

FastPolyEval

1.0

Generated by Doxygen 1.8.17

1 Fast Evaluation of Real and Complex Polynomials	1
1.1 Abstract	1
1.2 Complexity & accuracy	2
1.3 Description of the Fast Polynomial Evaluator (FPE) algorithm	3
1.3.1 General principles	3
1.3.2 Geometric selection principle	4
1.3.3 Parsimonious representation at an arbitrary point	4
1.3.4 Implementation of the algorithm	5
1.4 Installation of FastPolyEval	6
1.4.1 Prerequisites	6
1.4.2 Compile FastPolyEval	6
1.4.3 Additional help for Linux, MacOS, Windows	7
1.4.4 Uninstall FastPolyEval	7
1.5 Basic usage of FastPolyEval	7
1.5.1 Onboard help system	7
1.5.2 Number formats	7
1.5.3 Input and outputs	8
1.5.4 Tasks for generating and handling polynomials	9
1.5.5 Tasks for generating and handling sets of complex numbers	9
1.5.6 Tasks using the FPE algorithm for production use and benchmarking	10
1.6 Notes about the implementation	11
1.7 References, Contacts and Copyright	11
1.8 Thanks	11
1.9 License	12
2 Data Structure Index	13
2.1 Data Structures	13
3 File Index	15
3.1 File List	15
4 Data Structure Documentation	17
4.1 array_struct Struct Reference	17
4.1.1 Detailed Description	17
4.2 arrayf_struct Struct Reference	17
4.2.1 Detailed Description	18
4.3 comp_struct Struct Reference	18
4.3.1 Detailed Description	18
4.4 compf_struct Struct Reference	18
4.4.1 Detailed Description	18
4.5 concave_struct Struct Reference	18
4.5.1 Detailed Description	19
4.6 eval_struct Struct Reference	19

4.6.1 Detailed Description	20
4.7 evalf_struct Struct Reference	20
4.7.1 Detailed Description	20
4.8 help_struct Struct Reference	21
4.8.1 Detailed Description	21
4.9 list_struct Struct Reference	21
4.9.1 Detailed Description	22
4.10 poly_struct Struct Reference	22
4.10.1 Detailed Description	22
4.11 polyf_struct Struct Reference	22
4.11.1 Detailed Description	23
4.12 polyfr_struct Struct Reference	23
4.12.1 Detailed Description	23
4.13 polyr_struct Struct Reference	23
4.13.1 Detailed Description	24
4.14 pows_struct Struct Reference	24
4.14.1 Detailed Description	24
4.15 powsf_struct Struct Reference	24
4.15.1 Detailed Description	25
4.16 powsfr_struct Struct Reference	25
4.16.1 Detailed Description	25
4.17 powsr_struct Struct Reference	25
4.17.1 Detailed Description	26
5 File Documentation	27
5.1 /home/runner/work/FastPolyEval/FastPolyEval/code/apps/apps.h File Reference	27
5.1.1 Detailed Description	28
5.1.2 Function Documentation	28
5.1.2.1 app_analyse()	28
5.1.2.2 app_compare()	29
5.1.2.3 app_conj()	29
5.1.2.4 app_eval_d()	30
5.1.2.5 app_eval_n()	30
5.1.2.6 app_eval_p()	31
5.1.2.7 app_eval_r()	32
5.1.2.8 app_exp()	32
5.1.2.9 app_grid()	33
5.1.2.10 app_im()	33
5.1.2.11 app_join()	33
5.1.2.12 app_normal()	34
5.1.2.13 app_polar()	34
5.1.2.14 app_rand()	35

5.1.2.15 app_re()	35
5.1.2.16 app_rot()	36
5.1.2.17 app_sphere()	36
5.1.2.18 app_tensor()	36
5.1.2.19 app_unif()	38
5.2 /home/runner/work/FastPolyEval/FastPolyEval/code/apps/appsf.h File Reference	38
5.2.1 Detailed Description	39
5.2.2 Function Documentation	39
5.2.2.1 appf_compare()	39
5.2.2.2 appf_conj()	40
5.2.2.3 appf_eval_d()	40
5.2.2.4 appf_eval_n()	41
5.2.2.5 appf_eval_p()	41
5.2.2.6 appf_eval_r()	42
5.2.2.7 appf_exp()	43
5.2.2.8 appf_grid()	43
5.2.2.9 appf_im()	43
5.2.2.10 appf_join()	44
5.2.2.11 appf_polar()	44
5.2.2.12 appf_re()	44
5.2.2.13 appf_rot()	45
5.2.2.14 appf_tensor()	45
5.2.2.15 appf_unif()	46
5.3 /home/runner/work/FastPolyEval/FastPolyEval/code/apps/help.h File Reference	46
5.3.1 Detailed Description	47
5.3.2 Typedef Documentation	47
5.3.2.1 help	47
5.3.3 Function Documentation	48
5.3.3.1 fpe_help_get()	48
5.3.3.2 fpe_help_print()	48
5.3.3.3 stats_print()	48
5.4 /home/runner/work/FastPolyEval/FastPolyEval/code/apps/main.h File Reference	49
5.4.1 Detailed Description	50
5.5 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/concave.h File Reference	50
5.5.1 Detailed Description	51
5.5.2 Function Documentation	51
5.5.2.1 conc_free()	51
5.5.2.2 conc_new()	51
5.5.2.3 conc_range()	52
5.5.2.4 conc_range_der()	52
5.6 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/eval.h File Reference	53
5.6.1 Detailed Description	54

5.6.2 Typedef Documentation	54
5.6.2.1 eval_t	55
5.6.3 Function Documentation	55
5.6.3.1 eval_analyse()	55
5.6.3.2 eval_analyse_r()	55
5.6.3.3 eval_der()	56
5.6.3.4 eval_der_cc()	56
5.6.3.5 eval_der_cr()	57
5.6.3.6 eval_der_rc()	57
5.6.3.7 eval_der_rr()	58
5.6.3.8 eval_free()	58
5.6.3.9 eval_new()	59
5.6.3.10 eval_new_r()	59
5.6.3.11 eval_newton()	59
5.6.3.12 eval_newton_cc()	60
5.6.3.13 eval_newton_cr()	61
5.6.3.14 eval_newton_rc()	61
5.6.3.15 eval_newton_rr()	62
5.6.3.16 eval_val()	62
5.6.3.17 eval_val_cc()	63
5.6.3.18 eval_val_cr()	63
5.6.3.19 eval_val_der()	64
5.6.3.20 eval_val_der_cc()	64
5.6.3.21 eval_val_der_cr()	65
5.6.3.22 eval_val_der_rc()	65
5.6.3.23 eval_val_der_rr()	66
5.6.3.24 eval_val_rc()	66
5.6.3.25 eval_val_rr()	67
5.7 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/evalf.h File Reference	67
5.7.1 Detailed Description	69
5.7.2 Typedef Documentation	69
5.7.2.1 evalf_t	69
5.7.3 Function Documentation	69
5.7.3.1 evalf_analyse()	70
5.7.3.2 evalf_analyse_r()	70
5.7.3.3 evalf_der()	70
5.7.3.4 evalf_der_cc()	71
5.7.3.5 evalf_der_cr()	72
5.7.3.6 evalf_der_rc()	72
5.7.3.7 evalf_der_rr()	73
5.7.3.8 evalf_free()	73
5.7.3.9 evalf_new()	73

5.7.3.10 evalf_new_r()	74
5.7.3.11 evalf_newton()	74
5.7.3.12 evalf_newton_cc()	75
5.7.3.13 evalf_newton_cr()	75
5.7.3.14 evalf_newton_rc()	76
5.7.3.15 evalf_newton_rr()	76
5.7.3.16 evalf_val()	77
5.7.3.17 evalf_val_cc()	77
5.7.3.18 evalf_val_cr()	78
5.7.3.19 evalf_val_der()	78
5.7.3.20 evalf_val_der_cc()	79
5.7.3.21 evalf_val_der_cr()	79
5.7.3.22 evalf_val_der_rc()	80
5.7.3.23 evalf_val_der_rr()	80
5.7.3.24 evalf_val_rc()	81
5.7.3.25 evalf_val_rr()	81
5.8 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/list.h File Reference	82
5.8.1 Detailed Description	83
5.8.2 Typedef Documentation	83
5.8.2.1 list_t	83
5.8.3 Function Documentation	83
5.8.3.1 list_add()	83
5.8.3.2 list_clear()	84
5.8.3.3 list_clone()	84
5.8.3.4 list_free()	85
5.8.3.5 list_init()	85
5.8.3.6 list_new()	85
5.8.3.7 list_sort()	86
5.8.3.8 list_trim()	86
5.9 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/pows.h File Reference	86
5.9.1 Detailed Description	87
5.9.2 Typedef Documentation	87
5.9.2.1 pows_t	87
5.9.3 Function Documentation	87
5.9.3.1 pows_free()	87
5.9.3.2 pows_new()	88
5.9.3.3 pows_pow()	88
5.9.3.4 pows_pow_once()	89
5.9.3.5 pows_set()	89
5.10 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/powsf.h File Reference	89
5.10.1 Detailed Description	90
5.10.2 Typedef Documentation	90

5.10.2.1	powsf_t	90
5.10.3	Function Documentation	90
5.10.3.1	powsf_free()	90
5.10.3.2	powsf_new()	91
5.10.3.3	powsf_pow()	91
5.10.3.4	powsf_pow_once()	92
5.10.3.5	powsf_set()	92
5.11	/home/runner/work/FastPolyEval/FastPolyEval/code/eval/powsfr.h File Reference	93
5.11.1	Detailed Description	93
5.11.2	Typedef Documentation	93
5.11.2.1	powsfr_t	93
5.11.3	Function Documentation	94
5.11.3.1	powsfr_free()	94
5.11.3.2	powsfr_new()	94
5.11.3.3	powsfr_pow()	94
5.11.3.4	powsfr_pow_once()	95
5.11.3.5	powsfr_set()	95
5.12	/home/runner/work/FastPolyEval/FastPolyEval/code/eval/powsr.h File Reference	96
5.12.1	Detailed Description	96
5.12.2	Typedef Documentation	96
5.12.2.1	powsr_t	97
5.12.3	Function Documentation	97
5.12.3.1	powsr_free()	97
5.12.3.2	powsr_new()	97
5.12.3.3	powsr_pow()	98
5.12.3.4	powsr_pow_once()	98
5.12.3.5	powsr_set()	99
5.13	/home/runner/work/FastPolyEval/FastPolyEval/code/numbers/comp.h File Reference	99
5.13.1	Detailed Description	100
5.13.2	Macro Definition Documentation	100
5.13.2.1	comp_add	101
5.13.2.2	comp_addr	101
5.13.2.3	comp_amu	101
5.13.2.4	comp_clear	102
5.13.2.5	comp_div	102
5.13.2.6	comp_init	102
5.13.2.7	comp_initz	103
5.13.2.8	comp_mul	103
5.13.2.9	comp_muli	103
5.13.2.10	comp_mulr	104
5.13.2.11	comp_mulu	104
5.13.2.12	comp_neg	104

5.13.2.13 comp_set	105
5.13.2.14 comp_setr	105
5.13.2.15 comp_sqr	105
5.13.2.16 comp_sub	106
5.13.2.17 comp_subr	106
5.13.3 Typedef Documentation	106
5.13.3.1 comp	106
5.13.4 Function Documentation	106
5.13.4.1 comp_log2()	106
5.13.4.2 comp_s()	107
5.13.4.3 mpfr_s()	107
5.13.4.4 real_log2()	108
5.14 /home/runner/work/FastPolyEval/FastPolyEval/code/numbers/compf.h File Reference	108
5.14.1 Detailed Description	109
5.14.2 Macro Definition Documentation	109
5.14.2.1 compf_add	109
5.14.2.2 compf_addr	110
5.14.2.3 compf_amr	110
5.14.2.4 compf_div	110
5.14.2.5 compf_mod2	111
5.14.2.6 compf_mul	111
5.14.2.7 compf_mulr	111
5.14.2.8 compf_neg	112
5.14.2.9 compf_set	112
5.14.2.10 compf_setr	112
5.14.2.11 compf_sqr	112
5.14.2.12 compf_sub	113
5.14.2.13 compf_subr	113
5.14.3 Typedef Documentation	113
5.14.3.1 compf	113
5.15 /home/runner/work/FastPolyEval/FastPolyEval/code/numbers/ntypes.h File Reference	114
5.15.1 Detailed Description	115
5.15.2 Function Documentation	115
5.15.2.1 bits_sum()	115
5.15.2.2 nt_err()	116
5.16 /home/runner/work/FastPolyEval/FastPolyEval/code/poly/poly.h File Reference	116
5.16.1 Detailed Description	117
5.16.2 Typedef Documentation	117
5.16.2.1 poly_t	117
5.16.3 Function Documentation	117
5.16.3.1 poly_derivative()	117
5.16.3.2 poly_diff()	118

5.16.3.3 poly_eval()	118
5.16.3.4 poly_eval_r()	119
5.16.3.5 poly_free()	119
5.16.3.6 poly_from_roots()	119
5.16.3.7 poly_new()	120
5.16.3.8 poly_prod()	120
5.16.3.9 poly_set()	121
5.16.3.10 poly_sqr()	121
5.16.3.11 poly_sum()	121
5.17 /home/runner/work/FastPolyEval/FastPolyEval/code/poly/polyf.h File Reference	122
5.17.1 Detailed Description	123
5.17.2 Typedef Documentation	123
5.17.2.1 polyf_t	123
5.17.3 Function Documentation	123
5.17.3.1 polyf_derivative()	123
5.17.3.2 polyf_diff()	123
5.17.3.3 polyf_eval()	124
5.17.3.4 polyf_eval_r()	124
5.17.3.5 polyf_free()	125
5.17.3.6 polyf_from_roots()	125
5.17.3.7 polyf_new()	125
5.17.3.8 polyf_prod()	126
5.17.3.9 polyf_set()	126
5.17.3.10 polyf_sqr()	127
5.17.3.11 polyf_sum()	127
5.18 /home/runner/work/FastPolyEval/FastPolyEval/code/poly/polyfr.h File Reference	127
5.18.1 Detailed Description	128
5.18.2 Typedef Documentation	129
5.18.2.1 polyfr_t	129
5.18.3 Function Documentation	129
5.18.3.1 polyf_chheb()	129
5.18.3.2 polyf_her()	129
5.18.3.3 polyf_hyp()	130
5.18.3.4 polyf_lag()	130
5.18.3.5 polyf_leg()	130
5.18.3.6 polyfr_comp()	131
5.18.3.7 polyfr_derivative()	131
5.18.3.8 polyfr_diff()	131
5.18.3.9 polyfr_eval()	132
5.18.3.10 polyfr_eval_c()	132
5.18.3.11 polyfr_free()	133
5.18.3.12 polyfr_get()	133

5.18.3.13 polyfr_new()	133
5.18.3.14 polyfr_prod()	134
5.18.3.15 polyfr_set()	134
5.18.3.16 polyfr_sqr()	135
5.18.3.17 polyfr_sum()	135
5.19 /home/runner/work/FastPolyEval/FastPolyEval/code/poly/polyr.h File Reference	135
5.19.1 Detailed Description	136
5.19.2 Typedef Documentation	137
5.19.2.1 polyr_t	137
5.19.3 Function Documentation	137
5.19.3.1 poly_cheb()	137
5.19.3.2 poly_her()	137
5.19.3.3 poly_hyp()	138
5.19.3.4 poly_lag()	138
5.19.3.5 poly_leg()	139
5.19.3.6 polyr_derivative()	139
5.19.3.7 polyr_diff()	139
5.19.3.8 polyr_eval()	140
5.19.3.9 polyr_eval_c()	140
5.19.3.10 polyr_free()	141
5.19.3.11 polyr_new()	141
5.19.3.12 polyr_prod()	141
5.19.3.13 polyr_set()	142
5.19.3.14 polyr_seti()	142
5.19.3.15 polyr_sqr()	143
5.19.3.16 polyr_sum()	143
5.20 /home/runner/work/FastPolyEval/FastPolyEval/code/tools/array.h File Reference	143
5.20.1 Detailed Description	144
5.20.2 Typedef Documentation	145
5.20.2.1 array_t	145
5.20.3 Function Documentation	145
5.20.3.1 array_add()	145
5.20.3.2 array_append()	145
5.20.3.3 array_first_inf()	146
5.20.3.4 array_first_nan()	146
5.20.3.5 array_free()	147
5.20.3.6 array_get()	147
5.20.3.7 array_is_real()	147
5.20.3.8 array_new()	148
5.20.3.9 array_new_poly()	148
5.20.3.10 array_new_polyr()	148
5.20.3.11 array_poly()	149

5.20.3.12 array_plyr()	149
5.20.3.13 array_read()	150
5.20.3.14 array_write()	150
5.21 /home/runner/work/FastPolyEval/FastPolyEval/code/tools/arrayf.h File Reference	150
5.21.1 Detailed Description	152
5.21.2 Typedef Documentation	152
5.21.2.1 arrayf_t	152
5.21.3 Function Documentation	152
5.21.3.1 arrayf_add()	152
5.21.3.2 arrayf_append()	152
5.21.3.3 arrayf_first_inf()	154
5.21.3.4 arrayf_first_nan()	154
5.21.3.5 arrayf_free()	155
5.21.3.6 arrayf_get()	155
5.21.3.7 arrayf_is_real()	155
5.21.3.8 arrayf_new()	156
5.21.3.9 arrayf_new_polyf()	156
5.21.3.10 arrayf_new_polyfr()	156
5.21.3.11 arrayf_polyf()	157
5.21.3.12 arrayf_polyfr()	157
5.21.3.13 arrayf_read()	157
5.21.3.14 arrayf_write()	158
5.22 /home/runner/work/FastPolyEval/FastPolyEval/code/tools/chrono.h File Reference	158
5.22.1 Detailed Description	159
5.22.2 Function Documentation	159
5.22.2.1 lap()	159
5.22.2.2 millis()	160
5.22.2.3 nanos()	160
5.23 /home/runner/work/FastPolyEval/FastPolyEval/code/tools/debug.h File Reference	161
5.23.1 Detailed Description	161
5.23.2 Function Documentation	161
5.23.2.1 pc()	161
5.23.2.2 pcf()	162
5.23.2.3 pm()	162

Chapter 1

Fast Evaluation of Real and Complex Polynomials

This documentation is available online at <https://fvigneron.github.io/FastPolyEval>. It also exists in PDF form: [FastPolyEval_doc.pdf](#).

1.1 Abstract

FastPolyEval is a library, written in C, that aims at evaluating polynomials very efficiently, without compromising the accuracy of the result. It is based on the **FPE** algorithm (Fast Polynomial Evaluator) introduced in [reference \[1\]](#) (see Section [References, Contacts and Copyright](#)).

In **FastPolyEval**, the computations are done for real or complex numbers, in floating point arithmetic, with a fixed precision, which can be one of the machine types **FP32**, **FP64**, **FP80** or an arbitrary precision based on **MPFR**.

Evaluations are performed on arbitrary (finite...) sets of points in the complex plane or along the real line, without geometrical constraints.

The average speed-up achieved by **FastPolyEval** over Hörner's scheme is illustrated on Figure 1.

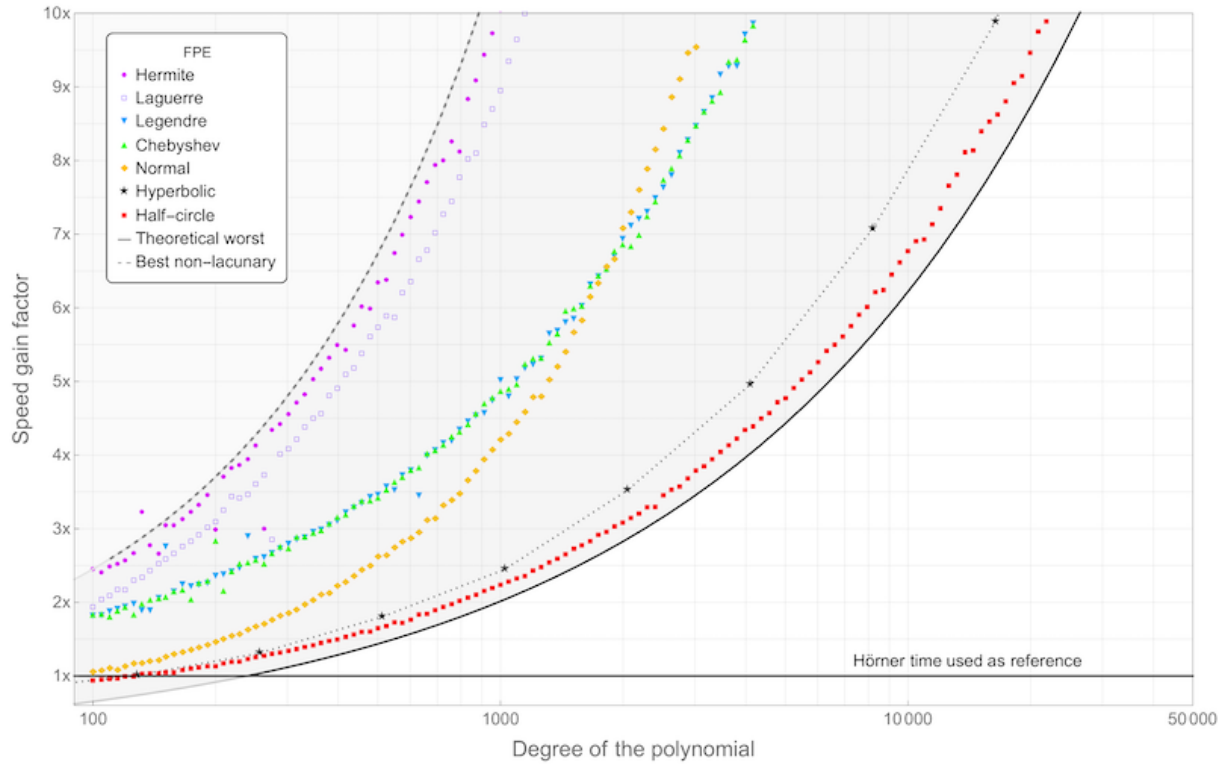


Figure 1.1 Speed gain of `FastPolyEval` versus Hörner for $O(d)$ evaluations in precision $p=53$ MPFR.

