$$\begin{aligned}
\nabla &= I - E^{-1} \\
&= E^{-1} \circ \Delta \\
&= \Delta \circ E^{-1} \\
\delta &= E^{\frac{1}{2}} - E^{-\frac{1}{2}} \\
\Delta &= E - I \\
&= E \circ \nabla \\
&= \nabla \circ E
\end{aligned}$$

where $(\circ)$ denotes composition
and where δ is the "central difference operator".

(This module does not provide operator arithmetic.)

3.10.2 Another Digression: Further Operator Relationships

There is a very curious formula linking a continuous and a discrete operator:

$$E = e^D$$

(or equivalently $E = \exp D$ or $D = \ln E$).
What does this mean?
Well, e^z is defined as follows:

$$\begin{aligned} e^z &= \sum_{n=0}^{\infty} \frac{z^n}{n!} \\ &= 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \cdots \end{aligned}$$

This formula works for any suitable mathematical object (element of a 'ring'), such as complex numbers $z \in \mathbb{C}$, square matrices, or operators.
So, for a polynomial P and our operators E and D:

$$\begin{aligned} EP &= \left(\sum_{n=0}^{\infty} \frac{D^n}{n!} \right) P \\ &= \left(I + D + \frac{D^2}{2} + \frac{D^3}{6} + \cdots \right) P \\ &= P + P' + \tfrac{1}{2}P'' + \tfrac{1}{6}P''' + \cdots \end{aligned}$$

and of course, for a polynomial P, this series terminates after a number of terms equal to one more than the degree of P.
Let's try this on a polynomial:

Example 55 $E = e^D$.

```
==> def P = poly_val "\\X . X^3 + 4*X^2 + 9"
==> shift P
poly_lambda '(\X . X^3+7*X^2+11*X+14)
==> P + derive P + (derive $ derive P % 2) + (derive $ derive $ derive P % 6)
poly_lambda '(\X . X^3+7*X^2+11*X+14)
==> derive (derive (derive P % 3 + P) % 2 + P) + P
poly_lambda '(\X . X^3+7*X^2+11*X+14)
```

4 Bézier Parameterisations

The module defines the type `Bezier`.

4.1 Introduction

The Béziers used here are 1-D parametric Béziers. The parameter is thought of as ranging over the unit interval $[0, 1]$.
The coefficients of a Bézier may be considered to be control points equally distributed across that interval. However, in general, only the first and last of these will lie of the Bézier's curve.
For work in 2-D, 3-D or higher dimensions, pairs, triples, and tuples of Béziers must be used.

4.2 Creating Béziers

A Bézier function may be created directly in terms of its coefficients (also known as 'control points').

`bezier_with_coeffs` CC — Bézier with given list of coefficients or 'control points'.

4.3 Bézier Functions

Few arithmetic functions or operators are supplied.
The function `apply` may be used with Bézier functions as with polynomials.

`apply` B T — $B(T)$.

4.4 Conversion between Polynomials and Béziers

There are functions to transform between polynomials and their Bézier parameterisations.

`bezier_from_poly` P — `Bezier` equivalent to `Polynomial` P; it has the same degree as P.

`bezier_from_deg_poly` Deg P — `Bezier` equivalent to `Polynomial` P; it has degree as Deg, which must be at least that of P.

`poly_from_bezier` B — `Polynomial` equivalent to `Bezier` B.

Example 56 *Bézier parameterisations.*

```
==> def P = poly_thru_points [(0,1),(1%4,0),(1%2,2),(3%4,-1),(1,-1%2)]; P
poly_lambda '(\X . 176*X^4-1048%3*X^3+209*X^2-223%6*X+1)
==> def B = bezier_from_poly P; B
bezier_with_coeffs [1,-199%24,69%4,-233%24,-1%2]
==> map (rat_simplify . (apply P)) (numsby (1%8) 0 1)
[1,-261%256,0,387%256,2,251%256,-1,-605%256,-1%2]
==> map (rat_simplify . (apply B)) (numsby (1%8) 0 1)
[1,-261%256,0,387%256,2,251%256,-1,-605%256,-1%2]
==> poly_from_bezier B //convert back
poly_lambda '(\X . 176*X^4-1048%3*X^3+209*X^2-223%6*X+1)
```

4.5 Adjusting the degree of Béziers

An additional control point (coefficient) may be added to a `Bezier`.

`inc_degree` B — equivalent `Bezier` of the next higher degree.

Example 57 *Increasing the degree.*

```
==> def B1 = bezier_with_coeffs [0,1,1,0]; B1
bezier_with_coeffs [0,1,1,0]
==> map (rat_simplify . (apply B1)) (numsby (1%6) 0 1)
[0,5%12,2%3,3%4,2%3,5%12,0]
==> def B2 = inc_degree B1; B2
bezier_with_coeffs [0,3%4,1,3%4,0]
==> map (rat_simplify . (apply B2)) (numsby (1%6) 0 1)
[0,5%12,2%3,3%4,2%3,5%12,0]
```

4.6 Rebasing Béziers

The `rebase` family of functions of §2.9 may also be applied to `Bezier`.

4.7 Splitting Béziers

Béziers may be split at some value of the parameter.

`bez_left` T B — the reparameterised `Bezier` for the 'left' or 'initial' segment of B split at T.

`bez_right` T B — the reparameterised `Bezier` for the 'right' or 'terminal segment of B split at T.

`bez_split` T B $\rightarrow (L,\ R)$ — the reparameterised `Beziers` for both segments of B split at T.

Example 58 *Splitting Béziers.*

```
==> def B = bezier_with_coeffs [10,0,5,20]; B
bezier_with_coeffs [10,0,5,20]
==> map (apply B) [0,1%3,1]
[10,130%27,20]
==> def (L, R) = bez_split (1%3) B
==> map (apply L) [0,1]
[10%1,130%27]
==> map (apply R) [0,1]
[130%27,20%1]
```

4.8 Bézier Calculus

The `derive` function can be applied to `Bezier`.

Example 59 *Deriving Béziers.*

```
==> def P = poly_with_coeffs [2,0,0,1,1]; P
poly_lambda '(\X . X^4+X^3+2)
==> def B = bezier_from_poly P; B
bezier_with_coeffs [2,2,2,9%4,4]
==> derive B
bezier_with_coeffs [0,0,1,7]
==> poly_from_bezier _
poly_lambda '(\X . 4*X^3+3*X^2)
```