1.2 Complexity & accuracy

`FastPolyEval` splits the evaluation process in two phases:

- A first phase, called **pre-processing**, analyses the coefficients of the polynomial (actually only the exponents) and determines an evaluator, based on a parcimonious representation of the polynomial.
- Subsequently, the **evaluator** is applied to each of the requested evaluation points. A second reduction is performed. Then the final result is computed.

The complexity of the pre-processing phase is bounded by $O(d \log d)$ where d is the degree of the polynomial $P(z)$. It is independent of the precision used to express the coefficients or requested for the rest of the computations. The `FastPolyEval` library is backed by theoretical results that guaranty that the average arithmetic complexity of the final evaluator is of order

$$O\left(\sqrt{d(p + \log d)}\right)$$

where p is the precision of the computation, in bits. The averaging process corresponds to evaluation points z uniformly distributed either on the Riemann sphere or on the unit disk of the complex plane. The worst complexity of the evaluator does not exceed that of Hörner, i.e. $O(d)$, and can drop as low as $O(\log^2 d)$ in some favorable cases (see Figure 1 in [Abstract](#)).

The memory requirement of `FastPolyEval` is minimal. To handle a polynomial of degree d with p bits of precision, the memory requirement is $O(dp)$. An additional memory $O(kp)$ is required to perform k evaluations in one call.

Regarding accuracy, the theory guaranties that the relative error between the exact value of $P(z)$ and the computed value does not exceed 2^{-p-c-1} where c is the number of cancelled leading bits, i.e.

$$c = \begin{cases} 0 & \text{if } |P(z)| \geq M(z), \\ \lfloor \log_2 |M(z)| \rfloor - \lfloor \log_2 |P(z)| \rfloor & \text{else,} \end{cases}$$

where $M(z) = \max |a_j z^j|$ is the maximum modulus of the monomials of $P(z)$ and $\lfloor \cdot \rfloor$ is the floor function.

Note

The geometric preprocessing uses only the exponents of the coefficients, which is a significantly smaller amount of data to process than reading all the bits of the coefficients. For a high-precision high-degree evaluation at a **single** point, `FastPolyEval` can often outperform Hörner. The preprocessing of the exponents followed by the evaluation with a high precision of a parcimonious representation of the polynomial is more efficient than handling indiscriminately all the coefficients. This fact does not contradict that Hörner is a theoretical best for one single evaluation. Hörner has the best estimate on complexity that holds for **any** evaluation point. The advantage of `FastPolyEval` can be substantial, but it holds **on average**. Note that the worst case for the evaluator (no coefficient dropped) has the same complexity as Hörner. However, in that case, one can prove that the average complexity is much better, typically $O(\log^2 d)$.

For further details, precise statements and proofs, please see [reference \[1\]](#) in section [References, Contacts and Copyright](#).

1.3 Description of the Fast Polynomial Evaluator (FPE) algorithm

In this section, we describe briefly the mathematical principles at the foundation of `FastPolyEval`.

1.3.1 General principles

`FastPolyEval` relies on two general principles.

1. **Lazy evaluation** in finite precision. When adding two floating points numbers with p bits, one can simply discard the smaller one if the orders of magnitude are so far apart that the leading bit of the smaller number cannot affect the bit in the last position of the larger number. When evaluating a polynomial of large degree, the orders of magnitude of the monomials tends to be extremely varied, which is a strong incentive to use lazy evaluations.
2. **Geometric selection principle.** For a given evaluation point, evaluating each monomial and sorting them in decreasing order of magnitude would be a very inefficient way of implementing lazy additions. Instead, `FastPolyEval` can identify the leading monomials using a simple geometric method that allows us to factor most of the work in a pre-processing phase, which is only needed once. At each evaluation point, a subsequent reduction produces a very parcimonious representation of the polynomial (see Figure 2).

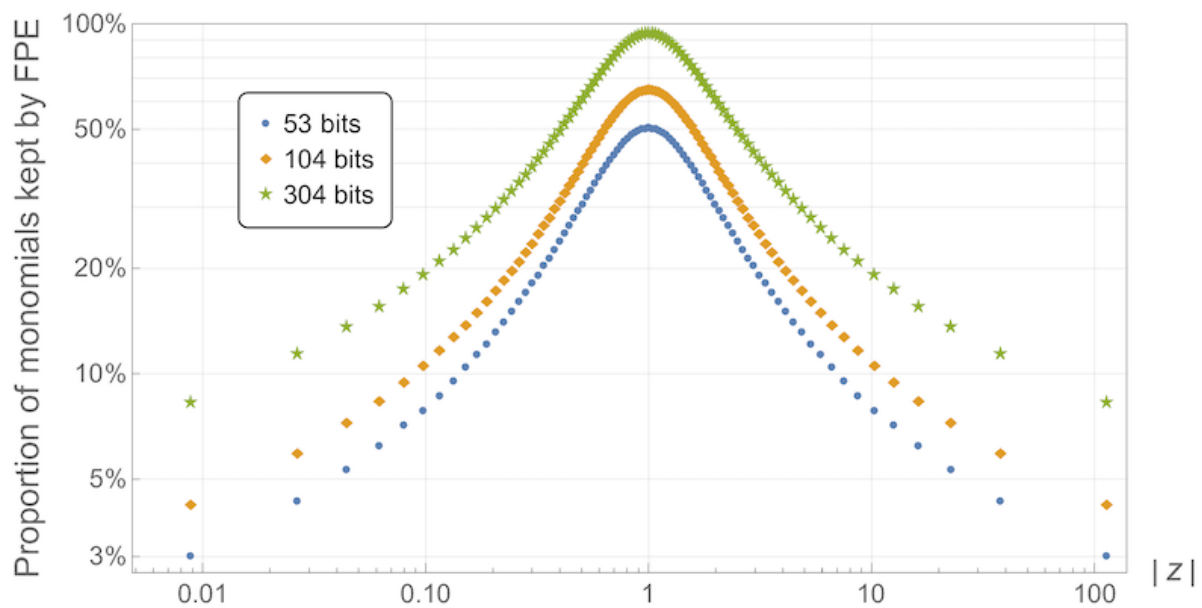


Figure 1.2 Parsimony of the representation used by `FastPolyEval` to evaluate a polynomial of degree 1000 (half-circle family).

1.3.2 Geometric selection principle

For a given polynomial P of degree d , one represents the modulus of the coefficients a_j in logarithmic coordinates, that is the scatter plot E_P of $\log_2 |a_j|$ in function of $j \in \{0, 1, \dots, d\}$. One then computes the concave envelope \hat{E}_P of E_P (obviously piecewise linear) and one identifies the strip $S_\delta(\hat{E}_P)$ situated below \hat{E}_P and of vertical thickness

$$\delta = p + \lfloor \log_2 d \rfloor + 4.$$

Note that the thickness of this strip is mostly driven by the precision p (in bits) that will be used to evaluate P . On Figure 3, the blue points represent E_P , the cyan line is \hat{E}_P and the strip S_δ is colored in light pink.

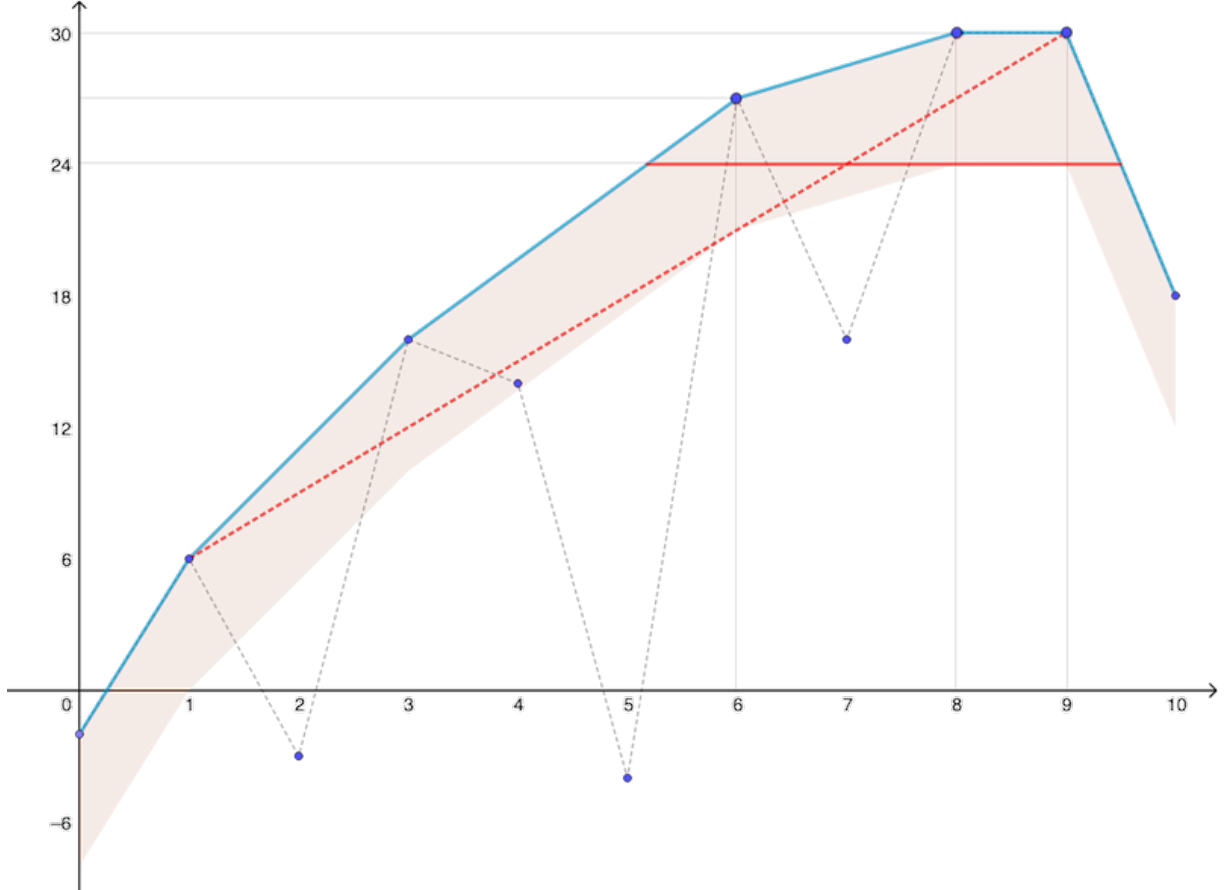


Figure 1.3 Concave cover of the coefficients of a polynomial (log-scale) and selection principle for a given precision.

The coefficients of P that lie in the strip S_δ are pertinent for an evaluation with precision p . The others can safely be discarded because they will never be used at this precision.

1.3.3 Parcimonious representation at an arbitrary point

For a given $z \in \mathbb{C}$, a parcimonious representation of $P(z) = \sum_{j=0}^d a_j z^j$ is obtained by selecting a proper subset of the indices $J_p(z) \subset \{0, 1, \dots, d\}$ and discarding the other monomials:

$$Q_p(z) = \sum_{j \in J_p(z)} a_j z^j.$$

For computations with a precision of p bits, the values of $Q_p(z)$ and $P(z)$ are equivalent because the relative error does not exceed 2^{-p-c-1} where c is the number of cancelled leading bits (defined in Section [Complexity & accuracy](#)).

The subset $J_p(z)$ is obtained as a subset of the (indices of the) coefficients that respect the two following criteria in the logarithmic representation (see Figure 3):

1. the coefficient belongs to the strip S_δ ,
2. the coefficient is above the longest segment of slope $\tan \theta = -\log_2 |z|$ contained in S_δ .

The selecting segment is always tangent to the lower edge of S_δ or, in case of ambiguity (e.g. if S_δ is a parallelogram), it is required to be. For $|z| = 1$, the selecting segment is an horizontal line illustrated in solid red in Figure 3 above. The red dashed line corresponds to some value $|z| < 1$. Figure 4 illustrates how the selection principle at an arbitrary point $z \in \mathbb{C}$ can be transposed from the analysis performed on the raw coefficients, i.e. for $|z| = 1$. In log-coordinates, one has indeed: $\log_2(|a_j z^j|) = \log_2 |a_j| - j \tan \theta$. Selecting the coefficients such that $\log_2(|a_j z^j|)$ differs from its maximum value by less than δ (solid red in Figure 4) is equivalent to selecting those above the segment of slope $\tan \theta$ in the original configuration (dashed line in Figure 3).

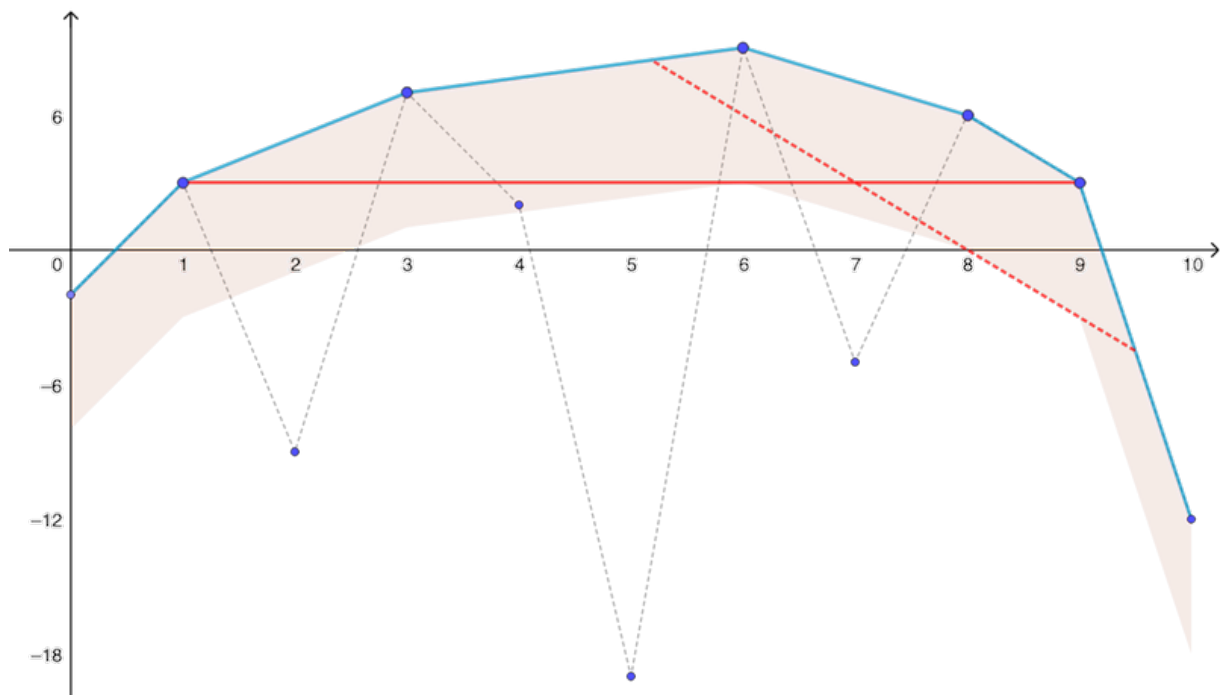


Figure 1.4 Transposition of the selection principle at an arbitrary evaluation point

1.3.4 Implementation of the algorithm

FastPolyEval computes Figure 3 and the strip S_δ in the pre-processing phase and keeps only the subset of the coefficients that belong to that strip (the so-called "good" set G_p in [reference \[1\]](#)). In the evaluation phase, the subset of coefficients is thinned to $J_p(z)$ for each evaluation point z using the 2nd criterion, thus providing the parcimonious representation $Q_p(z)$ of $P(z)$ at that point. The value of the polynomial is then computed using Hörner's method for Q_p . Theoretical results guaranty that, on average, the number of coefficients kept in $J_p(z)$ is only of order $\sqrt{d\delta}$, which explains why the complexity of FastPolyEval is so advantageous.

For further details, see [reference \[1\]](#) in Section [References, Contacts and Copyright](#).

1.4 Installation of FastPolyEval

FastPolyEval is distributed as source code, written in the C language. The source code can be downloaded from our public [github](#) repository.

1.4.1 Prerequisites

First of all, don't panic, it's easy...

Obvioulsy, you will need a computer and a C compiler, for example [GCC](#), though any other C compiler will be fine.

FastPolyEval relies on [MPFR](#) to handle floating point numbers with arbitrary precision. You need to install this library on your system if it is not already installed. Please follow the MPFR installation [instructions](#) to do so. Note that MPFR has one main depency, [GMP](#), which will also need to be installed (see the GMP installation [instructions](#)). If you do not have admin privileges on your machine, you can still build those libraries in a local directory and instruct your compiler to look for the appropriate library path. For example, if you use GCC, see the [documentation](#) of option `-L`.

Note that on most OS, one can also install those libraries through a package manager (see [Additional help for Linux, MacOS, Windows](#)

Note

The libraries GMP and MPFR are **free** to download and use and are distributed under the Lesser General Public License, which enables developers of non-free programs to use GMP/MPFR in their programs. FastPolyEval is released under the BSD licence, with an attribution clause (see [References, Contacts and Copyright](#)), which means that you are **free to download, use, modify, distribute or sell, provided you give us proper credit**.

1.4.2 Compile FastPolyEval

Download the [source code](#). In the command line, go to the `code` subfolder then type:

```
make
```

to get a basic help message. The default build command is:

```
make fpe
```

This will create and populate a `bin` subdirectory of the downloaded folder with an app, called FastPolyEval. You may move the executable file to a convenient location. Alternatively, consider adding this folder to your `PATH`. The Makefile does not attempt this step for you. See [Basic usage of FastPolyEval](#) for additional help on how to use this app.

If you prefer to compile by hand, the structure of your command line should be:

```
gcc all_source_files.c -o bin/FastPolyEval -I each_source_sub_folder_containing_header_files -Wall -lm -lmpfr -O3
```

Note that the `file.c` and `-I folder` parameters must be repeated as necessary. The Makefile is simple enough and understanding it can provide additional guidance.

FastPolyEval switches automatically from hardware machine precision to emulated high-precision using MPFR (see [Number formats](#)). The format used for hardware numbers is chosen at the time of the compilation. You may edit the source file [code/numbers/ntypes.h](#) before the default build. Alternatively, you can execute

```
make hardware
```

This should generate an automatically edited copy of this header file in a temporary folder and launch successive compilations. This build will populate the `bin` subdirectory with three apps, called FastPolyEval_FP32, FastPolyEval_FP64 and FastPolyEval_FP80 corresponding to each possible choice.

1.4.3 Additional help for Linux, MacOS, Windows

Github offers access to virtual machines that run on their servers. The corresponding tasks are called [workflows](#). Each workflow serves as a guideline (read the subsection `job:steps` in the `yml` files) to perform similar actions on your computer. The directory `.github/workflows` on our [GitHub repository](#) contains a `build` workflow for each major OS : Linux, MacOS and Windows.

1.4.4 Uninstall FastPolyEval

As the `Makefile` does not move the binaries and does not alter your `PATH`, cleanup is minimal. To remove the binary files generated in the build process, execute

```
make clean
```

To uninstall `FastPolyEval` from your system, simply delete the executable file and remove the downloaded sources.

1.5 Basic usage of FastPolyEval

`FastPolyEval` is used at the command line, either directly or through a shell script. A typical command line call has the following structure:

```
FastPolyEval -task prec parameter [optional_parameter]
```

where `prec` is the requested precision for the computations.

1.5.1 Onboard help system

To call the onboard help with a list and brief description of all possible tasks, type

```
FastPolyEval -help
```

A detailed help exists for each task. For example, try `FastPolyEval -eval -help`.

The different tasks belong to 3 groups:

- [Tasks for generating and handling polynomials](#),
- [Tasks for generating and handling sets of complex numbers](#),
- [Tasks using the FPE algorithm for production use and benchmarking](#)

and will be detailed below.

Note

In absence of argument, the default behavior is a call for help. The previous message can also be displayed respectively with `FastPolyEval` and `FastPolyEval -eval`.

1.5.2 Number formats

The flags `MACHINE_LOW_PREC` and `MACHINE_EXTRA_PREC` in [ntypes.h](#) define what kind of machine numbers `FastPolyEval` defaults to for low-precision computations, and the corresponding precision threshold that defines how low is "low". Note that the main limitation of machine numbers is the limited range of exponents. The numerical limits can easily be reached when evaluating a polynomial of high degree. On the contrary, MPFR number are virtually unlimited, with a default ceiling set to $\simeq 2^{\pm 4 \times 10^{18}}$.

Table 1.1 Number formats

numbers/ntypes.h	Precision requested	Format	Numerical limit
MACHINE_LOW_PREC	up to 24	FP32, float	about 10^{-38} to 10^{+38}
	25 and up	MPFR	practically unlimited
no flag	up to 53	FP64, double	about 10^{-308} to 10^{+308}
	54 and up	MPFR	practically unlimited
MACHINE_EXTRA_PREC	up to 64	FP80, long double	about 10^{-4930} to 10^{+4930}
	65 and up	MPFR	practically unlimited

Warning

If a number either read as input from a file or computed and destined to the output cannot be represented with the selected precision (`inf` or `NaN` exception), a warning is issued and the operation fails. Usually, the solution consists in using a higher precision (i.e. MPFR numbers). Another possible issue (if you are running Newton steps) is that you have entered the immediate neighborhood of a critical point, which induces a division by zero or almost zero. Try neutralizing the offending point.

Other numerical settings can be adjusted in [ntypes.h](#), in particular :

- `HUGE_DEGREES` sets the highest polynomial degree that can be handled (default $2^{64} - 2$),
- `HUGE_MP_EXP` sets the highest exponent that MPFR numbers can handle (default $\pm 4 \times 10^{18}$).

1.5.3 Input and outputs

`FastPolyEval` reads its input data from files and produces its output as file. The file format is CSV. To be valid, the rest of the CSV must contain a listing of complex numbers, written

`a, b`

where `a` is the real part and `b` is the imaginary part, in decimal form.

Depending on the context, the file will be interpreted as a set of evaluation points, a set of values, or the coefficients of a polynomial. For example, here is a valid file:

```
// An approximation of pi
3.14, 0
// A complex number close to the real axis
1.89755593218329076e+99, 2.35849881747969046e-427
```

Polynomials are written with the lowest degree first, so $P(z) = 1 - 2iz$ is represented by the file

```
1, 0
0, -2
```

Comments line are allowed: they start either with `#`, `//` or `;`. Comment lines will be ignored. Be mindful of the fact that comments can induce an **offset** between the line count of the file and the internal line count of `FastPolyEval`. Blank lines are not allowed. Lines are limited to 10 000 symbols (see [array_read](#)), which puts a practical limit to the precision at around 15 000 bits. You can easily edit the code to push this limitation if you have the need for it.

Warning

As a general rule for the tasks of `FastPolyEval`, if an output file name matches the one of an input file, the operation is **safe**. However, the desired output will overwrite the previous version of the file.

Note

For users that want a turnkey solution to high-performance polynomial evaluations where the outputs of some computations are fed back as input to others, we recommend the use of `FastPolyEval` with data residing in a virtual RAM disk, on local solid-state drives or on similar low latency/high bandwidth storage solutions. If you have doubts on how to do this, please contact your system administrator.

1.5.4 Tasks for generating and handling polynomials

FastPolyEval contains a comprehensive set of tools to generate new polynomials, either from scratch or by performing simple operations on other ones.

- Classical polynomials are generated with the tasks `-Chebyshev`, `-Legendre`, `-Hermite`, `-Laguerre`. The family `-hyperbolic` is defined recursively by

$$p_1(z) = z, \quad p_{n+1}(z) = p_n(z)^2 + z$$

and plays a central role in the study of the Mandelbrot set (see e.g. reference [2] in [References, Contacts and Copyright](#)).

- One can build a polynomial from a predetermined set of roots (listed with multiplicity) by invoking the task `-roots`.
- One can compute the sum (`-sum`), difference (`-diff`), product (`-prod`) or the derivative (`-der`) of polynomials.

1.5.5 Tasks for generating and handling sets of complex numbers

FastPolyEval contains a comprehensive set of tools to generate new sets of real and complex numbers, either from scratch or by performing simple operations on other ones.

- The real (`-re`) and imaginary (`-im`) parts of a list of complex numbers can be extracted. The results are real and therefore of the form `x, 0`. The complex conjugate (sign change of the imaginary part) is computed with `-conj` and the complex exponential with `-exp`.
- FastPolyEval can perform various interactions between valid CSV files.

- The concatenation (`-cat`) of two files writes a copy of the second one after a copy of the first one. For example, to rewrite a high-precision output `file_high_prec.csv` with a lower precision `prec`, use

```
FastPolyEval prec file_high_prec.csv /dev/null file_low_prec.csv
```

- The `-join` task takes the real parts of two sequences and builds a complex number out of it. The imaginary parts are lost.

```
a, *      (entry k from file_1)
b, *      (entry k from file_2)
----
a, b      (output k)
```

If the files have unequal length, the operation succeeds but stops once the shortest file runs out of entries. Be mindful of offsets induced by commented lines (see [Input and outputs](#)).

- The `-tensor` task computes a tensor product (i.e. complex multiplication line by line).

```
a, b      (entry k from file_1)
c, d      (entry k from file_2)
----
ac-bd, ad+bc (output k)
```

If the files have unequal length, the operation succeeds but stops once the shortest file runs out of entries. Be mindful of offsets induced by commented lines (see [Input and outputs](#)).

- The `-grid` task computes the grid product of the real part of the entries. The output file contains `nb_lines_file_1 × nb_lines_file_2` lines and lists all the products possibles between the real part of entries in the first file with the real part of entries in the second file. The imaginary parts are ignored.

- Rotations (`-rot`) map the complex number $a + ib$ to $a \times e^{ib}$. For example, to generate complex values from a real profile `profile.csv` and a set of phases `phases.csv`, you can use

```
FastPolyEval -join prec profile.csv phases.csv rot_tmp.csv
FastPolyEval -rot prec rot_tmp.csv complex_output.csv
rm -f rot_tmp.csv
```

- The `-unif` task writes real numbers in an arithmetic progression whose characteristics are specified by the parameters.
- The `-sphere` task writes polar coordinates approximating an uniform distribution on the [Riemann sphere](#). The output value (a, b) represents Euler angles in the (vertical, horizontal) convention. It represents the point

$$(\cos a \cos b, \cos a \sin b, \sin a) \in \mathbb{S}^2 \subset \mathbb{R}^3.$$

The `-polar` task maps a pair of (vertical, horizontal) angles onto the complex plane by [stereographical projection](#). To generate complex points that are uniformly distributed on the Riemann sphere, starting with `SEED` points at the equator, you can use the following code that combines those tasks. Note that there will be about $SEED^2/\pi$ points on the sphere and that the points are computed in a deterministic way (they will be identical from one call to the next). A `SEED` of 178 produces about 10 000 complex points.

```
SEED=178
FastPolyEval -sphere prec $SEED tmp_file.csv
FastPolyEval -polar prec tmp_file.csv sphere.csv
rm -f tmp_file.csv
```

- `FastPolyEval` also contains two random generators (actually based on MPFR). The `-rand` task produces real random numbers that are uniformly distributed in an interval. The `-normal` task produces real random numbers with a [Gaussian distribution](#) whose characteristics are specified by the parameters. Use the `-join` task to merge the outputs of two `-normal` tasks into a [complex valued normal distribution](#).

1.5.6 Tasks using the FPE algorithm for production use and benchmarking

The main tasks of the `FastPolyEval` library are the following:

- `-eval` evaluates a polynomial on a set of points using the FPE algorithm.
- `-evalD` evaluates the derivative of a polynomial on a set of points using the FPE algorithm.

The precision of the computation and the name of the inputs (polynomial and points) & output (values) files are given as parameters. The first optional parameter specifies a second output file that contains a complementary report on the estimated quality of the evaluation at the precision chosen. For each evaluation point, this report contains an upper bound for the evaluation errors (in bits), a conservative estimate on the number of correct bits of the result, and the number of terms that were kept by the FPE algorithm.

For benchmark purposes (see [reference \[1\]](#) in Section [References, Contacts and Copyright](#)), we propose a set of optional parameters that specify how many times the preprocessing and the FPE algorithm should be run at each point (to improve the accuracy of the time measurement), whether the Hörner algorithm should also be run and, if so, how many times it should run.

The task `-evalN` evaluates one **Newton step** of a polynomial on a set of points, i.e.

$$NS(P, z) = \frac{P(z)}{P'(z)}.$$

Recall that [Newton's method](#) for finding roots of P consists in computing the sequence defined recursively by

$$z_{n+1} = z_n - NS(P, z_n).$$

The `-iterN` task iterates the Newton method a certain amount of times and with a given precision. It thus realizes a partial search of the roots of the polynomial. For best results, we recommend successive calls where the precision is gradually increased and the maximum number of iterations is reduced. For guidance in the choice of the starting points, please refer to [reference \[3\]](#) in Section [References, Contacts and Copyright](#).

`-analyse` computes the concave cover and the intervals of $|z|$ for which the evaluation strategy changes

The `-analyse` task computes the concave cover \hat{E}_P , the strip S_δ (see Section [Description of the Fast Polynomial Evaluator \(FPE\)](#)) and the intervals of $|z|$ for which the evaluation strategy of FPE (i.e. the reduced polynomial $Q_p(z)$) changes. It is intended mostly for an illustrative purpose on low degrees, when the internals of the FPE algorithm can still be checked by hand. However, the intervals where a parsimonious representation is valid may also be of practical use (see [reference \[1\]](#) in Section [References, Contacts and Copyright](#)).

1.6 Notes about the implementation

For further details, please read the documentation associated to each [source file](#).

Do not hesitate to contact us (see [References, Contacts and Copyright](#)) to signal bugs and to request new features. We are happy to help the development of the community of the users of `FastPolyEval`.

1.7 References, Contacts and Copyright

Please cite the reference [1] below if you use or distribute this software. The other citations are in order of appearance in the text:

- [1] R. Anton, N. Mihalache & F. Vigneron. Fast evaluation of real and complex polynomials. 2022. [[hal-03820369](#)].
- [2] N. Mihalache & F. Vigneron. How to split a tera-polynomial. In preparation.
- [3] J.H. Hubbard, D. Schleicher, S. Sutherland. How to find all roots of complex polynomials by Newton's method. Invent. math., 146:1-33, 2001. [[Link](#)] on author's page.

Authors

Nicolae Mihalache – Université Paris-Est Créteil (nicolae.mihalache@u-pec.fr)

François Vigneron – Université de Reims Champagne-Ardenne (francois.vigneron@univ-reims.fr)

Copyright

This software is released under the BSD licence, with an attribution clause (see [License](#)).

Copyright 2022 – Université Paris-Est Créteil & Université de Reims Champagne-Ardenne.

1.8 Thanks

It is our pleasure to thank our families who supported us in the long and, at times stressful, process that gave birth to the `FastPolyEval` library, in particular Ramona who co-authored reference [1], Sarah for a thorough proof-reading of [1] and Anna who suggested the lightning pattern for our logo.

We also thank [Romeo](#), the HPC center of the University of Reims Champagne-Ardenne, on which we were able to benchmark our code graciously.

1.9 License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
4. Redistributions of any form whatsoever must retain the following acknowledgment:

'This product includes software developed by Nicolae Mihalache & François Vigneron at Univ. Paris-Est Créteil & Univ. de Reims Champagne-Ardenne. Please cite the following reference if you use or distribute this software: R. Anton, N. Mihalache & F. Vigneron. Fast evaluation of real and complex polynomials. 2022. [[hal-03820369](#)]'

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

array_struct	A variable length array of machine floating point complex numbers	17
arrayf_struct	A variable length list of machine floating point complex numbers	17
comp_struct	Multi-precision floating point complex numbers	18
compf_struct	Machine complex numbers	18
concave_struct	Description of a concave function computed from the coefficients of some polynomial	18
eval_struct	Evaluator of polynomials with multi-precision floating point coefficients	19
evalf_struct	Evaluator of polynomials with machine floating point coefficients	20
help_struct	Description of a basic help screen	21
list_struct	A list of real numbers that can be sorted, while keeping track of original order	21
poly_struct	Polynomial with multi-precision floating point complex coefficients	22
polyf_struct	Polynomial with machine floating point complex coefficients	22
polyfr_struct	Polynomial with machine floating point real coefficients	23
polyr_struct	Polynomial with multi-precision floating point complex coefficients	23
pows_struct	The powers of the complex number z using multi-precision floating point numbers	24
powsf_struct	The powers of the complex number z using machine floating point numbers	24
powsfr_struct	The powers of the real number x using machine floating point numbers	25
powsr_struct	The powers of the real number x using multi-precision floating point numbers	25

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

/home/runner/work/FastPolyEval/FastPolyEval/code/apps/ apps.h	
The implementation of the mini-apps with MPFR numbers	27
/home/runner/work/FastPolyEval/FastPolyEval/code/apps/ appsf.h	
The implementation of the mini-apps with machine numbers	38
/home/runner/work/FastPolyEval/FastPolyEval/code/apps/ help.h	
A basic help system for the mini-apps	46
/home/runner/work/FastPolyEval/FastPolyEval/code/apps/ main.h	
This is the entry point in the main app	49
/home/runner/work/FastPolyEval/FastPolyEval/code/eval/ concave.h	
Definition of a concave function computed from the coefficients of some polynomial	50
/home/runner/work/FastPolyEval/FastPolyEval/code/eval/ eval.h	
Definition of polynomial evaluator with arbitrary precision coefficients	53
/home/runner/work/FastPolyEval/FastPolyEval/code/eval/ evalf.h	
Definition of polynomials evaluator with machine floating point coefficients	67
/home/runner/work/FastPolyEval/FastPolyEval/code/eval/ list.h	
Definition of a list that can quickly sort real machine floating-point numbers and keep track of their permutation	82
/home/runner/work/FastPolyEval/FastPolyEval/code/eval/ pows.h	
Definition of a buffer for pre-computed powers of a complex number with arbitrary precision	86
/home/runner/work/FastPolyEval/FastPolyEval/code/eval/ powsf.h	
Definition of a buffer for pre-computed powers of a machine complex number	89
/home/runner/work/FastPolyEval/FastPolyEval/code/eval/ powsfr.h	
Definition of a buffer for pre-computed powers of a machine real number	93
/home/runner/work/FastPolyEval/FastPolyEval/code/eval/ powsr.h	
Definition of a buffer for pre-computed powers of a real number with arbitrary precision	96
/home/runner/work/FastPolyEval/FastPolyEval/code/numbers/ comp.h	
Definition of MPFR complex numbers	99
/home/runner/work/FastPolyEval/FastPolyEval/code/numbers/ compf.h	
Definition of machine complex numbers	108
/home/runner/work/FastPolyEval/FastPolyEval/code/numbers/ ntypes.h	
Definition of basic types	114
/home/runner/work/FastPolyEval/FastPolyEval/code/poly/ poly.h	
Definition of complex polynomials with arbitrary precision coefficients	116
/home/runner/work/FastPolyEval/FastPolyEval/code/poly/ polyf.h	
Definition of complex polynomials with machine floating point coefficients	122

/home/runner/work/FastPolyEval/FastPolyEval/code/poly/ polyfr.h	
Definition of real polynomials with machine floating point coefficients	127
/home/runner/work/FastPolyEval/FastPolyEval/code/poly/ polyr.h	
Definition of real polynomials with arbitrary precision coefficients	135
/home/runner/work/FastPolyEval/FastPolyEval/code/tools/ array.h	
Definition of a variable length array of arbitrary precision complex numbers that are based on	
mpfr	143
/home/runner/work/FastPolyEval/FastPolyEval/code/tools/ arrayf.h	
Definition of a variable length array of machine floating point complex numbers	150
/home/runner/work/FastPolyEval/FastPolyEval/code/tools/ chrono.h	
Tools for precise time measuring	158
/home/runner/work/FastPolyEval/FastPolyEval/code/tools/ debug.h	
A few basic tool for easy debug when usinf MPFR	161

Chapter 4

Data Structure Documentation

4.1 array_struct Struct Reference

A variable length array of machine floating point complex numbers.

Data Fields

- `ulong len`
the number of complex numbers in the array
- `ulong size`
the present capacity of the array (memory size in number of complex numbers)
- `comp_ptr zi`
the elements of the array

4.1.1 Detailed Description

A variable length array of machine floating point complex numbers.

Definition at line 37 of file array.h.

4.2 arrayf_struct Struct Reference

A variable length list of machine floating point complex numbers.

Data Fields

- `ulong len`
the number of complex numbers in the list
- `ulong size`
the present capacity of the list (memory size in number of complex numbers)
- `compf_ptr zi`
the elements of the list

4.2.1 Detailed Description

A variable length list of machine floating point complex numbers.

Definition at line 38 of file arrayf.h.

4.3 comp_struct Struct Reference

Multi-precision floating point complex numbers.

Data Fields

- `mpfr_t x`
the real part of the complex number
- `mpfr_t y`
the imaginary part of the complex number

4.3.1 Detailed Description

Multi-precision floating point complex numbers.

Definition at line 31 of file comp.h.

4.4 compf_struct Struct Reference

Machine complex numbers.

Data Fields

- `coeff_t x`
the real part of the complex number
- `coeff_t y`
the imaginary part of the complex number

4.4.1 Detailed Description

Machine complex numbers.

Definition at line 29 of file compf.h.

4.5 concave_struct Struct Reference

Description of a concave function computed from the coefficients of some polynomial.

Data Fields

- [prec_t extraBits](#)
extra bits for guarding, depending on the degree of the polynomial
- [prec_t prec](#)
the precision that will be used to evaluate the polynomial, excluding `extraBits`
- [list_t def](#)
the definition of the concave map
- [list_t all](#)
all terms of the polynomial that may be used for evaluation
- [deg_t start](#)
the position in `allPow` of the largest power to evaluate (for given slope)
- [deg_t mid](#)
the position in `allPow` of the power which gives the maximum modulus
- [deg_t end](#)
the position in `allPow` of the least power to evaluate (for given slope)

4.5.1 Detailed Description

Description of a concave function computed from the coefficients of some polynomial.

Definition at line 35 of file `concave.h`.

4.6 eval_struct Struct Reference

Evaluator of polynomials with multi-precision floating point coefficients.

Data Fields

- [poly P](#)
the polynomial, with complex coefficients
- [polyr Q](#)
the polynomial, with real coefficients
- [bool real](#)
the type of polynomial to evaluate
- [prec_t prec](#)
the precision of evaluations, in bits
- [concave f](#)
concave cover of the scales of coefficients
- [pows zn](#)
powers of a complex argument
- [powersr xn](#)
powers of a real argument
- [real_t valErr](#)
the [approximative] upper bound for the absolute error of the last evaluation, in bits
- [real_t derErr](#)
the [approximative] upper bound for the absolute error of the last derivative evaluation, in bits
- [real_t ntErr](#)

- the [approximative] upper bound for the absolute error of the last Newton term evaluation, in bits*
- `deg_t terms`
 - the number of polynomial terms computed by the last operation*
- `comp buf`
 - a buffer for internal computations*
- `mpfr_t br`
 - a real buffer for internal computations*

4.6.1 Detailed Description

Evaluator of polynomials with multi-precision floating point coefficients.

Definition at line 41 of file eval.h.

4.7 evalf_struct Struct Reference

Evaluator of polynomials with machine floating point coefficients.

Data Fields

- `polyf P`
 - the polynomial, with complex coefficients*
- `polyfr Q`
 - the polynomial, with real coefficients*
- `bool real`
 - the type of polynomial to evaluate*
- `concave f`
 - concave cover of the magnitude of coefficients*
- `powsf zn`
 - powers of a complex argument*
- `powsfr xn`
 - powers of a real argument*
- `real_t valErr`
 - the [approximative] upper bound for the absolute error of the last evaluation, in bits*
- `real_t derErr`
 - the [approximative] upper bound for the absolute error of the last derivative evaluation, in bits*
- `real_t ntErr`
 - the [approximative] upper bound for the absolute error of the last Newton term evaluation, in bits*
- `deg_t terms`
 - the number of polynomial terms computed by the last operation*

4.7.1 Detailed Description

Evaluator of polynomials with machine floating point coefficients.

Definition at line 35 of file evalf.h.

4.8 help_struct Struct Reference

Description of a basic help screen.

Data Fields

- char * `before`
the first line to display
- int `columnCount`
the number of columns
- int * `columnWidths`
the widths of all but the last column (array length should be 1 less than columnCount)
- char ** `headers`
the column headers
- int `linesCount`
the number of lines / commands
- char *** `lines`
the double array of messages
- char * `after`
the last line to display

4.8.1 Detailed Description

Description of a basic help screen.

Each help screen contains a brief summary (`before`), some `lines` of messages (commands) and a final discussion (`after`). Lines are structures as an array of variable size, with descriptive headers.

Definition at line 73 of file help.h.

4.9 list_struct Struct Reference

A list of real numbers that can be sorted, while keeping track of original order.

Data Fields

- `deg_t` `size`
the memory size allocated
- `deg_t` `count`
the number of elements
- `deg_t` * `k`
indexes, powers, the permutation of the numbers when sorted
- `real_t` * `s`
the list of numbers
- `bool` `sorted`
the state of the list

4.9.1 Detailed Description

A list of real numbers that can be sorted, while keeping track of original order.

Definition at line 35 of file list.h.

4.10 poly_struct Struct Reference

Polynomial with multi-precision floating point complex coefficients.

Data Fields

- `deg_t degree`
the degree of the polynomial
- `prec_t prec`
the precision of the coefficients, in bits
- `comp_ptr a`
the coefficients
- `bool modified`
the status of the coefficients
- `mpfr_t buf1`
a buffer
- `mpfr_t buf2`
another buffer

4.10.1 Detailed Description

Polynomial with multi-precision floating point complex coefficients.

Definition at line 31 of file poly.h.

4.11 polyf_struct Struct Reference

Polynomial with machine floating point complex coefficients.

Data Fields

- `deg_t degree`
the degree of the polynomial
- `compf_ptr a`
the coefficients
- `bool modified`
the status of the coefficients

4.11.1 Detailed Description

Polynomial with machine floating point complex coefficients.

Definition at line 30 of file polyf.h.

4.12 polyfr_struct Struct Reference

Polynomial with machine floating point real coefficients.

Data Fields

- `deg_t degree`
the degree of the polynomial
- `coeff_t * a`
the coefficients
- `bool modified`
the status of the coefficients

4.12.1 Detailed Description

Polynomial with machine floating point real coefficients.

Definition at line 31 of file polyfr.h.

4.13 polyr_struct Struct Reference

Polynomial with multi-precision floating point complex coefficients.

Data Fields

- `deg_t degree`
the degree of the polynomial
- `prec_t prec`
the precision of the coefficients, in bits
- `mpfr_ptr a`
the coefficients
- `bool modified`
the status of the coefficients
- `mpfr_t buf1`
a buffer
- `mpfr_t buf2`
another buffer

4.13.1 Detailed Description

Polynomial with multi-precision floating point complex coefficients.

Definition at line 31 of file polyr.h.

4.14 pows_struct Struct Reference

The powers of the complex number z using multi-precision floating point numbers.

Data Fields

- `prec_t prec`
the precision of the powers of z , in bits
- `deg_t size`
the memory size allocated
- `byte tps`
the largest non-negative integer such that $2^{tps} \leq size$
- `bool * computed`
the status of powers
- `comp_ptr zn`
the powers of z
- `mpfr_t buf1`
a buffer
- `mpfr_t buf2`
another buffer
- `comp pth`
a buffer
- `comp res`
another buffer

4.14.1 Detailed Description

The powers of the complex number z using multi-precision floating point numbers.

Definition at line 30 of file pows.h.

4.15 powsf_struct Struct Reference

The powers of the complex number z using machine floating point numbers.

Data Fields

- `deg_t size`
the memory size allocated
- `byte tps`
the largest non-negative integer such that $2^{tps} \leq size$
- `bool * computed`
the status of powers
- `compf_ptr zn`
the powers of z
- `compf res`
a buffer for results

4.15.1 Detailed Description

The powers of the complex number z using machine floating point numbers.

Definition at line 30 of file powsf.h.

4.16 powsfr_struct Struct Reference

The powers of the real number x using machine floating point numbers.

Data Fields

- `bool initd`
the status of the value x
- `coeff_t x`
the real number

4.16.1 Detailed Description

The powers of the real number x using machine floating point numbers.

Definition at line 35 of file powsfr.h.

4.17 powsr_struct Struct Reference

The powers of the real number x using multi-precision floating point numbers.

Data Fields

- `prec_t prec`
the precision of the powers of x , in bits
- `bool initd`
the status of the value x
- `mpfr_t x`
the real number
- `mpfr_t res`
a buffer

4.17.1 Detailed Description

The powers of the real number x using multi-precision floating point numbers.

Definition at line 37 of file `powsr.h`.

Chapter 5

File Documentation

5.1 /home/runner/work/FastPolyEval/FastPolyEval/code/apps/apps.h File Reference

The implementation of the mini-apps with MPFR numbers.

Functions

- `array app_join` (long prec, `array` z1, `array` z2)
Joins the real parts of the sequences `z1` and `z2` as the real parts and imaginary parts of a new list.
- `array app_grid` (long prec, `array` z1, `array` z2)
Computes the set product of the real parts of two sequences `z1` and `z2`.
- `array app_exp` (long prec, `array` z)
Computes the complex exponential of a sequence `z`.
- `array app_rot` (long prec, `array` z)
Computes $a \cdot \exp(ib)$ for each term $a+ib$ of a sequence `z` of complex numbers.
- `array app_polar` (long prec, `array` z)
Computes the complex numbers given by a sequence `z` of their polar coordinates on the sphere.
- `array app_unif` (long prec, `ulong` n, `mpfr_t` st, `mpfr_t` en)
Returns the list of `n` real numbers that split the interval `[st,en]` in `n-1` equal intervals.
- `array app_rand` (long prec, `ulong` n, `mpfr_t` st, `mpfr_t` en)
Returns the list of `n` random real numbers in the interval `[st,en]`, uniformly distributed.
- `array app_normal` (long prec, `ulong` n, `mpfr_t` cent, `mpfr_t` var)
Returns the list of `n` random real numbers, normally distributed with center `cent` and variance `var`.
- `array app_sphere` (long prec, `ulong` n)
Returns a list of `n` complex numbers, uniformly distributed on the Riemann sphere.
- `bool app_compare` (long prec, `array` z1, `array` z2, `char` *output)
Compares two list of complex values and reports the eventual errors.
- `array app_re` (long prec, `array` z)
Returns the real parts of the numbers in the sequence `z`.
- `array app_im` (long prec, `array` z)
Returns the imaginary parts of the numbers in the sequence `z`.
- `array app_conj` (long prec, `array` z)
Returns the conjugates of the numbers in the sequence `z`.

- `array app_tensor` (long prec, `array` z1, `array` z2)
Computes tensor (or piontwise) product of the sequences z1 and z2.
- `bool app_eval_p` (long prec, `poly` P, `array` z, char *outFile, char *outError, long count, char *outHorner, long countHorner, const char **inFiles)
*Evaluates the polynomial P in all points in the list z, using the **FPE** algorithm.*
- `bool app_eval_d` (long prec, `poly` P, `array` z, char *outFile, char *outError, long count, char *outHorner, long countHorner, const char **inFiles)
*Evaluates the derivative of the polynomial P in all points in the list z, using the **FPE** algorithm.*
- `bool app_eval_n` (long prec, `poly` P, `array` z, char *outFile, char *outError, long count, char *outHorner, long countHorner, const char **inFiles)
*Evaluates the Newton terms of the polynomial P in all points in the list z, using the **FPE** algorithm.*
- `bool app_eval_r` (long prec, `poly` P, `array` z, `ulong` maxIter, char *outFile, char *outError, long count, char *outHorner, long countHorner, const char **inFiles)
*Evaluates the Newton iterates of the polynomial P in all points in the list z, using the **FPE** algorithm.*
- `bool app_analyse` (long prec, `poly` P, char *outFile, char *outChange)
Analyses the polynomail P and outputs its concave cover, ignored coefficients and (optionally) the modules of complex numbers for which the reduced polynomial changes.

5.1.1 Detailed Description

The implementation of the mini-apps with MPFR numbers.

5.1.2 Function Documentation

5.1.2.1 app_analyse()

```
bool app_analyse (
    long prec,
    poly P,
    char * outFile,
    char * outChange )
```

Analyses the polynomail P and outputs its concave cover, ignored coefficients and (optionally) the modules of complex numbers for which the reduced polynomial changes.

These analysis is performed for the evaluations of the P with prec bits of precision. The concave cover corresponds to the pre-processing phase of the **FPE** algortihm. The map $k \rightarrow \log_2 |a_k|$ is considered.

Parameters

<i>prec</i>	the precision of evaluations
<i>P</i>	the polynomial
<i>outFile</i>	the output file containing the concave cover and the ognored coefficients
<i>outChange</i>	the output file containing the intervals for $\log_2 z $ and the corresponding reduced polynomials of P

Returns

`true` if the analysis is complete, `false` if some error occurred.

5.1.2.2 app_compare()

```
bool app_compare (
    long prec,
    array z1,
    array z2,
    char * output )
```

Compares two list of complex values and reports the eventual errors.

Parameters

<i>prec</i>	the precision of the result
<i>z1</i>	the fist list
<i>z2</i>	the second list
<i>output</i>	a file name for the output file, NULL if not used

Returns

`true` if the comparison is complete, `false` if some error occurred.

5.1.2.3 app_conj()

```
array app_conj (
    long prec,
    array z )
```

Returns the conjugates of the numbers in the sequence *z*.

Parameters

<i>prec</i>	the precision of the result
<i>z</i>	the list of points

Returns

the conjugates of *z*, NULL if some error occurred.

5.1.2.4 app_eval_d()

```
bool app_eval_d (
    long prec,
    poly P,
    array z,
    char * outFile,
    char * outError,
    long count,
    char * outHorner,
    long countHorner,
    const char ** inFiles )
```

Evaluates the derivative of the polynomial P in all points in the list z , using the **FPE** algorithm.

Parameters

<i>prec</i>	the precision of intermediary computations and of the result
<i>P</i>	the polynomial
<i>z</i>	the list of evaluation points
<i>outFile</i>	the results output file name
<i>outError</i>	the errors list file name
<i>count</i>	the number of repetitions of the computing task
<i>outHorner</i>	the results output file name, computed by Horner's method
<i>countHorner</i>	the number of repetitions of the computing task by Horner's method
<i>inFiles</i>	a vector with the names of input files, for printing stats only

Returns

true if the evaluaion is complete, **false** if some error occurred.

5.1.2.5 app_eval_n()

```
bool app_eval_n (
    long prec,
    poly P,
    array z,
    char * outFile,
    char * outError,
    long count,
    char * outHorner,
    long countHorner,
    const char ** inFiles )
```

Evaluates the Newton terms of the polynomial P in all points in the list z , using the **FPE** algorithm.

Parameters

<i>prec</i>	the precision of intermediary computations and of the result
<i>P</i>	the polynomial

Parameters

<i>z</i>	the list of evaluation points
<i>outFile</i>	the results output file name
<i>outError</i>	the errors list file name
<i>count</i>	the number of repetitions of the computing task
<i>outHorner</i>	the results output file name, computed by Horner's method
<i>countHorner</i>	the number of repetitions of the computing task by Horner's method
<i>inFiles</i>	a vector with the names of input files, for printing stats only

Returns

`true` if the evaluaion is complete, `false` if some error occurred.

5.1.2.6 app_eval_p()

```
bool app_eval_p (
    long prec,
    poly P,
    array z,
    char * outFile,
    char * outError,
    long count,
    char * outHorner,
    long countHorner,
    const char ** inFiles )
```

Evaluates the polynomial P in all points in the list z , using the **FPE** algorithm.

Parameters

<i>prec</i>	the precision of intermediary computations and of the result
<i>P</i>	the polynomial
<i>z</i>	the list of evaluation points
<i>outFile</i>	the results output file name
<i>outError</i>	the errors list file name
<i>count</i>	the number of repetitions of the computing task
<i>outHorner</i>	the results output file name, computed by Horner's method
<i>countHorner</i>	the number of repetitions of the computing task by Horner's method
<i>inFiles</i>	a vector with the names of input files, for printing stats only

Returns

`true` if the evaluaion is complete, `false` if some error occurred.

5.1.2.7 app_eval_r()

```
bool app_eval_r (
    long prec,
    poly P,
    array z,
    ulong maxIter,
    char * outFile,
    char * outError,
    long count,
    char * outHorner,
    long countHorner,
    const char ** inFiles )
```

Evaluates the Newton iterates of the polynomial P in all points in the list z , using the **FPE** algorithm.

This is a method to approximate the roots of P , but there are no guarantees for the convergence. If the iterates escape far from the origin, the algorithm stops (for those starting points).

Parameters

<i>prec</i>	the precision of intermediary computations and of the result
<i>P</i>	the polynomial
<i>z</i>	the list of evaluation points
<i>maxIter</i>	the maximum iterates for each starting point
<i>outFile</i>	the results output file name
<i>outError</i>	the errors list file name
<i>count</i>	the number of repetitions of the computing task
<i>outHorner</i>	the results output file name, computed by Horner's method
<i>countHorner</i>	the number of repetitions of the computing task by Horner's method
<i>inFiles</i>	a vector with the names of input files, for printing stats only

Returns

true if the evaluation is complete, **false** if some error occurred.

5.1.2.8 app_exp()

```
array app_exp (
    long prec,
    array z )
```

Computes the complex exponential of a sequence z .

Parameters

<i>prec</i>	the precision of the result
<i>z</i>	the list of points

Returns

the image by the complex exponential, `NULL` if some error occurred.

5.1.2.9 app_grid()

```
array app_grid (
    long prec,
    array z1,
    array z2 )
```

Computes the set product of the real parts of two sequences `z1` and `z2`.

Parameters

<i>prec</i>	the precision of the result
<i>z1</i>	the first list
<i>z2</i>	the second list

Returns

the set product, `NULL` if some error occurred.

5.1.2.10 app_im()

```
array app_im (
    long prec,
    array z )
```

Returns the imaginary parts of the numbers in the sequence `z`.

Parameters

<i>prec</i>	the precision of the result
<i>z</i>	the list of points

Returns

the imaginary parts in `z`, `NULL` if some error occurred.

5.1.2.11 app_join()

```
array app_join (
    long prec,
```

```

    array z1,
    array z2 )

```

Joins the real parts of the sequences `z1` and `z2` as the real parts and imaginary parts of a new list.

Parameters

<i>prec</i>	the precision of the result
<i>z1</i>	the first list
<i>z2</i>	the second list

Returns

the joint list, `NULL` if some error occurred.

5.1.2.12 `app_normal()`

```

array app_normal (
    long prec,
    ulong n,
    mpfr_t cent,
    mpfr_t var )

```

Returns the list of `n` random real numbers, normally distributed with center `cent` and variance `var`.

Parameters

<i>prec</i>	the precision of the result
<i>n</i>	the number of points, at least 2
<i>cent</i>	the center of the normal distribution
<i>var</i>	the variance of the normal distribution

Returns

the list of random real numbers (in complex format), `NULL` if some error occurred.

5.1.2.13 `app_polar()`

```

array app_polar (
    long prec,
    array z )

```

Computes the complex numbers given by a sequence `z` of their polar coordinates on the sphere.

Parameters

<i>prec</i>	the precision of the result
<i>z</i>	the list of polar coordinates

Returns

the complex numbers given by their polar coordinates, `NULL` if some error occurred.

5.1.2.14 `app_rand()`

```
array app_rand (
    long prec,
    ulong n,
    mpfr_t st,
    mpfr_t en )
```

Returns the list of *n* random real numbers in the interval [*st*,*en*], uniformly distributed.

Parameters

<i>prec</i>	the precision of the result
<i>n</i>	the number of points, at least 2
<i>st</i>	the start point
<i>en</i>	the end point

Returns

the list of random real numbers (in complex format), `NULL` if some error occurred.

5.1.2.15 `app_re()`

```
array app_re (
    long prec,
    array z )
```

Returns the real parts of the numbers in the sequence *z*.

Parameters

<i>prec</i>	the precision of the result
<i>z</i>	the list of points

Returns

the real parts in `z`, `NULL` if some error occurred.

5.1.2.16 app_rot()

```
array app_rot (
    long prec,
    array z )
```

Computes $a \cdot \exp(ib)$ for each term $a+ib$ of a sequence `z` of complex numbers.

Parameters

<i>prec</i>	the precision of the result
<i>z</i>	the list of points

Returns

the image by $a+ib \rightarrow a \cdot \exp(ib)$, `NULL` if some error occurred.

5.1.2.17 app_sphere()

```
array app_sphere (
    long prec,
    ulong n )
```

Returns a list of `n` complex numbers, uniformly distributed on the Riemann sphere.

Parameters

<i>prec</i>	the precision of the result
<i>n</i>	the number of points, at least 2

Returns

the list of points uniformly distributed on the sphere, `NULL` if some error occurred.

5.1.2.18 app_tensor()

```
array app_tensor (
    long prec,
```



```
array z1,  
array z2 )
```

Computes tensor (or piontwise) product of the sequences `z1` and `z2`.

Parameters

<i>prec</i>	the precision of the result
<i>z1</i>	the first list
<i>z2</i>	the second list

Returns

the tensor product, NULL if some error occurred.

5.1.2.19 app_unif()

```
array app_unif (
    long prec,
    ulong n,
    mpfr_t st,
    mpfr_t en )
```

Returns the list of n real numbers that split the interval $[st,en]$ in $n-1$ equal intervals.

Parameters

<i>prec</i>	the precision of the result
<i>n</i>	the number of points, at least 2
<i>st</i>	the start point
<i>en</i>	the end point

Returns

the list of real numbers (in complex format), NULL if some error occurred.

5.2 /home/runner/work/FastPolyEval/FastPolyEval/code/apps/appsf.h

File Reference

The implementation of the mini-apps with machine numbers.

Functions

- `arrayf appf_join (arrayf z1, arrayf z2)`
Joins the real parts of the sequences $z1$ and $z2$ as the real parts and imaginary parts of a new list.
- `arrayf appf_grid (arrayf z1, arrayf z2)`
Computes the set product of the real parts of two sequences $z1$ and $z2$.
- `arrayf appf_exp (arrayf z)`
Computes the complex exponential of a sequence z .

- `arrayf appf_rot (arrayf z)`
Computes $a \cdot \exp(ib)$ for each term $a+ib$ of a sequence z of complex numbers.
- `arrayf appf_polar (arrayf z)`
Computes the complex numbers given by a sequence z of their polar coordinates on the sphere.
- `arrayf appf_unif (ulong n, coeff_t st, coeff_t en)`
Returns the list of n real numbers that split the interval $[st, en]$ in $n-1$ equal intervals.
- `bool appf_compare (arrayf z1, arrayf z2, char *output)`
Compares two list of complex values and reports the eventual errors.
- `arrayf appf_re (arrayf z)`
Returns the real parts of the numbers in the sequence z .
- `arrayf appf_im (arrayf z)`
Returns the imaginary parts of the numbers in the sequence z .
- `arrayf appf_conj (arrayf z)`
Returns the conjugates of the numbers in the sequence z .
- `arrayf appf_tensor (arrayf z1, arrayf z2)`
Computes tensor (or pointwise) product of the sequences $z1$ and $z2$.
- `bool appf_eval_p (polyf P, arrayf z, char *outFile, char *outError, long count, char *outHorner, long count↔
Horner, const char **inFiles)`
Evaluates the polynomial P in all points in the list z , using the **FPE** algorithm.
- `bool appf_eval_d (polyf P, arrayf z, char *outFile, char *outError, long count, char *outHorner, long count↔
Horner, const char **inFiles)`
Evaluates the derivative of the polynomial P in all points in the list z , using the **FPE** algorithm.
- `bool appf_eval_n (polyf P, arrayf z, char *outFile, char *outError, long count, char *outHorner, long count↔
Horner, const char **inFiles)`
Evaluates the Newton terms of the polynomial P in all points in the list z , using the **FPE** algorithm.
- `bool appf_eval_r (polyf P, arrayf z, ulong maxIter, char *outFile, char *outError, long count, char *outHorner,
long countHorner, const char **inFiles)`
Evaluates the Newton iterates of the polynomial P in all points in the list z , using the **FPE** algorithm.

5.2.1 Detailed Description

The implementation of the mini-apps with machine numbers.

5.2.2 Function Documentation

5.2.2.1 appf_compare()

```
bool appf_compare (
    arrayf z1,
    arrayf z2,
    char * output )
```

Compares two list of complex values and reports the eventual errors.

Parameters

<code>z1</code>	the first list
<code>z2</code>	the second list
<code>output</code>	the name for the output file, NULL if not used

Returns

`true` if the comparison is complete, `false` if some error occurred.

5.2.2.2 appf_conj()

```
arrayf appf_conj (
    arrayf z )
```

Returns the conjugates of the numbers in the sequence `z`.

Parameters

<code>z</code>	the list of points
----------------	--------------------

Returns

the conjugates of `z`, `NULL` if some error occurred.

5.2.2.3 appf_eval_d()

```
bool appf_eval_d (
    polyf P,
    arrayf z,
    char * outFile,
    char * outError,
    long count,
    char * outHorner,
    long countHorner,
    const char ** inFiles )
```

Evaluates the derivative of the polynomial `P` in all points in the list `z`, using the **FPE** algorithm.

Parameters

<code>P</code>	the polynomial
<code>z</code>	the list of evaluation points
<code>outFile</code>	the results output file name
<code>outError</code>	the errors list file name
<code>count</code>	the number of repetitions of the computing task
<code>outHorner</code>	the results output file name, computed by Horner's method
<code>countHorner</code>	the number of repetitions of the computing task by Horner's method
<code>inFiles</code>	a vector with the names of input files, for printing stats only

Returns

`true` if the evaluaion is complete, `false` if some error occurred.

5.2.2.4 appf_eval_n()

```
bool appf_eval_n (
    polyf P,
    arrayf z,
    char * outFile,
    char * outError,
    long count,
    char * outHorner,
    long countHorner,
    const char ** inFiles )
```

Evaluates the Newton terms of the polynomial P in all points in the list z , using the **FPE** algorithm.

Parameters

P	the polynomial
z	the list of evaluation points
<i>outFile</i>	the results output file name
<i>outError</i>	the errors list file name
<i>count</i>	the number of repetitions of the computing task
<i>outHorner</i>	the results output file name, computed by Horner's method
<i>countHorner</i>	the number of repetitions of the computing task by Horner's method
<i>inFiles</i>	a vector with the names of input files, for printing stats only

Returns

`true` if the evaluaion is complete, `false` if some error occurred.

5.2.2.5 appf_eval_p()

```
bool appf_eval_p (
    polyf P,
    arrayf z,
    char * outFile,
    char * outError,
    long count,
    char * outHorner,
    long countHorner,
    const char ** inFiles )
```

Evaluates the polynomial P in all points in the list z , using the **FPE** algorithm.

Parameters

<i>P</i>	the polynomial
<i>z</i>	the list of evaluation points
<i>outFile</i>	the results output file name
<i>outError</i>	the errors list file name
<i>count</i>	the number of repetitions of the computing task
<i>outHorner</i>	the results output file name, computed by Horner's method
<i>countHorner</i>	the number of repetitions of the computing task by Horner's method
<i>inFiles</i>	a vector with the names of input files, for printing stats only

Returns

`true` if the evaluaion is complete, `false` if some error occurred.

5.2.2.6 appf_eval_r()

```
bool appf_eval_r (
    polyf P,
    arrayf z,
    ulong maxIter,
    char * outFile,
    char * outError,
    long count,
    char * outHorner,
    long countHorner,
    const char ** inFiles )
```

Evaluates the Newton iterates of the polynomial P in all points in the list z , using the **FPE** algorithm.

This is a method to approximate the roots of P , but there are no guarantees for the convergence. If the iterates escape far from the origin, the alorith stops (for those starting points).

Parameters

<i>P</i>	the polynomial
<i>z</i>	the list of evaluation points
<i>maxIter</i>	the maximum iterates for each starting point
<i>outFile</i>	the results output file name
<i>outError</i>	the errors list file name
<i>count</i>	the number of repetitions of the computing task
<i>outHorner</i>	the results output file name, computed by Horner's method
<i>countHorner</i>	the number of repetitions of the computing task by Horner's method
<i>inFiles</i>	a vector with the names of input files, for printing stats only

Returns

`true` if the evaluaion is complete, `false` if some error occurred.

5.2.2.7 appf_exp()

```
arrayf appf_exp (  
    arrayf z )
```

Computes the complex exponential of a sequence z .

Parameters

z	the list of points
-----	--------------------

Returns

the image by the complex exponential, `NULL` if some error occurred.

5.2.2.8 appf_grid()

```
arrayf appf_grid (  
    arrayf z1,  
    arrayf z2 )
```

Computes the set product of the real parts of two sequences $z1$ and $z2$.

Parameters

$z1$	the first list
$z2$	the second list

Returns

the set product, `NULL` if some error occurred.

5.2.2.9 appf_im()

```
arrayf appf_im (  
    arrayf z )
```

Returns the imaginary parts of the numbers in the sequence z .

Parameters

z	the list of points
-----	--------------------

Returns

the imaginary parts in `z`, `NULL` if some error occurred.

5.2.2.10 appf_join()

```
arrayf appf_join (  
    arrayf z1,  
    arrayf z2 )
```

Joins the real parts of the sequences `z1` and `z2` as the real parts and imaginary parts of a new list.

Parameters

<code>z1</code>	the first list
<code>z2</code>	the second list

Returns

the joint list, `NULL` if some error occurred.

5.2.2.11 appf_polar()

```
arrayf appf_polar (  
    arrayf z )
```

Computes the complex numbers given by a sequence `z` of their polar coordinates on the sphere.

Parameters

<code>z</code>	the list of polar coordinates
----------------	-------------------------------

Returns

the complex numbers given by their polar coordinates, `NULL` if some error occurred.

5.2.2.12 appf_re()

```
arrayf appf_re (  
    arrayf z )
```

Returns the real parts of the numbers in the sequence `z`.

Parameters

z	the list of points
----------	--------------------

Returns

the real parts in *z*, `NULL` if some error occurred.

5.2.2.13 appf_rot()

```
arrayf appf_rot (  
    arrayf z )
```

Computes $a \cdot \exp(ib)$ for each term $a+ib$ of a sequence *z* of complex numbers.

Parameters

z	the list of points
----------	--------------------

Returns

the image by $a+ib \rightarrow a \cdot \exp(ib)$, `NULL` if some error occurred.

5.2.2.14 appf_tensor()

```
arrayf appf_tensor (  
    arrayf z1,  
    arrayf z2 )
```

Computes tensor (or pointwise) product of the sequences *z1* and *z2*.

Parameters

z1	the first list
z2	the second list

Returns

the tensor product, `NULL` if some error occurred.

5.2.2.15 appf_unif()

```
arrayf appf_unif (
    ulong n,
    coeff_t st,
    coeff_t en )
```

Returns the list of n real numbers that split the interval $[st, en]$ in $n-1$ equal intervals.

Parameters

n	the number of points, at least 2
st	the start point
en	the end point

Returns

the list of real numbers (in complex format), NULL if some error occurred.

5.3 /home/runner/work/FastPolyEval/FastPolyEval/code/apps/help.h File Reference

A basic help system for the mini-apps.

Data Structures

- struct [help_struct](#)
Description of a basic help screen.

Macros

- #define [HELP_MAIN](#) 0
Indexes for help pages.
- #define [HELP_HYP](#) 1
- #define [HELP_CHEBYshev](#) 2
- #define [HELP_LEGENDRE](#) 3
- #define [HELP_HERMITE](#) 4
- #define [HELP_LAGUERRE](#) 5
- #define [HELP_SUM](#) 6
- #define [HELP_DIFF](#) 7
- #define [HELP_PROD](#) 8
- #define [HELP_CONCAT](#) 9
- #define [HELP_JOIN](#) 10
- #define [HELP_GRID](#) 11
- #define [HELP_EXP](#) 12
- #define [HELP_ROT](#) 13
- #define [HELP_POLAR](#) 14
- #define [HELP_ROOTS](#) 15
- #define [HELP_DER](#) 16

- `#define HELP_UNIF 17`
- `#define HELP_RAND 18`
- `#define HELP_NORM 19`
- `#define HELP_SPHERE 20`
- `#define HELP_EVAL 21`
- `#define HELP_EVALD 22`
- `#define HELP_EVALN 23`
- `#define HELP_NEWTON 24`
- `#define HELP_COMP 25`
- `#define HELP_RE 26`
- `#define HELP_IM 27`
- `#define HELP_CONJ 28`
- `#define HELP_TENSOR 29`
- `#define HELP_ANALYSE 30`
- `#define HELP_COUNT 31`
The number of help pages.
- `#define APP_COUNT (HELP_COUNT - 1)`
The number of mini-apps.

Typedefs

- `typedef help_struct * help`
Practical wrapper for `arrayf_struct`.

Functions

- `bool fpe_help_print (help h)`
Prints the help screen described by `h`.
- `help fpe_help_get (int ind)`
Return the help system in the list with index `ind`.
- `void stats_print (char *mes, long prec, deg_t deg, char *numType, bool real, char *polyFile, char *ptsFile, char *outFile, char *outError, double timePP, long count, double timeEV, ulong tpts, char *outHorner, double timeHo, ulong tptsho, ulong tit, ulong titHo)`
Prints computation stats in a standard format for automatic processing.

5.3.1 Detailed Description

A basic help system for the mini-apps.

5.3.2 Typedef Documentation

5.3.2.1 help

```
typedef help_struct* help
```

Practical wrapper for `arrayf_struct`.

To avoid the constant use `*` and `&` the type `help_t` is a pointer.

Definition at line 86 of file `help.h`.

5.3.3 Function Documentation

5.3.3.1 fpe_help_get()

```
help fpe_help_get (
    int ind )
```

Return the help system in the list with index `ind`.

Parameters

<code>ind</code>	the index, use constants <code>HELP_XXX</code>
------------------	--

Returns

the help system with the given index. Returns the list, `NULL` if `ind` is an invalid request.

5.3.3.2 fpe_help_print()

```
bool fpe_help_print (
    help h )
```

Prints the help screen described by `h`.

Parameters

<code>h</code>	the help screen to display
----------------	----------------------------

Returns

Boolean value expressing whether help could indeed be provided.

5.3.3.3 stats_print()

```
void stats_print (
    char * mes,
    long prec,
    deg_t deg,
    char * numType,
    bool real,
    char * polyFile,
    char * ptsFile,
```

```

char * outFile,
char * outError,
double timePP,
long count,
double timeEV,
ulong tpts,
char * outHorner,
double timeHo,
ulong tptsho,
ulong tit,
ulong titho )

```

Prints computation stats in a standard format for automatic processing.

Parameters

<i>mes</i>	a short description of the computing task
<i>prec</i>	the precision used, in bits
<i>deg</i>	the degree of the polynomial
<i>numType</i>	the number type
<i>real</i>	<code>true</code> if it is a real polynomial, <code>false</code> otherwise
<i>polyFile</i>	the polynomial file name
<i>ptsFile</i>	the points list file name
<i>outFile</i>	the results output file name
<i>outError</i>	the errors list file name
<i>timePP</i>	pre-processing time in ns
<i>count</i>	the number of repetitions of the computing task
<i>timeEV</i>	evaluation time in ns
<i>tpts</i>	the total number of evaluation points (taking into account the repetitions)
<i>outHorner</i>	the results output file name, computed by Horner's method
<i>timeHo</i>	the computing time in ns, by Horner's method
<i>tptsho</i>	the total number of evaluation by Horner's method
<i>tit</i>	total number of iterates, only for the iterative Newton method
<i>titho</i>	total number of iterates, only for the iterative Newton method, computed by Horner's method

5.4 /home/runner/work/FastPolyEval/FastPolyEval/code/apps/main.h File Reference

This is the entry point in the main app.

Macros

- `#define APP_OK 0`
- `#define APP_ERROR 1`
- `#define APP_PARAMS 2`

Typedefs

- typedef int(* [mainFunc](#)) (int argv, const char **args)
The signature of a main function of an app in this project with machine numbers.
- typedef int(* [mainFunc](#)) (long prec, int argv, const char **args)
The signature of a main function of an app in this project with arbitrary precision.

5.4.1 Detailed Description

This is the entry point in the main app.

Depending on the first command line parameter, it launches an internal app or presents a help screen to guide the user.

5.5 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/concave.h File Reference

Definition of a concave function computed from the coefficients of some polynomial.

Data Structures

- struct [concave_struct](#)
Description of a concave function computed from the coefficients of some polynomial.

Macros

- #define [BITS_GUARD](#) 6
The extra bits that are needed as explained in [1].

Typedefs

- typedef [concave_struct](#) * [concave](#)
Convenience pointer to [concave_struct](#).

Functions

- [concave conc_new](#) ([list](#) l, [prec_t](#) prec)
Computes the concave cover of the graph of a list l sorted in decreasing order.
- [bool conc_free](#) ([concave](#) f)
Frees all the memory used by the concave map f, assuming the struct has been allocated with malloc(), for example with [conc_new\(\)](#).
- [bool conc_range](#) ([concave](#) f, [real_t](#) la)
Computes the range of indexes in $f \rightarrow allPow$ to use for evaluating the original polynomial at z with $la = \log_2 |z|$.
- [bool conc_range_der](#) ([concave](#) f, [real_t](#) la)
Computes the range of indexes in $f \rightarrow allPow$ to use for evaluating the derivative of the original polynomial at z with $la = \log_2 |z|$.

5.5.1 Detailed Description

Definition of a concave function computed from the coefficients of some polynomial.

Admissible powers are for any of the original polynomial P or its derivative P' .

5.5.2 Function Documentation

5.5.2.1 conc_free()

```
bool conc_free (
    concave f )
```

Frees all the memory used by the concave map f , assuming the struct has been allocated with `malloc()`, for example with `conc_new()`.

Parameters

f	the map
-----	---------

Returns

`true` if successfull, `false` otherwise.

5.5.2.2 conc_new()

```
concave conc_new (
    list l,
    prec_t prec )
```

Computes the concave cover of the graph of a list l sorted in decreasing order.

Note

Retains the list of non-zero coefficients in `all`.

Parameters

l	the sorted list
$prec$	the precision that will be used to evaluate the polynomial

Returns

the concave function above the graph of l , NULL if some error occurred.

5.5.2.3 conc_range()

```
bool conc_range (
    concave f,
    real_t la )
```

Computes the range of indexes in $f \rightarrow \text{allPow}$ to use for evaluating the original polynomial at z with $la = \log_2 |z|$.

Stores the result in the $f \rightarrow \text{start}$ and respectively $f \rightarrow \text{end}$.

Note

$f \rightarrow \text{start} \geq f \rightarrow \text{end}$, otherwise the value of the polynomial is 0.

Parameters

f	the map
la	the slope, or $\log_2 z $

Returns

true if successfull, **false** otherwise.

5.5.2.4 conc_range_der()

```
bool conc_range_der (
    concave f,
    real_t la )
```

Computes the range of indexes in $f \rightarrow \text{allPow}$ to use for evaluating the derivative of the original polynomial at z with $la = \log_2 |z|$.

Stores the result in the $f \rightarrow \text{start}$ and respectively $f \rightarrow \text{end}$.

Note

$f \rightarrow \text{start} \geq f \rightarrow \text{end}$, otherwise the value of the derivative is 0.

Parameters

f	the map
la	the slope, or $\log_2 z $

Returns

`true` if successful, `false` otherwise.

5.6 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/eval.h File Reference

Definition of polynomial evaluator with arbitrary precision coefficients.

Data Structures

- struct `eval_struct`
Evaluator of polynomials with multi-precision floating point coefficients.

Macros

- #define `MIN_EVAL_PREC` 8
Minimum precision for evaluators.
- #define `MIN_EVAL_PREC_STR` "8"
- #define `eval_lastValError`(ev) (ev->valErr)
*The [approximative] upper bound for the error of the last evaluation, in bits. **Not** reported by the Newton term computation.*
- #define `eval_lastDerError`(ev) (ev->derErr)
*The [approximative] upper bound for the error of the last derivative evaluation, in bits. **Not** reported by the Newton term computation.*

Typedefs

- typedef `eval_struct eval_t`[1]
Practical wrapper for `eval_struct`.
- typedef `eval_struct * eval`
Convenience pointer to `eval_struct`.

Functions

- `eval eval_new` (poly P, `prec_t` prec)
Returns a new evaluator of the complex polynomial P.
- `eval eval_new_r` (polyr Q, `prec_t` prec)
Returns a new evaluator of the real polynomial Q.
- `bool eval_free` (eval ev)
Frees all the memory used by the evaluator ev, assuming the struct has been allocated with `malloc()`, for example with `eval_new()` or with `eval_new_r()`.
- `bool eval_val` (comp v, eval ev, comp z)
Evaluates $ev \rightarrow P(z)$ (or $ev \rightarrow Q(z)$) using the method described in [1].
- `bool eval_der` (comp d, eval ev, comp z)
Evaluates $ev \rightarrow P'(z)$ (or $ev \rightarrow Q'(z)$) using the method described in [1].
- `bool eval_val_der` (comp v, comp d, eval ev, comp z)

- Evaluates $ev \rightarrow P(z)$ and $ev \rightarrow P'(z)$ (or $ev \rightarrow Q(z)$ and $ev \rightarrow Q'(z)$) using the method described in [1].*

 - `bool eval_newton (comp nt, eval ev, comp z)`

Computes the Newton method step of $ev \rightarrow P$ (or $ev \rightarrow Q$) using the method described in [1].

 - `bool eval_analyse (eval ev)`

Analyses the complex polynomial $ev \rightarrow P$, after some of its coefficients have been changed.

 - `bool eval_analyse_r (eval ev)`

Analyses the real polynomial $ev \rightarrow Q$, after some of its coefficients have been changed.

 - `bool eval_val_cc (comp v, eval ev, comp z)`

Evaluates $ev \rightarrow P(x)$ using the method described in [1].

 - `bool eval_der_cc (comp d, eval ev, comp z)`

Evaluates $ev \rightarrow P'(x)$ using the method described in [1].

 - `bool eval_val_der_cc (comp v, comp d, eval ev, comp z)`

Evaluates $ev \rightarrow P(x)$ and $ev \rightarrow P'(x)$ using the method described in [1].

 - `bool eval_newton_cc (comp nt, eval ev, comp z)`

Computes the Newton method step of $ev \rightarrow P$ using the method described in [1].

 - `bool eval_val_cr (comp v, eval ev, mpfr_t x)`

Evaluates $ev \rightarrow P(x)$ using the method described in [1].

 - `bool eval_der_cr (comp d, eval ev, mpfr_t x)`

Evaluates $ev \rightarrow P'(x)$ using the method described in [1].

 - `bool eval_val_der_cr (comp v, comp d, eval ev, mpfr_t x)`

Evaluates $ev \rightarrow P(x)$ and $ev \rightarrow P'(x)$ using the method described in [1].

 - `bool eval_newton_cr (comp nt, eval ev, mpfr_t x)`

Computes the Newton method step of $ev \rightarrow P$ using the method described in [1].

 - `bool eval_val_rc (comp v, eval ev, comp z)`

Evaluates $ev \rightarrow Q(x)$ using the method described in [1].

 - `bool eval_der_rc (comp d, eval ev, comp z)`

Evaluates $ev \rightarrow Q'(x)$ using the method described in [1].

 - `bool eval_val_der_rc (comp v, comp d, eval ev, comp z)`

Evaluates $ev \rightarrow Q(x)$ and $ev \rightarrow Q'(x)$ using the method described in [1].

 - `bool eval_newton_rc (comp nt, eval ev, comp z)`

Computes the Newton method step of $ev \rightarrow Q$ using the method described in [1].

 - `bool eval_val_rr (mpfr_t v, eval ev, mpfr_t x)`

Evaluates the real polynomial $ev \rightarrow Q(x)$ using the method described in [1].

 - `bool eval_der_rr (mpfr_t d, eval ev, mpfr_t x)`

Evaluates the derivative of real polynomial $ev \rightarrow Q'(x)$ using the method described in [1].

 - `bool eval_val_der_rr (mpfr_t v, mpfr_t d, eval ev, mpfr_t x)`

Evaluates $ev \rightarrow Q(x)$ and $ev \rightarrow Q'(x)$ using the method described in [1].

 - `bool eval_newton_rr (mpfr_t nt, eval ev, mpfr_t x)`

Computes the Newton method step of the real polynomial $ev \rightarrow Q$ using the method described in [1].

5.6.1 Detailed Description

Definition of polynomial evaluator with arbitrary precision coefficients.

5.6.2 Typedef Documentation

5.6.2.1 eval_t

```
typedef eval_struct eval_t[1]
```

Practical wrapper for `eval_struct`.

To avoid the constant use `*` and `&` the type `eval_t` is a pointer.

Definition at line 60 of file `eval.h`.

5.6.3 Function Documentation

5.6.3.1 eval_analyse()

```
bool eval_analyse (  
    eval ev )
```

Analyses the complex polynomial $ev \rightarrow P$, after some of its coefficients have been changed.

Note

The general functions `eval_val()`, `eval_der()`, `eval_val_der()` and `eval_newton()` automatically analyse the appropriate polynomial. Use this function only with the optimized versions like `eval_val_cx()`.

Parameters

<code>ev</code>	the evaluator
-----------------	---------------

Returns

`true` if `ev` is ready to use, `false` otherwise.

5.6.3.2 eval_analyse_r()

```
bool eval_analyse_r (  
    eval ev )
```

Analyses the real polynomial $ev \rightarrow Q$, after some of its coefficients have been changed.

Note

The general functions `eval_val()`, `eval_der()`, `eval_val_der()` and `eval_newton()` automatically analyse the appropriate polynomial. Use this function only with the optimized versions like `eval_val_rx()`.

Parameters

<i>ev</i>	the evaluator
-----------	---------------

Returns

`true` if *ev* is ready to use, `false` otherwise.

5.6.3.3 `eval_der()`

```
bool eval_der (
    comp d,
    eval ev,
    comp z )
```

Evaluates $ev \rightarrow P'(z)$ (or $ev \rightarrow Q'(z)$) using the method described in [1].

Note

This function chooses the quickest variant to compute, depending if the polynomial of *ev* and / or *z* are real.

Parameters

<i>d</i>	the result
<i>ev</i>	the evaluator
<i>z</i>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.4 `eval_der_cc()`

```
bool eval_der_cc (
    comp d,
    eval ev,
    comp z )
```

Evaluates $ev \rightarrow P'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by *ev* nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<i>d</i>	the result
<i>ev</i>	the evaluator
<i>z</i>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.5 `eval_der_cr()`

```
bool eval_der_cr (
    comp d,
    eval ev,
    mpfr_t x )
```

Evaluates $ev \rightarrow P'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<i>d</i>	the result
<i>ev</i>	the evaluator
<i>x</i>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.6 `eval_der_rc()`

```
bool eval_der_rc (
    comp d,
    eval ev,
    comp z )
```

Evaluates $ev \rightarrow Q'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<i>d</i>	the result
<i>ev</i>	the evaluator
<i>z</i>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.7 `eval_der_rr()`

```
bool eval_der_rr (
    mpfr_t d,
    eval ev,
    mpfr_t x )
```

Evaluates the derivative of real polynomial $ev \rightarrow \mathbb{Q}'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<i>d</i>	the derivative
<i>ev</i>	the evaluator
<i>x</i>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.8 `eval_free()`

```
bool eval_free (
    eval ev )
```

Frees all the memory used by the evaluator `ev`, assuming the struct has been allocated with `malloc()`, for example with `eval_new()` or with `eval_new_r()`.

Parameters

<i>ev</i>	the evaluator
-----------	---------------

Returns

`true` if successfull, `false` otherwise.

5.6.3.9 eval_new()

```
eval eval_new (
    poly P,
    prec_t prec )
```

Returns a new evaluator of the complex polynomial P .

Parameters

P	the complex polynomial
$prec$	the precision of evaluations, in bits

Returns

the new evaluator, `NULL` if some error occurred.

5.6.3.10 eval_new_r()

```
eval eval_new_r (
    polyr Q,
    prec_t prec )
```

Returns a new evaluator of the real polynomial Q .

Parameters

Q	the real polynomial
$prec$	the precision of evaluations, in bits

Returns

the new evaluator, `NULL` if some error occurred.

5.6.3.11 eval_newton()

```
bool eval_newton (
    comp nt,
```

```
eval ev,
comp z )
```

Computes the Newton method step of $ev \rightarrow P$ (or $ev \rightarrow Q$) using the method described in [1].

Note

This function chooses the quickest variant to compute, depending if the polynomial of ev and / or z are real.

Parameters

<i>nt</i>	the Newton term
<i>ev</i>	the evaluator
<i>z</i>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.12 eval_newton_cc()

```
bool eval_newton_cc (
    comp nt,
    eval ev,
    comp z )
```

Computes the Newton method step of $ev \rightarrow P$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by ev nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<i>nt</i>	the Newton term
<i>ev</i>	the evaluator
<i>z</i>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.13 eval_newton_cr()

```
bool eval_newton_cr (
    comp nt,
    eval ev,
    mpfr_t x )
```

Computes the Newthton method step of $ev \rightarrow P$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by ev nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

nt	the Newton term
ev	the evaluator
x	the real argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.14 eval_newton_rc()

```
bool eval_newton_rc (
    comp nt,
    eval ev,
    comp z )
```

Computes the Newthton method step of $ev \rightarrow Q$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by ev nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

nt	the Newton term
ev	the evaluator
z	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.15 eval_newton_rr()

```
bool eval_newton_rr (
    mpfr_t nt,
    eval ev,
    mpfr_t x )
```

Computes the Newton method step of the real polynomial $ev \rightarrow \mathbb{Q}$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by ev nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

nt	the Newton term
ev	the evaluator
x	the real argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.16 eval_val()

```
bool eval_val (
    comp v,
    eval ev,
    comp z )
```

Evaluates $ev \rightarrow \mathbb{P}(z)$ (or $ev \rightarrow \mathbb{Q}(z)$) using the method described in [1].

Note

This function chooses the quickest variant to compute, depending if the polynomial of ev and / or z are real.

Parameters

v	the result
ev	the evaluator
z	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.17 eval_val_cc()

```
bool eval_val_cc (
    comp v,
    eval ev,
    comp z )
```

Evaluates $ev \rightarrow P(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>v</code>	the result
<code>ev</code>	the evaluator
<code>z</code>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.18 eval_val_cr()

```
bool eval_val_cr (
    comp v,
    eval ev,
    mpfr_t x )
```

Evaluates $ev \rightarrow P(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>v</code>	the result
<code>ev</code>	the evaluator
<code>x</code>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.19 eval_val_der()

```
bool eval_val_der (
    comp v,
    comp d,
    eval ev,
    comp z )
```

Evaluates $ev \rightarrow P(z)$ and $ev \rightarrow P'(z)$ (or $ev \rightarrow Q(z)$ and $ev \rightarrow Q'(z)$) using the method described in [1].

Note

This function chooses the quickest variant to compute, depending if the polynomial of ev and / or z are real.

Parameters

v	the value
d	the derivative
ev	the evaluator
z	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.20 eval_val_der_cc()

```
bool eval_val_der_cc (
    comp v,
    comp d,
    eval ev,
    comp z )
```

Evaluates $ev \rightarrow P(x)$ and $ev \rightarrow P'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by ev nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

v	the value
d	the derivative
ev	the evaluator
z	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.21 eval_val_der_cr()

```
bool eval_val_der_cr (
    comp v,
    comp d,
    eval ev,
    mpfr_t x )
```

Evaluates $ev \rightarrow P(x)$ and $ev \rightarrow P'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>v</code>	the value
<code>d</code>	the derivative
<code>ev</code>	the evaluator
<code>x</code>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.22 eval_val_der_rc()

```
bool eval_val_der_rc (
    comp v,
    comp d,
    eval ev,
    comp z )
```

Evaluates $ev \rightarrow Q(x)$ and $ev \rightarrow Q'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

v	the value
d	the derivative
ev	the evaluator
z	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.23 `eval_val_der_rr()`

```
bool eval_val_der_rr (
    mpfr_t v,
    mpfr_t d,
    eval ev,
    mpfr_t x )
```

Evaluates $ev \rightarrow Q(x)$ and $ev \rightarrow Q'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

v	the value
d	the derivative
ev	the evaluator
x	the real argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.24 `eval_val_rc()`

```
bool eval_val_rc (
    comp v,
    eval ev,
    comp z )
```

Evaluates $ev \rightarrow Q(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>v</code>	the result
<code>ev</code>	the evaluator
<code>z</code>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.6.3.25 eval_val_rr()

```
bool eval_val_rr (
    mpfr_t v,
    eval ev,
    mpfr_t x )
```

Evaluates the real polynomial $ev \rightarrow \mathbb{Q}(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>v</code>	the value
<code>ev</code>	the evaluator
<code>x</code>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.7 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/evalf.h File Reference

Definition of polynomials evaluator with machine floating point coefficients.

Data Structures

- struct `evalf_struct`
Evaluator of polynomials with machine floating point coefficients.

Macros

- #define `evalf_lastValError`(ev) (ev->valErr)
*The [approximative] upper bound for the error of the last evaluation, in bits. **Not** reported by the Newton term computation.*
- #define `evalf_lastDerError`(ev) (ev->derErr)
*The [approximative] upper bound for the error of the last derivative evaluation, in bits. **Not** reported by the Newton term computation.*

Typedefs

- typedef `evalf_struct evalf_t`[1]
Practical wrapper for `evalf_struct`.
- typedef `evalf_struct * evalf`
Convenience pointer to `evalf_struct`.

Functions

- `evalf evalf_new` (polyf P)
Returns a new evaluator of the complex polynomial P.
- `evalf evalf_new_r` (polyfr Q)
Returns a new evaluator of the real polynomial Q.
- `bool evalf_free` (evalf ev)
Frees all the memory used by the evaluator ev, assuming the struct has been allocated with `malloc()`, for example with `evalf_new()` or with `evalf_new_r()`.
- `bool evalf_val` (compf v, evalf ev, compf z)
Evaluates $ev \rightarrow P(z)$ (or $ev \rightarrow Q(z)$) using the method described in [1].
- `bool evalf_der` (compf d, evalf ev, compf z)
Evaluates $ev \rightarrow P'(z)$ (or $ev \rightarrow Q'(z)$) using the method described in [1].
- `bool evalf_val_der` (compf v, compf d, evalf ev, compf z)
Evaluates $ev \rightarrow P(z)$ and $ev \rightarrow P'(z)$ (or $ev \rightarrow Q(z)$ and $ev \rightarrow Q'(z)$) using the method described in [1].
- `bool evalf_newton` (compf nt, evalf ev, compf z)
Computes the Newton method step of $ev \rightarrow P$ (or $ev \rightarrow Q$) using the method described in [1].
- `bool evalf_analyse` (evalf ev)
Analyses the complex polynomial $ev \rightarrow P$, after some of its coefficients have been changed.
- `bool evalf_analyse_r` (evalf ev)
Analyses the real polynomial $ev \rightarrow Q$, after some of its coefficients have been changed.
- `bool evalf_val_cc` (compf v, evalf ev, compf z)
Evaluates $ev \rightarrow P(x)$ using the method described in [1].
- `bool evalf_der_cc` (compf d, evalf ev, compf z)
Evaluates $ev \rightarrow P'(x)$ using the method described in [1].
- `bool evalf_val_der_cc` (compf v, compf d, evalf ev, compf z)
Evaluates $ev \rightarrow P(x)$ and $ev \rightarrow P'(x)$ using the method described in [1].
- `bool evalf_newton_cc` (compf nt, evalf ev, compf z)

- Computes the Newthton method step of $ev \rightarrow P$ using the method described in [1].*
- `bool evalf_val_cr (compf v, evalf ev, coeff_t x)`
Evaluates $ev \rightarrow P(x)$ using the method described in [1].
- `bool evalf_der_cr (compf d, evalf ev, coeff_t x)`
Evaluates $ev \rightarrow P'(x)$ using the method described in [1].
- `bool evalf_val_der_cr (compf v, compf d, evalf ev, coeff_t x)`
Evaluates $ev \rightarrow P(x)$ and $ev \rightarrow P'(x)$ using the method described in [1].
- `bool evalf_newton_cr (compf nt, evalf ev, coeff_t x)`
Computes the Newthton method step of $ev \rightarrow P$ using the method described in [1].
- `bool evalf_val_rc (compf v, evalf ev, compf z)`
Evaluates $ev \rightarrow Q(x)$ using the method described in [1].
- `bool evalf_der_rc (compf d, evalf ev, compf z)`
Evaluates $ev \rightarrow Q'(x)$ using the method described in [1].
- `bool evalf_val_der_rc (compf v, compf d, evalf ev, compf z)`
Evaluates $ev \rightarrow Q(x)$ and $ev \rightarrow Q'(x)$ using the method described in [1].
- `bool evalf_newton_rc (compf nt, evalf ev, compf z)`
Computes the Newthton method step of $ev \rightarrow Q$ using the method described in [1].
- `coeff_t evalf_val_rr (evalf ev, coeff_t x)`
Evaluates the real polynomial $ev \rightarrow Q(x)$ using the method described in [1].
- `coeff_t evalf_der_rr (evalf ev, coeff_t x)`
Evaluates the derivative of real polynomial $ev \rightarrow Q'(x)$ using the method described in [1].
- `bool evalf_val_der_rr (coeff_t *v, coeff_t *d, evalf ev, coeff_t x)`
Evaluates $ev \rightarrow Q(x)$ and $ev \rightarrow Q'(x)$ using the method described in [1].
- `coeff_t evalf_newton_rr (evalf ev, coeff_t x)`
Computes the Newthton method step of the real polynomial $ev \rightarrow Q$ using the method described in [1].

5.7.1 Detailed Description

Definition of polynomials evaluator with machine floating point coefficients.

5.7.2 Typedef Documentation

5.7.2.1 evalf_t

```
typedef evalf_struct evalf_t[1]
```

Practical wrapper for `evalf_struct`.

To avoid the constant use `*` and `&` the type `evalf_t` is a pointer.

Definition at line 51 of file `evalf.h`.

5.7.3 Function Documentation

5.7.3.1 evalf_analyse()

```
bool evalf_analyse (
    evalf ev )
```

Analyses the complex polynomial $ev \rightarrow P$, after some of its coefficients have been changed.

Note

The general functions `evalf_val()`, `evalf_der()`, `evalf_val_der()` and `evalf_newton()` automatically analyse the appropriate polynomial. Use this function only with the optimized versions like `evalf_val_cx()`.

Parameters

<code>ev</code>	the evaluator
-----------------	---------------

Returns

`true` if `ev` is ready to use, `false` otherwise.

5.7.3.2 evalf_analyse_r()

```
bool evalf_analyse_r (
    evalf ev )
```

Analyses the real polynomial $ev \rightarrow Q$, after some of its coefficients have been changed.

Note

The general functions `evalf_val()`, `evalf_der()`, `evalf_val_der()` and `evalf_newton()` automatically analyse the appropriate polynomial. Use this function only with the optimized versions like `evalf_val_rx()`.

Parameters

<code>ev</code>	the evaluator
-----------------	---------------

Returns

`true` if `ev` is ready to use, `false` otherwise.

5.7.3.3 evalf_der()

```
bool evalf_der (
    compf d,
```

```

evalf ev,
compf z )

```

Evaluates $ev \rightarrow P'(z)$ (or $ev \rightarrow Q'(z)$) using the method described in [1].

Note

This function chooses the quickest variant to compute, depending if the polynomial of ev and / or z are real.

Parameters

d	the result
ev	the evaluator
z	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.4 evalf_der_cc()

```

bool evalf_der_cc (
    compf d,
    evalf ev,
    compf z )

```

Evaluates $ev \rightarrow P'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by ev nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

d	the result
ev	the evaluator
z	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.5 evalf_der_cr()

```
bool evalf_der_cr (
    compf d,
    evalf ev,
    coeff_t x )
```

Evaluates $ev \rightarrow P'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<i>d</i>	the result
<i>ev</i>	the evaluator
<i>x</i>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.6 evalf_der_rc()

```
bool evalf_der_rc (
    compf d,
    evalf ev,
    compf z )
```

Evaluates $ev \rightarrow Q'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<i>d</i>	the result
<i>ev</i>	the evaluator
<i>z</i>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.7 evalf_der_rr()

```
coeff_t evalf_der_rr (
    evalf ev,
    coeff_t x )
```

Evaluates the derivative of real polynomial $ev \rightarrow \mathbb{Q}'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>ev</code>	the evaluator
<code>x</code>	the real argument

Returns

the result, NaN is some error occurred.

5.7.3.8 evalf_free()

```
bool evalf_free (
    evalf ev )
```

Frees all the memory used by the evaluator `ev`, assuming the struct has been allocated with `malloc()`, for example with `evalf_new()` or with `evalf_new_r()`.

Parameters

<code>ev</code>	the evaluator
-----------------	---------------

Returns

`true` if successfull, `false` otherwise.

5.7.3.9 evalf_new()

```
evalf evalf_new (
    polyf P )
```

Returns a new evaluator of the complex polynomial P .

Parameters

<i>P</i>	the complex polynomial
----------	------------------------

Returns

the new evaluator, `NULL` if some error occurred.

5.7.3.10 evalf_new_r()

```
evalf evalf_new_r (
    polyfr Q )
```

Returns a new evaluator of the real polynomial `Q`.

Parameters

<i>Q</i>	the real polynomial
----------	---------------------

Returns

the new evaluator, `NULL` if some error occurred.

5.7.3.11 evalf_newton()

```
bool evalf_newton (
    compf nt,
    evalf ev,
    compf z )
```

Computes the Newton method step of $ev \rightarrow P$ (or $ev \rightarrow Q$) using the method described in [1].

Note

This function chooses the quickest variant to compute, depending if the polynomial of `ev` and / or `z` are real.

Parameters

<i>nt</i>	the Newton term
<i>ev</i>	the evaluator
<i>z</i>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.12 evalf_newton_cc()

```
bool evalf_newton_cc (
    compf nt,
    evalf ev,
    compf z )
```

Computes the Newthton method step of $ev \rightarrow P$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<i>nt</i>	the Newton term
<i>ev</i>	the evaluator
<i>z</i>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.13 evalf_newton_cr()

```
bool evalf_newton_cr (
    compf nt,
    evalf ev,
    coeff_t x )
```

Computes the Newthton method step of $ev \rightarrow P$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<i>nt</i>	the Newton term
<i>ev</i>	the evaluator
<i>x</i>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.14 evalf_newton_rc()

```
bool evalf_newton_rc (
    compf nt,
    evalf ev,
    compf z )
```

Computes the Newthton method step of $ev \rightarrow Q$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>nt</code>	the Newton term
<code>ev</code>	the evaluator
<code>z</code>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.15 evalf_newton_rr()

```
coeff_t evalf_newton_rr (
    evalf ev,
    coeff_t x )
```

Computes the Newthton method step of the real polynomial $ev \rightarrow Q$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>ev</code>	the evaluator
<code>x</code>	the real argument

Returns

the result, NaN is some error occurred.

5.7.3.16 evalf_val()

```
bool evalf_val (
    compf v,
    evalf ev,
    compf z )
```

Evaluates $ev \rightarrow P(z)$ (or $ev \rightarrow Q(z)$) using the method described in [1].

Note

This function chooses the quickest variant to compute, depending if the polynomial of ev and / or z are real.

Parameters

v	the result
ev	the evaluator
z	the complex argument

Returns

true if successfull, **false** otherwise.

5.7.3.17 evalf_val_cc()

```
bool evalf_val_cc (
    compf v,
    evalf ev,
    compf z )
```

Evaluates $ev \rightarrow P(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by ev nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

v	the result
ev	the evaluator
z	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.18 evalf_val_cr()

```
bool evalf_val_cr (
    compf v,
    evalf ev,
    coeff_t x )
```

Evaluates $ev \rightarrow P(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>v</code>	the result
<code>ev</code>	the evaluator
<code>x</code>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.19 evalf_val_der()

```
bool evalf_val_der (
    compf v,
    compf d,
    evalf ev,
    compf z )
```

Evaluates $ev \rightarrow P(z)$ and $ev \rightarrow P'(z)$ (or $ev \rightarrow Q(z)$ and $ev \rightarrow Q'(z)$) using the method described in [1].

Note

This function chooses the quickest variant to compute, depending if the polynomial of `ev` and / or `z` are real.

Parameters

<code>v</code>	the value
<code>d</code>	the derivative
<code>ev</code>	the evaluator
<code>z</code>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.20 evalf_val_der_cc()

```
bool evalf_val_der_cc (
    compf v,
    compf d,
    evalf ev,
    compf z )
```

Evaluates $ev \rightarrow P(x)$ and $ev \rightarrow P'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>v</code>	the value
<code>d</code>	the derivative
<code>ev</code>	the evaluator
<code>z</code>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.21 evalf_val_der_cr()

```
bool evalf_val_der_cr (
    compf v,
    compf d,
    evalf ev,
    coeff_t x )
```

Evaluates $ev \rightarrow P(x)$ and $ev \rightarrow P'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

v	the value
d	the derivative
ev	the evaluator
x	the real argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.22 `evalf_val_der_rc()`

```
bool evalf_val_der_rc (
    compf v,
    compf d,
    evalf ev,
    compf z )
```

Evaluates $ev \rightarrow Q(x)$ and $ev \rightarrow Q'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

v	the value
d	the derivative
ev	the evaluator
z	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.23 `evalf_val_der_rr()`

```
bool evalf_val_der_rr (
    coeff_t * v,
    coeff_t * d,
    evalf ev,
    coeff_t x )
```

Evaluates $ev \rightarrow Q(x)$ and $ev \rightarrow Q'(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>v</code>	the value
<code>d</code>	the derivative
<code>ev</code>	the evaluator
<code>x</code>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.24 evalf_val_rc()

```
bool evalf_val_rc (
    compf v,
    evalf ev,
    compf z )
```

Evaluates $ev \rightarrow \mathbb{Q}(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>v</code>	the result
<code>ev</code>	the evaluator
<code>z</code>	the complex argument

Returns

`true` if successfull, `false` otherwise.

5.7.3.25 evalf_val_rr()

```
coeff_t evalf_val_rr (
    evalf ev,
    coeff_t x )
```

Evaluates the real polynomial $ev \rightarrow \mathbb{Q}(x)$ using the method described in [1].

Warning

For maximum speed, no checks are performed on the parameters, on the type of polynomial represented by `ev` nor if the polynomial has been pre-conditionned after the last coefficient update.

Parameters

<code>ev</code>	the evaluator
<code>x</code>	the real argument

Returns

the result, NaN is some error occurred.

5.8 `/home/runner/work/FastPolyEval/FastPolyEval/code/eval/list.h` File Reference

Definition of a list that can quickly sort real machine floating-point numbers and keep track of their permutation.

Data Structures

- struct [list_struct](#)

A list of real numbers that can be sorted, while keeping track of original order.

Macros

- #define [LIST_FACTOR](#) 1.2

The multiplicative factor by which to grow the lists.

- #define [LIST_TERM](#) 100

The additive term by which to grow the lists.

Typedefs

- typedef [list_struct](#) [list_t](#)[1]

Practical wrapper for [list_struct](#).

- typedef [list_struct](#) * [list](#)

Convenience pointer to [list_struct](#).

Functions

- `list list_new (deg_t size)`
Returns a new list of machine real numbers, with storage space for *size* numbers.
- `bool list_init (list l, deg_t size)`
Initializes an existing list *l* with storage space for *size* numbers.
- `bool list_free (list l)`
Frees all the memory used by the list *l*, assuming the struct has been allocated with `malloc()`, for example with `list_new()`.
- `bool list_clear (list l)`
Frees all the memory used by the list *l*, but not the list *l* itself.
- `list list_clone (list l)`
Returns a copy of the list *l*.
- `bool list_add (list l, deg_t k, real_t s)`
Adds the couple (*k*,*s*) at a new position at the end of the list *l*.
- `bool list_trim (list l)`
Trims the list *l* to minimal size to contain all its elements.
- `bool list_sort (list l)`
Sorts the list in descending order while preserving couples ($k[i], s[i]$) for all *i*.

5.8.1 Detailed Description

Definition of a list that can quickly sort real machine floating-point numbers and keep track of their permutation.

5.8.2 Typedef Documentation

5.8.2.1 list_t

```
typedef list_struct list_t[1]
```

Practical wrapper for `list_struct`.

To avoid the constant use `*` and `&` the type `list_t` is a pointer.

Definition at line 46 of file `list.h`.

5.8.3 Function Documentation

5.8.3.1 list_add()

```
bool list_add (
    list l,
    deg_t k,
    real_t s )
```

Adds the couple (*k*,*s*) at a new position at the end of the list *l*.

Parameters

<i>l</i>	the list
<i>k</i>	the index
<i>s</i>	the real number

Returns

`true` if successfull, `false` otherwise.

5.8.3.2 list_clear()

```
bool list_clear (
    list l )
```

Frees all the memory used by the list `l`, but not the list `l` itself.

Parameters

<i>l</i>	the list
----------	----------

Returns

`true` if successfull, `false` otherwise.

5.8.3.3 list_clone()

```
list list_clone (
    list l )
```

Returns a copy of the list `l`.

Parameters

<i>l</i>	the list
----------	----------

Returns

the new list, `NULL` if some error occurred.

5.8.3.4 list_free()

```
bool list_free (
    list l )
```

Frees all the memory used by the list `l`, assuming the struct has been allocated with `malloc()`, for example with `list_new()`.

Parameters

<code>l</code>	the list
----------------	----------

Returns

`true` if successfull, `false` otherwise.

5.8.3.5 list_init()

```
bool list_init (
    list l,
    deg_t size )
```

Initializes an existing list `l` with storage space for `size` numbers.

Parameters

<code>l</code>	the list
<code>size</code>	the size of the list

Returns

`true` if successfull, `false` otherwise.

5.8.3.6 list_new()

```
list list_new (
    deg_t size )
```

Returns a new list of machine real numbers, with storage space for `size` numbers.

Parameters

<code>size</code>	the size of the list
-------------------	----------------------

Returns

the new list, `NULL` if some error occurred.

5.8.3.7 list_sort()

```
bool list_sort (
    list l )
```

Sorts the list in descending order while preserving couples $(k[i], s[i])$ for all i .

Warning

The memory usage will double during the operation, all cleared up before returning. In case of failure, the original list is destroyed.

Parameters

/	the list
---	----------

Returns

`true` if successfull, `false` otherwise.

5.8.3.8 list_trim()

```
bool list_trim (
    list l )
```

Trims the list `l` to minimal size to contain all its elements.

Parameters

/	the list
---	----------

Returns

`true` if successfull, `false` otherwise.

5.9 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/pows.h File Reference

Definition of a buffer for pre-computed powers of a complex number with arbitrary precision.

Data Structures

- struct `pows_struct`
The powers of the complex number z using multi-precision floating point numbers.

Typedefs

- typedef `pows_struct pows_t[1]`
Practical wrapper for `pows_struct`.
- typedef `pows_struct * pows`
Convenience pointer to `eval_struct`.

Functions

- `pows pows_new` (`prec_t` `prec`, `deg_t` `size`)
Returns a new buffer of powers of complex numbers of precision `prec`, with initial storage space for `size` powers.
- `bool pows_free` (`pows` `zn`)
Frees all the memory used by the buffer `zn`, assuming the struct has been allocated with `malloc()`, for example with `pows_new()`.
- `bool pows_set` (`pows` `zn`, `comp` `z`)
Sets the complex number of which the powers will be computed by the buffer `zn`.
- `comp_ptr pows_pow` (`pows` `zn`, `deg_t` `pow`)
Computes z^{pow} using repeated squares method and the cache of previously computed powers.
- `comp_ptr pows_pow_once` (`pows` `zn`, `deg_t` `pow`)
Computes z^{pow} using repeated squares method and the cache of previously computed powers.

5.9.1 Detailed Description

Definition of a buffer for pre-computed powers of a complex number with arbitrary precision.

5.9.2 Typedef Documentation

5.9.2.1 `pows_t`

```
typedef pows_struct pows_t[1]
```

Practical wrapper for `pows_struct`.

To avoid the constant use `*` and `&` the type `pows_t` is a pointer.

Definition at line 45 of file `pows.h`.

5.9.3 Function Documentation

5.9.3.1 `pows_free()`

```
bool pows_free (  
    pows zn )
```

Frees all the memory used by the buffer `zn`, assuming the struct has been allocated with `malloc()`, for example with `pows_new()`.

Parameters

<i>zn</i>	the powers buffer
-----------	-------------------

Returns

`true` if successfull, `false` otherwise.

5.9.3.2 `pows_new()`

```
pows pows_new (
    prec_t prec,
    deg_t size )
```

Returns a new buffer of powers of complex numbers of precision `prec`, with initial storage space for `size` powers.

Warning

`prec` must be at least `precf` and `size` at most `MAX_DEG`.

Parameters

<i>prec</i>	the precision of the coefficients, in bits
<i>size</i>	the size of the buffer

Returns

the new buffer, `NULL` if some error occurred.

5.9.3.3 `pows_pow()`

```
comp_ptr pows_pow (
    pows zn,
    deg_t pow )
```

Computes z^{pow} using repeated squares method and the cache of previously computed powers.

It caches intermediary powers of z that have been computed to accelerate later calls of this function. Also, if `pow` is larger than the size of the buffer `zn`, it is automatically increased to store the result.

Parameters

<i>zn</i>	the powers buffer
<i>pow</i>	the power to compute

Returns

the result z^{pow} , `NULL` if some error occurred.

5.9.3.4 pows_pow_once()

```
comp_ptr pows_pow_once (
    pows zn,
    deg_t pow )
```

Computes z^{pow} using repeated squares method and the cache of previously computed powers.

It does **NOT** cache intermediary powers of z into zn .

Parameters

<i>zn</i>	the powers buffer
<i>pow</i>	the power to compute

Returns

the result z^{pow} , `NULL` if some error occurred.

5.9.3.5 pows_set()

```
bool pows_set (
    pows zn,
    comp z )
```

Sets the complex number of which the powers will be computed by the buffer zn .

Parameters

<i>zn</i>	the powers buffer
<i>z</i>	the complex number

Returns

`true` if successfull, `false` otherwise.

5.10 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/powsf.h File Reference

Definition of a buffer for pre-computed powers of a machine complex number.

Data Structures

- struct `powsf_struct`

The powers of the complex number z using machine floating point numbers.

Typedefs

- typedef `powsf_struct powsf_t[1]`
Practical wrapper for `powsf_struct`.
- typedef `powsf_struct * powsf`
Convenience pointer to `eval_struct`.

Functions

- `powsf powsf_new (deg_t size)`
Returns a new buffer of powers of machine complex numbers, with initial storage space for `size` powers.
- `bool powsf_free (powsf zn)`
Frees all the memory used by the buffer `zn`, assuming the struct has been allocated with `malloc()`, for example with `powsf_new()`.
- `bool powsf_set (powsf zn, compf z)`
Sets the complex number of which the powers will be computed by the buffer `zn`.
- `compf_ptr powsf_pow (powsf zn, deg_t pow)`
Computes z^{pow} using repeated squares method and the cache of previously computed powers.
- `compf_ptr powsf_pow_once (powsf zn, deg_t pow)`
Computes z^{pow} using repeated squares method and the cache of previously computed powers.

5.10.1 Detailed Description

Definition of a buffer for pre-computed powers of a machine complex number.

5.10.2 Typedef Documentation

5.10.2.1 `powsf_t`

```
typedef powsf_struct powsf_t[1]
```

Practical wrapper for `powsf_struct`.

To avoid the constant use `*` and `&` the type `powsf_t` is a pointer.

Definition at line 41 of file `powsf.h`.

5.10.3 Function Documentation

5.10.3.1 `powsf_free()`

```
bool powsf_free (  
    powsf zn )
```

Frees all the memory used by the buffer `zn`, assuming the struct has been allocated with `malloc()`, for example with `powsf_new()`.

Parameters

<code>zn</code>	the powers buffer
-----------------	-------------------

Returns

`true` if successfull, `false` otherwise.

5.10.3.2 `powsf_new()`

```
powsf powsf_new (  
    deg_t size )
```

Returns a new buffer of powers of machine complex numbers, with initial storage space for `size` powers.

Warning

`size` must be at most `MAX_DEG`.

Parameters

<code>size</code>	the size of the buffer
-------------------	------------------------

Returns

the new buffer, `NULL` if some error occurred.

5.10.3.3 `powsf_pow()`

```
compf_ptr powsf_pow (  
    powsf zn,  
    deg_t pow )
```

Computes z^{pow} using repeated squares method and the cache of previously computed powers.

It caches intermediary powers of z that have been computed to accelerate later calls of this function. Also, if `pow` is larger than the size of the buffer `zn`, it is automatically increased to store the result.

Warning

fails if `pow==0`

Parameters

<i>zn</i>	the powers buffer
<i>pow</i>	the power to compute

Returns

the result z^{pow} , NULL if some error occurred.

5.10.3.4 powsf_pow_once()

```
compf_ptr powsf_pow_once (
    powsf zn,
    deg_t pow )
```

Computes z^{pow} using repeated squares method and the cache of previously computed powers.

It does **NOT** cache intermediary powers of z into zn .

Parameters

<i>zn</i>	the powers buffer
<i>pow</i>	the power to compute

Returns

the result z^{pow} , NULL if some error occurred.

5.10.3.5 powsf_set()

```
bool powsf_set (
    powsf zn,
    compf z )
```

Sets the complex number of which the powers will be computed by the buffer zn .

Parameters

<i>zn</i>	the powers buffer
<i>z</i>	the complex number

Returns

true if successfull, **false** otherwise.

5.11 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/powsfr.h File Reference

Definition of a buffer for pre-computed powers of a machine real number.

Data Structures

- struct [powsfr_struct](#)
The powers of the real number x using machine floating point numbers.

Typedefs

- typedef [powsfr_struct](#) [powsfr_t](#)[1]
Practical wrapper for [powsfr_struct](#).
- typedef [powsfr_struct](#) * [powsfr](#)
Convenience pointer to [eval_struct](#).

Functions

- [powsfr](#) [powsfr_new](#) ([deg_t](#) size)
*Returns a new buffer of powers of machine real numbers, with initial storage space for *size* powers.*
- [bool](#) [powsfr_free](#) ([powsfr](#) xn)
*Frees all the memory used by the buffer *xn*, assuming the struct has been allocated with `malloc()`, for example with [powsfr_new\(\)](#).*
- [bool](#) [powsfr_set](#) ([powsfr](#) xn, [coeff_t](#) x)
*Sets the real number of which the powers will be computed by the buffer *xn*.*
- [coeff_t](#) [powsfr_pow](#) ([powsfr](#) xn, [deg_t](#) pow)
Computes x^{pow} using repeated squares method and the cache of previously computed powers.
- [coeff_t](#) [powsfr_pow_once](#) ([powsfr](#) xn, [deg_t](#) pow)
Computes x^{pow} using repeated squares method and the cache of previously computed powers.

5.11.1 Detailed Description

Definition of a buffer for pre-computed powers of a machine real number.

5.11.2 Typedef Documentation

5.11.2.1 [powsfr_t](#)

```
typedef powsfr\_struct powsfr\_t[1]
```

Practical wrapper for [powsfr_struct](#).

To avoid the constant use * and & the type [powsfr_t](#) is a pointer.

Definition at line 56 of file [powsfr.h](#).

5.11.3 Function Documentation

5.11.3.1 powsfr_free()

```
bool powsfr_free (
    powsfr xn )
```

Frees all the memory used by the buffer `xn`, assuming the struct has been allocated with `malloc()`, for example with `powsfr_new()`.

Parameters

<code>xn</code>	the powers buffer
-----------------	-------------------

Returns

`true` if successfull, `false` otherwise.

5.11.3.2 powsfr_new()

```
powsfr powsfr_new (
    deg_t size )
```

Returns a new buffer of powers of machine real numbers, with initial storage space for `size` powers.

Warning

`size` must be at most `MAX_DEG`.

Parameters

<code>size</code>	the size of the buffer
-------------------	------------------------

Returns

the new buffer, `NULL` if some error occurred.

5.11.3.3 powsfr_pow()

```
coeff_t powsfr_pow (
    powsfr xn,
    deg_t pow )
```

Computes x^{pow} using repeated squares method and the cache of previously computed powers.

It caches intermediary powers of x that have been computed to accelerate later calls of this function. Also, if `pow` is larger than the size of the buffer `xn`, it is automatically increased to store the result.

Parameters

<code>xn</code>	the powers buffer
<code>pow</code>	the power to compute

Returns

the result x^{pow} , NaN if some error occurred.

5.11.3.4 powsfr_pow_once()

```
coeff_t powsfr_pow_once (
    powsfr xn,
    deg_t pow )
```

Computes x^{pow} using repeated squares method and the cache of previously computed powers.

It does **NOT** cache intermediary powers of x .

Parameters

<code>xn</code>	the powers buffer
<code>pow</code>	the power to compute

Returns

the result x^{pow} , NULL if some error occurred.

5.11.3.5 powsfr_set()

```
bool powsfr_set (
    powsfr xn,
    coeff_t x )
```

Sets the real number of which the powers will be computed by the buffer `xn`.

Parameters

<code>xn</code>	the powers buffer
<code>x</code>	the real number

Returns

`true` if successfull, `false` otherwise.

5.12 /home/runner/work/FastPolyEval/FastPolyEval/code/eval/powsr.h File Reference

Definition of a buffer for pre-computed powers of a real number with arbitrary precision.

Data Structures

- struct `powsr_struct`
The powers of the real number x using multi-precision floating point numbers.

Typedefs

- typedef `powsr_struct powsr_t[1]`
Practical wrapper for `powsr_struct`.
- typedef `powsr_struct * powsr`
Convenience pointer to `eval_struct`.

Functions

- `powsr powsr_new` (`prec_t` `prec`, `deg_t` `size`)
Returns a new buffer of powers of real numbers of precision `prec`, with initial storage space for `size` powers.
- `bool powsr_free` (`powsr` `xn`)
Frees all the memory used by the buffer `xn`, assuming the struct has been allocated with `malloc()`, for example with `powsr_new()`.
- `bool powsr_set` (`powsr` `xn`, `mpfr_t` `x`)
Sets the real number of which the powers will be computed by the buffer `xn`.
- `mpfr_ptr powsr_pow` (`powsr` `xn`, `deg_t` `pow`)
Computes x^{pow} using repeated squares method and the cache of previously computed powers.
- `mpfr_ptr powsr_pow_once` (`powsr` `xn`, `deg_t` `pow`)
Computes x^{pow} using repeated squares method and the cache of previously computed powers.

5.12.1 Detailed Description

Definition of a buffer for pre-computed powers of a real number with arbitrary precision.

5.12.2 Typedef Documentation

5.12.2.1 powsr_t

```
typedef powsr_struct powsr_t[1]
```

Practical wrapper for `powsr_struct`.

To avoid the constant use `*` and `&` the type `powsr_t` is a pointer.

Definition at line 63 of file `powsr.h`.

5.12.3 Function Documentation

5.12.3.1 powsr_free()

```
bool powsr_free (
    powsr xn )
```

Frees all the memory used by the buffer `xn`, assuming the struct has been allocated with `malloc()`, for example with `powsr_new()`.

Parameters

<code>xn</code>	the powers buffer
-----------------	-------------------

Returns

`true` if successfull, `false` otherwise.

5.12.3.2 powsr_new()

```
powsr powsr_new (
    prec_t prec,
    deg_t size )
```

Returns a new buffer of powers of real numbers of precision `prec`, with initial storage space for `size` powers.

Warning

`prec` must be at least `prec_f` and `size` at most `MAX_DEG`.

Parameters

<code>prec</code>	the precision of the coefficients, in bits
<code>size</code>	the size of the buffer

Returns

the new buffer, `NULL` if some error occurred.

5.12.3.3 powsr_pow()

```
mpfr_ptr powsr_pow (
    powsr xn,
    deg_t pow )
```

Computes x^{pow} using repeated squares method and the cache of previously computed powers.

It caches intermediary powers of x that have been computed to accelerate later calls of this function. Also, if `pow` is larger than the size of the buffer `xn`, it is automatically increased to store the result.

Parameters

<code>xn</code>	the powers buffer
<code>pow</code>	the power to compute

Returns

the result x^{pow} , `NULL` if some error occurred.

5.12.3.4 powsr_pow_once()

```
mpfr_ptr powsr_pow_once (
    powsr xn,
    deg_t pow )
```

Computes x^{pow} using repeated squares method and the cache of previously computed powers.

It does **NOT** cache intermediary powers of x into `xn`.

Parameters

<code>xn</code>	the powers buffer
<code>pow</code>	the power to compute

Returns

the result x^{pow} , `NULL` if some error occurred.

5.12.3.5 powsr_set()

```
bool powsr_set (
    powsr xn,
    mpfr_t x )
```

Sets the real number of which the powers will be computed by the buffer `xn`.

Parameters

<code>xn</code>	the powers buffer
<code>x</code>	the complex number

Returns

`true` if successfull, `false` otherwise.

5.13 /home/runner/work/FastPolyEval/FastPolyEval/code/numbers/comp.h File Reference

Definition of MPFR complex numbers.

Data Structures

- struct `comp_struct`
Multi-precision floating point complex numbers.

Macros

- #define `comp_init(d, prec)`
Allocates memory for the digits of `d`.
- #define `comp_initz(d, prec)`
Allocates memory for the digits of `d` and sets its value to 0.
- #define `comp_clear(d)`
De-allocates the memory used by the digits of `d`.
- #define `comp_zero(d) (mpfr_zero_p((d)->x) && mpfr_zero_p((d)->y))`
Tests if `d==0`.
- #define `comp_add(d, a, b)`
Adds `a` to `b` and stores the result in `d`, all of type `comp` or `comp_ptr`.
- #define `comp_set(d, a)`
Sets `d` to `a`.
- #define `comp_setr(d, a)`
Sets `d` to the real value `a`.
- #define `comp_neg(d, a)`
Sets `d` to $-a$.
- #define `comp_addr(d, a, r)`

- Adds the complex number a to the real number b and stores the result in d .

 - `#define comp_amu(d, a, r, buf)`

Adds the complex number $a \cdot r$ to d , where d, a are complex and r is unsigned integer.
 - `#define comp_sub(d, a, b)`
- Subtracts b from a and stores the result in d , all of type `comp` or `comp_ptr`.
 - `#define comp_subr(d, a, r)`

Subtracts the real number b from the complex number a and stores the result in d .
 - `#define comp_mul(d, a, b, buf1, buf2)`

Multiplies a to b and stores the result in d , all of type `comp` or `comp_ptr`.
 - `#define comp_div(d, a, b, b1, b2, b3)`

Divides a by b and stores the result in d , all of type `comp` or `comp_ptr`.
 - `#define comp_mod(m, a) mpfr_hypot((m), (a)->x, (a)->y, MPFR_RNDN);`

Computes the modulus of a and stores the result in m , a of type `comp` and m of type `mpfr_t`.
 - `#define comp_mulr(d, a, r)`

Multiplies the complex number a to the real number r and stores the result in d .
 - `#define comp_muli(d, a, i)`

Multiplies the complex number a to the integer i and stores the result in d .
 - `#define comp_mulu(d, a, i)`

Multiplies the complex number a to the unsigned integer i and stores the result in d .
 - `#define comp_sqr(d, a, buf)`

Squares a and stores the result in d , all of type `comp` or `comp_ptr`.

Typedefs

- `typedef comp_struct comp[1]`
- Practical wrapper for `comp_struct`.
 - `typedef comp_struct * comp_ptr`

Convenience pointer to `comp_struct`.

Functions

- `real_t comp_log2 (comp z)`
- Computes the base 2 \log of $|z|$.
 - `real_t real_log2 (mpfr_t x)`

Computes the base 2 \log of $|x|$.
 - `real_t comp_s (comp z)`

Computes $s(z)$.
 - `real_t mpfr_s (mpfr_t x)`

Computes $s(x)$.

5.13.1 Detailed Description

Definition of MPFR complex numbers.

5.13.2 Macro Definition Documentation

5.13.2.1 comp_add

```
#define comp_add(
    d,
    a,
    b )
```

Value:

```
mpfr_add((d)->x, (a)->x, (b)->x, MPFR_RNDN); \
mpfr_add((d)->y, (a)->y, (b)->y, MPFR_RNDN);
```

Adds *a* to *b* and stores the result in *d*, all of type [comp](#) or [comp_ptr](#).

Definition at line 64 of file `comp.h`.

5.13.2.2 comp_addr

```
#define comp_addr(
    d,
    a,
    r )
```

Value:

```
mpfr_add((d)->x, (a)->x, (r), MPFR_RNDN); \
mpfr_set((d)->y, (a)->y, MPFR_RNDN);
```

Adds the complex number *a* to the real number *b* and stores the result in *d*.

Definition at line 80 of file `comp.h`.

5.13.2.3 comp_amu

```
#define comp_amu(
    d,
    a,
    r,
    buf )
```

Value:

```
mpfr_mul_ui((buf), (a)->x, (r), MPFR_RNDN); \
mpfr_add((d)->x, (d)->x, (buf), MPFR_RNDN); \
mpfr_mul_ui((buf), (a)->y, (r), MPFR_RNDN); \
mpfr_add((d)->y, (d)->y, (buf), MPFR_RNDN);
```

Adds the complex number *a***r* to *d*, where *d*,*a* are complex and *r* is unsigned integer.

Definition at line 84 of file `comp.h`.

5.13.2.4 comp_clear

```
#define comp_clear(  
    d )
```

Value:

```
mpfr_clear((d)->x); \  
mpfr_clear((d)->y);
```

De-allocates the memory used by the digits of *d*.

Definition at line 57 of file comp.h.

5.13.2.5 comp_div

```
#define comp_div(  
    d,  
    a,  
    b,  
    b1,  
    b2,  
    b3 )
```

Value:

```
mpfr_sqr(b3), (b)->x, MPFR_RNDN); \  
mpfr_sqr(b2), (b)->y, MPFR_RNDN); \  
mpfr_add(b3), (b3), (b2), MPFR_RNDN); \  
mpfr_mul(b1), (a)->x, (b)->x, MPFR_RNDN); \  
mpfr_mul(b2), (a)->y, (b)->y, MPFR_RNDN); \  
mpfr_add(b1), (b1), (b2), MPFR_RNDN); \  
mpfr_mul(b2), (a)->x, (b)->y, MPFR_RNDN); \  
mpfr_mul((d)->y, (a)->y, (b)->x, MPFR_RNDN); \  
mpfr_sub((d)->y, (d)->y, (b2), MPFR_RNDN); \  
mpfr_div((d)->y, (d)->y, (b3), MPFR_RNDN); \  
mpfr_div((d)->x, (b1), (b3), MPFR_RNDN);
```

Divides *a* by *b* and stores the result in *d*, all of type [comp](#) or [comp_ptr](#).

Definition at line 107 of file comp.h.

5.13.2.6 comp_init

```
#define comp_init(  
    d,  
    prec )
```

Value:

```
mpfr_init2((d)->x, prec); \  
mpfr_init2((d)->y, prec);
```

Allocates memory for the digits of *d*.

Definition at line 47 of file comp.h.

5.13.2.7 comp_initz

```
#define comp_initz(
    d,
    prec )
```

Value:

```
mpfr_init2((d)->x, prec); \
mpfr_init2((d)->y, prec); \
mpfr_set_zero((d)->x, 1); \
mpfr_set_zero((d)->y, 1);
```

Allocates memory for the digits of *d* and sets its value to 0.

Definition at line 51 of file comp.h.

5.13.2.8 comp_mul

```
#define comp_mul(
    d,
    a,
    b,
    buf1,
    buf2 )
```

Value:

```
mpfr_mul(buf1, (a)->x, (b)->x, MPFR_RNDN); \
mpfr_mul(buf2, (a)->y, (b)->y, MPFR_RNDN); \
mpfr_sub(buf1, (buf1), (buf2), MPFR_RNDN); \
mpfr_mul(buf2, (a)->x, (b)->y, MPFR_RNDN); \
mpfr_mul((d)->y, (a)->y, (b)->x, MPFR_RNDN); \
mpfr_add((d)->y, (d)->y, (buf2), MPFR_RNDN); \
mpfr_set((d)->x, (buf1), MPFR_RNDN);
```

Multiplies *a* to *b* and stores the result in *d*, all of type [comp](#) or [comp_ptr](#).

Definition at line 98 of file comp.h.

5.13.2.9 comp_muli

```
#define comp_muli(
    d,
    a,
    i )
```

Value:

```
mpfr_mul_si((d)->x, (a)->x, (i), MPFR_RNDN); \
mpfr_mul_si((d)->y, (a)->y, (i), MPFR_RNDN);
```

Multiplies the complex number *a* to the integer *i* and stores the result in *d*.

Definition at line 127 of file comp.h.

5.13.2.10 comp_mulr

```
#define comp_mulr(  
    d,  
    a,  
    r )
```

Value:

```
mpfr_mul((d)->x, (a)->x, (r), MPFR_RNDN); \  
mpfr_mul((d)->y, (a)->y, (r), MPFR_RNDN);
```

Multiplies the complex number *a* to the real number *r* and stores the result in *d*.

Definition at line 123 of file comp.h.

5.13.2.11 comp_mulu

```
#define comp_mulu(  
    d,  
    a,  
    i )
```

Value:

```
mpfr_mul_ui((d)->x, (a)->x, (i), MPFR_RNDN); \  
mpfr_mul_ui((d)->y, (a)->y, (i), MPFR_RNDN);
```

Multiplies the complex number *a* to the unsigned integer *i* and stores the result in *d*.

Definition at line 131 of file comp.h.

5.13.2.12 comp_neg

```
#define comp_neg(  
    d,  
    a )
```

Value:

```
mpfr_neg((d)->x, (a)->x, MPFR_RNDN); \  
mpfr_neg((d)->y, (a)->y, MPFR_RNDN);
```

Sets *d* to $-a$.

Definition at line 76 of file comp.h.

5.13.2.13 comp_set

```
#define comp_set(
    d,
    a )
```

Value:

```
mpfr_set((d)->x, (a)->x, MPFR_RNDN); \
mpfr_set((d)->y, (a)->y, MPFR_RNDN);
```

Sets `d` to `a`.

Definition at line 68 of file `comp.h`.

5.13.2.14 comp_setr

```
#define comp_setr(
    d,
    a )
```

Value:

```
mpfr_set((d)->x, (a), MPFR_RNDN); \
mpfr_set_zero((d)->y, 1);
```

Sets `d` to the real value `a`.

Definition at line 72 of file `comp.h`.

5.13.2.15 comp_sqr

```
#define comp_sqr(
    d,
    a,
    buf )
```

Value:

```
mpfr_sqr(buf, (a)->y, MPFR_RNDN); \
mpfr_mul((d)->y, (a)->x, (a)->y, MPFR_RNDN); \
mpfr_mul_2si((d)->y, (d)->y, 1, MPFR_RNDN); \
mpfr_sqr((d)->x, (a)->x, MPFR_RNDN); \
mpfr_sub((d)->x, (d)->x, (buf), MPFR_RNDN);
```

Squares `a` and stores the result in `d`, all of type `comp` or `comp_ptr`.

Definition at line 135 of file `comp.h`.

5.13.2.16 comp_sub

```
#define comp_sub(
    d,
    a,
    b )
```

Value:

```
mpfr_sub((d)->x, (a)->x, (b)->x, MPFR_RNDN); \
mpfr_sub((d)->y, (a)->y, (b)->y, MPFR_RNDN);
```

Subtracts `b` from `a` and stores the result in `d`, all of type `comp` or `comp_ptr`.

Definition at line 90 of file `comp.h`.

5.13.2.17 comp_subr

```
#define comp_subr(
    d,
    a,
    r )
```

Value:

```
mpfr_sub((d)->x, (a)->x, (r), MPFR_RNDN); \
mpfr_set((d)->y, (a)->y, MPFR_RNDN);
```

Subtracts the real number `b` from the complex number `a` and stores the result in `d`.

Definition at line 94 of file `comp.h`.

5.13.3 Typedef Documentation

5.13.3.1 comp

```
typedef comp_struct comp[1]
```

Practical wrapper for `comp_struct`.

To avoid the constant use `*` and `&` the type `compf` is a pointer.

Definition at line 39 of file `comp.h`.

5.13.4 Function Documentation

5.13.4.1 comp_log2()

```
real_t comp_log2 (
    comp z )
```

Computes the base 2 \log of $|z|$.

Parameters

z	the complex number
-----	--------------------

Returns
 $\log_2(|z|)$
5.13.4.2 comp_s()

```
real_t comp_s (
    comp z )
```

Computes $s(z)$.

See also

[1]

Parameters

z	the complex number
-----	--------------------

Returns
 $[\log_2(|z|)]+1$
5.13.4.3 mpfr_s()

```
real_t mpfr_s (
    mpfr_t x )
```

Computes $s(x)$.

See also

[1]

Parameters

x	the real number
-----	-----------------

Returns

$$\lceil \log_2(|z|) \rceil + 1$$

5.13.4.4 real_log2()

```
real_t real_log2 (
    mpfr_t x )
```

Computes the base 2 \log of $|x|$.

Parameters

x	the real number
---	-----------------

Returns

$$\log_2(|x|)$$

5.14 /home/runner/work/FastPolyEval/FastPolyEval/code/numbers/compf.h File Reference

Definition of machine complex numbers.

Data Structures

- struct [compf_struct](#)
Machine complex numbers.

Macros

- #define [compf_set](#)(d, a)
Sets d to a .
- #define [compf_setr](#)(d, a)
Sets d to the real value a .
- #define [compf_neg](#)(d, a)
Sets d to $-a$.
- #define [compf_add](#)(d, a, b)
Adds a to b and stores the result in d , all of type [compf](#) or [compf_ptr](#).
- #define [compf_addr](#)(d, a, r)
Adds the complex number a to the real number b and stores the result in d .
- #define [compf_sub](#)(d, a, b)
Subtracts b from a and stores the result in d , all of type [compf](#) or [compf_ptr](#).
- #define [compf_subr](#)(d, a, r)

- Subtracts the real number b from the complex number a and stores the result in d .
- `#define compf_mul(d, a, b)`
- `#define compf_mulr(d, a, r)`
 - Multiplies the complex number a to the real number r and stores the result in d .
- `#define compf_amr(d, a, r)`
 - Adds the complex number $a*r$ to d , where d, a are complex and r is real.
- `#define compf_sqr(d, a)`
 - Squares a and stores the result in d , all of type `compf` or `compf_ptr`.
- `#define compf_div(d, a, b)`
 - Divides a by b and stores the result in d , all of type `compf` or `compf_ptr`.
- `#define compf_mod(a) fhypot((a)->x, (a)->y)`
 - Computes the modulus of the complex number a .
- `#define compf_dist(a, b) fhypot((a)->x - (b)->x, (a)->y - (b)->y)`
 - Computes the distance between the complex numbers a and b .
- `#define compf_mod2(a) ((a)->x * (a)->x + (a)->y * (a)->y)`
- `#define compf_log2(a) (plog2(compf_mod(a)))`
 - Computes the \log_2 of the modulus of the complex number a .
- `#define coeff_log2(a) (plog2((a) < 0 ? -(a) : (a)))`
 - Computes the \log_2 of the absolute value of the real number a .
- `#define compf_s(a) (pfloor(plog2(compf_mod(a))) + 1)`
 - Computes the scale of the complex number a , see [1].
- `#define coeff_s(a) (pfloor(coeff_log2(a)) + 1)`
 - Computes the scale of the real number a , see [1].

Typedefs

- `typedef compf_struct compf[1]`
 - Practical wrapper for `compf_struct`.
- `typedef compf_struct * compf_ptr`
 - Convenience pointer to `compf_struct`.

5.14.1 Detailed Description

Definition of machine complex numbers.

5.14.2 Macro Definition Documentation

5.14.2.1 compf_add

```
#define compf_add(
    d,
    a,
    b )
```

Value:

```
(d)->x = (a)->x + (b)->x; \
(d)->y = (a)->y + (b)->y;
```

Adds a to b and stores the result in d , all of type `compf` or `compf_ptr`.

Definition at line 63 of file `compf.h`.

5.14.2.2 compf_addr

```
#define compf_addr(
    d,
    a,
    r )
```

Value:

```
(d)->x = (a)->x + (r); \
(d)->y = (a)->y;
```

Adds the complex number `a` to the real number `b` and stores the result in `d`.

Definition at line 67 of file `compf.h`.

5.14.2.3 compf_amr

```
#define compf_amr(
    d,
    a,
    r )
```

Value:

```
(d)->x += (a)->x * (r); \
(d)->y += (a)->y * (r);
```

Adds the complex number `a*r` to `d`, where `d,a` are complex and `r` is real.

Definition at line 87 of file `compf.h`.

5.14.2.4 compf_div

```
#define compf_div(
    d,
    a,
    b )
```

Value:

```
{ \
    coeff_t m2 = compf_mod2(b); \
    coeff_t px = ((a)->x * (b)->x + (a)->y * (b)->y) / m2; \
    (d)->y = ((a)->y * (b)->x - (a)->x * (b)->y) / m2; \
    (d)->x = px;\
}
```

Divides `a` by `b` and stores the result in `d`, all of type `compf` or `compf_ptr`.

Definition at line 96 of file `compf.h`.

5.14.2.5 compf_mod2

```
#define compf_mod2(  
    a ) ((a)->x * (a)->x + (a)->y * (a)->y)
```

Computes the square of the modulus of the complex number `a`.

Warning

There is a danger of overflow, better use the slower `compf_mod()` instead.

Definition at line 111 of file `compf.h`.

5.14.2.6 compf_mul

```
#define compf_mul(  
    d,  
    a,  
    b )
```

Value:

```
coeff_t px = (a)->x * (b)->x - (a)->y * (b)->y; \  
(d)->y = (a)->x * (b)->y + (a)->y * (b)->x; \  
(d)->x = px;
```

Definition at line 78 of file `compf.h`.

5.14.2.7 compf_mulr

```
#define compf_mulr(  
    d,  
    a,  
    r )
```

Value:

```
(d)->x = (a)->x * (r); \  
(d)->y = (a)->y * (r);
```

Multiplies the complex number `a` to the real number `r` and stores the result in `d`.

Definition at line 83 of file `compf.h`.

5.14.2.8 compf_neg

```
#define compf_neg(
    d,
    a )
```

Value:

```
(d)->x = -(a)->x; \
(d)->y = -(a)->y;
```

Sets d to -a.

Definition at line 59 of file compf.h.

5.14.2.9 compf_set

```
#define compf_set(
    d,
    a )
```

Value:

```
(d)->x = (a)->x; \
(d)->y = (a)->y;
```

Sets d to a.

Definition at line 51 of file compf.h.

5.14.2.10 compf_setr

```
#define compf_setr(
    d,
    a )
```

Value:

```
(d)->x = (a); \
(d)->y = 0;
```

Sets d to the real value a.

Definition at line 55 of file compf.h.

5.14.2.11 compf_sqr

```
#define compf_sqr(
    d,
    a )
```

Value:

```
coeff_t px = (a)->x * (a)->x - (a)->y * (a)->y; \
(d)->y = 2 * (a)->x * (a)->y; \
(d)->x = px;
```

Squares a and stores the result in d, all of type [compf](#) or [compf_ptr](#).

Definition at line 91 of file compf.h.

5.14.2.12 compf_sub

```
#define compf_sub(
    d,
    a,
    b )
```

Value:

```
(d)->x = (a)->x - (b)->x; \
(d)->y = (a)->y - (b)->y;
```

Subtracts `b` from `a` and stores the result in `d`, all of type `compf` or `compf_ptr`.

Definition at line 71 of file `compf.h`.

5.14.2.13 compf_subr

```
#define compf_subr(
    d,
    a,
    r )
```

Value:

```
(d)->x = (a)->x - (r); \
(d)->y = (a)->y;
```

Subtracts the real number `b` from the complex number `a` and stores the result in `d`.

Definition at line 75 of file `compf.h`.

5.14.3 Typedef Documentation

5.14.3.1 compf

```
typedef compf_struct compf[1]
```

Practical wrapper for `compf_struct`.

To avoid the constant use `*` and `&` the type `compf` is a pointer.

Example of use:

```
compf c;
polyf cx = c->x;
```

Definition at line 43 of file `compf.h`.

5.15 /home/runner/work/FastPolyEval/FastPolyEval/code/numbers/ntypes.h File Reference

Definition of basic types.

Macros

- #define **true** 1
Boolean value true.
- #define **false** 0
Boolean value false.
- #define **PI** (3.14159265358979323846264338327950288L)
 π with fp80 precision
- #define **flog2** log2
- #define **fhypot** hypot
- #define **ffloor** floor
- #define **fpow** pow
- #define **fexp** exp
- #define **fsin** sin
- #define **fcos** cos
- #define **ftan** tan
- #define **mpfr_getf** mpfr_get_d
- #define **precf** 53
- #define **PRECF_STR** "53"
- #define **TYPEFP_STR** "FP64"
- #define **TYPEF_STR** "double"
- #define **INF_M** (-HUGE_VAL)
- #define **INF_P** HUGE_VAL
- #define **FMT_COEFF** "l"
- #define **MAX_EXP** (sizeof(**coeff_t**) == 8 ? 960 : 16320)
- #define **HUGE_DEGREES**
- #define **MAX_DEG** (UINT64_MAX - 1)
- #define **FMT_DEG** PRIu64
- #define **plog2** log2
- #define **pexp2** exp2
- #define **phypot** hypot
- #define **pfloor** floor
- #define **pceil** ceil
- #define **mpfr_getp** mpfr_get_d
- #define **PEPS** 1E-200
- #define **PDEL** 1E-12
- #define **pexp** exp
- #define **pcos** cos
- #define **psin** sin
- #define **ptan** tan
- #define **pldexp** ldexp
- #define **pfrexp** frexp
- #define **FMT_REAL** "l"
- #define **HUGE_PREC**
- #define **MAX_PREC** (16000000000000000000UL)
- #define **HUGE_MP_EXP**
- #define **MAX_MP_EXP** (4000000000000000000L)
The integer number type to use for polynomial degrees and indexes.
- #define **NEWTON_CONV_BITS** 5
- #define **NEWTON_ESCAPE_BITS** 4

Typedefs

- typedef uint8_t `byte`
byte is uint8
- typedef uint16_t `word`
word is uint16
- typedef uint32_t `uint`
uint is uint32
- typedef uint64_t `ulong`
ulong is uint64
- typedef `byte` `bool`
Logic type bool can take values true or false.
- typedef double `coeff_t`
The machine number type to use for polynomial coefficients and evaluation.
- typedef `ulong` `deg_t`
The integer number type to use for polynomial degrees and indexes.
- typedef double `real_t`
The machine number type to use for polynomial analysis and preconditionning.
- typedef `ulong` `prec_t`
The integer number type to use for polynomial degrees and indexes.

Functions

- `bool` `ntypes_check` (void)
Use this function to check the compatibility of the machine with the settings above.
- `real_t` `bits_sum` (`real_t` b1, `real_t` b2)
Computes $\log_2(2^{b1}+2^{b2})$, even if b1 or b2 are outside the exponent range of `real_t`.
- `real_t` `nt_err` (`real_t` vb, `real_t` evb, `real_t` db, `real_t` edb)
Computes $\log_2(ntErr)$, where ntErr is an upper bound for the error of the Newton term.

5.15.1 Detailed Description

Definition of basic types.

5.15.2 Function Documentation

5.15.2.1 bits_sum()

```
real_t bits_sum (
    real_t b1,
    real_t b2 )
```

Computes $\log_2(2^{b1}+2^{b2})$, even if b1 or b2 are outside the exponent range of `real_t`.

Parameters

<i>b1</i>	first operand, in bits
<i>b2</i>	second operand, in bits

Returns

the bits sum of the operands, that is the base 2 \log of their base 2 exponentials

5.15.2.2 nt_err()

```
real_t nt_err (
    real_t vb,
    real_t evb,
    real_t db,
    real_t edb )
```

Computes $\log_2(\text{ntErr})$, where `ntErr` is an upper bound for the error of the Newton term.

Parameters

<i>vb</i>	$\log_2(\text{value})$
<i>evb</i>	$\log_2(\text{valErr})$
<i>db</i>	$\log_2(\text{derivative})$
<i>edb</i>	$\log_2(\text{derErr})$

Returns

absolute error of the Newton term, as a power of 2

5.16 /home/runner/work/FastPolyEval/FastPolyEval/code/poly/poly.h File Reference

Definition of complex polynomials with arbitrary precision coefficients.

Data Structures

- struct `poly_struct`
Polynomial with multi-precision floating point complex coefficients.

Typedefs

- typedef `poly_struct poly_t[1]`
Practical wrapper for `poly_struct`.
- typedef `poly_struct * poly`
Convenience pointer to `poly_struct`.

Functions

- `poly poly_new (deg_t degree, prec_t prec)`
Returns a new complex polynomial of given *degree*, with coefficients of precision *prec*.
- `poly poly_from_roots (comp_ptr roots, deg_t degree, prec_t prec)`
Returns a new complex polynomial given the list of its *roots*, with coefficients of precision *prec*.
- `bool poly_free (poly P)`
Frees all the memory used by the polynomial *P*, assuming the struct has been allocated with `malloc()`, for example with `poly_new()`.
- `bool poly_set (poly P, comp coeff, deg_t ind)`
Sets the coefficient of the polynomial *P* corresponding to the power *ind* to *coeff*.
- `bool poly_eval (comp res, poly P, comp z)`
Evaluates $P(z)$ using Horner's method.
- `bool poly_eval_r (comp res, poly P, mpfr_t x)`
Evaluates $P(x)$ using Horner's method.
- `poly poly_derivative (poly P)`
Computes the derivative of *P*.
- `poly poly_sum (poly P, poly Q)`
Computes $P+Q$.
- `poly poly_diff (poly P, poly Q)`
Computes $P-Q$.
- `poly poly_prod (poly P, poly Q)`
Computes $P*Q$.
- `poly poly_sqr (poly P)`
Computes the square of *P*.

5.16.1 Detailed Description

Definition of complex polynomials with arbitrary precision coefficients.

5.16.2 Typedef Documentation

5.16.2.1 poly_t

```
typedef poly_struct poly_t[1]
```

Practical wrapper for `poly_struct`.

To avoid the constant use `*` and `&` the type `poly_t` is a pointer.

Definition at line 43 of file `poly.h`.

5.16.3 Function Documentation

5.16.3.1 poly_derivative()

```
poly poly_derivative (
    poly P )
```

Computes the derivative of *P*.

Parameters

P	the polynomial
-----	----------------

Returns

the resulted polynomial, `NULL` if some error occurred.

5.16.3.2 `poly_diff()`

```
poly poly_diff (
    poly P,
    poly Q )
```

Computes $P - Q$.

Parameters

P	a polynomial
Q	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.16.3.3 `poly_eval()`

```
bool poly_eval (
    comp res,
    poly P,
    comp z )
```

Evaluates $P(z)$ using Horner's method.

Parameters

res	the result
P	the polynomial
z	the argument

Returns

`true` if successfull, `false` otherwise.

5.16.3.4 poly_eval_r()

```
bool poly_eval_r (
    comp res,
    poly P,
    mpfr_t x )
```

Evaluates $P(x)$ using Horner's method.

Parameters

<i>res</i>	the result
<i>P</i>	the polynomial
<i>x</i>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.16.3.5 poly_free()

```
bool poly_free (
    poly P )
```

Frees all the memory used by the polynomial P , assuming the struct has been allocated with `malloc()`, for example with `poly_new()`.

Parameters

<i>P</i>	the polynomial
----------	----------------

Returns

`true` if successfull, `false` otherwise.

5.16.3.6 poly_from_roots()

```
poly poly_from_roots (
    comp_ptr roots,
    deg_t degree,
    prec_t prec )
```

Returns a new complex polynomial given the list of its `roots`, with coefficients of precision `prec`.

Warning

`prec` must be at least `precf`

Parameters

<i>roots</i>	the roots of the polynomial
<i>degree</i>	the degree of the polynomial
<i>prec</i>	the precision of the coefficients, in bits

Returns

the new polynomial, `NULL` if some error occurred.

5.16.3.7 `poly_new()`

```
poly poly_new (
    deg_t degree,
    prec_t prec )
```

Returns a new complex polynomial of given `degree`, with coefficients of precision `prec`.

Warning

`prec` must be at least `precf`

Parameters

<i>degree</i>	the degree of the polynomial
<i>prec</i>	the precision of the coefficients, in bits

Returns

the new polynomial, `NULL` if the degree is larger than `MAX_DEG`.

5.16.3.8 `poly_prod()`

```
poly poly_prod (
    poly P,
    poly Q )
```

Computes $P \cdot Q$.

Parameters

<i>P</i>	a polynomial
<i>Q</i>	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.16.3.9 poly_set()

```
bool poly_set (
    poly P,
    comp coeff,
    deg_t ind )
```

Sets the coefficient of the polynomial `P` corresponding to the power `ind` to `coeff`.

Parameters

<i>P</i>	the polynomial
<i>coeff</i>	the coefficient
<i>ind</i>	the index

Returns

`true` if successfull, `false` otherwise.

5.16.3.10 poly_sqr()

```
poly poly_sqr (
    poly P )
```

Computes the square of `P`.

Parameters

<i>P</i>	the polynomial
----------	----------------

Returns

the resulted polynomial, `NULL` if some error occurred.

5.16.3.11 poly_sum()

```
poly poly_sum (
    poly P,
    poly Q )
```

Computes $P+Q$.

Parameters

P	a polynomial
Q	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.17 /home/runner/work/FastPolyEval/FastPolyEval/code/poly/polyf.h File Reference

Definition of complex polynomials with machine floating point coefficients.

Data Structures

- struct `polyf_struct`
Polynomial with machine floating point complex coefficients.

Typedefs

- typedef `polyf_struct polyf_t[1]`
Practical wrapper for `polyf_struct`.
- typedef `polyf_struct * polyf`
Convenience pointer to `polyf_struct`.

Functions

- `polyf polyf_new (deg_t degree)`
Returns a new complex polynomial of given `degree`, with machine floating point coefficients.
- `polyf polyf_from_roots (compf_ptr roots, deg_t degree)`
Returns a new complex polynomial given the list of its `roots`, with machine floating point coefficients.
- `bool polyf_free (polyf P)`
Frees all the memory used by the polynomial `P`, assuming the struct has been allocated with `malloc()`, for example with `polyr_new()`.
- `bool polyf_set (polyf P, compf coeff, deg_t ind)`
Sets the coefficient of the polynomial `P` corresponding to the power `ind` to `coeff`.
- `bool polyf_eval (compf res, polyf P, compf z)`
Evaluates $P(z)$ using Horner's method.
- `bool polyf_eval_r (compf res, polyf P, coeff_t x)`
Evaluates $P(x)$ using Horner's method.
- `polyf polyf_derivative (polyf P)`
Computes the derivative of `P`.
- `polyf polyf_sum (polyf P, polyf Q)`
Computes $P+Q$.
- `polyf polyf_diff (polyf P, polyf Q)`
Computes $P-Q$.
- `polyf polyf_prod (polyf P, polyf Q)`
*Computes $P*Q$.*
- `polyf polyf_sqr (polyf P)`
Computes the square of `P`.

5.17.1 Detailed Description

Definition of complex polynomials with machine floating point coefficients.

5.17.2 Typedef Documentation

5.17.2.1 polyf_t

```
typedef polyf_struct polyf_t[1]
```

Practical wrapper for `polyf_struct`.

To avoid the constant use `*` and `&` the type `polyf_t` is a pointer.

Definition at line 39 of file `polyf.h`.

5.17.3 Function Documentation

5.17.3.1 polyf_derivative()

```
polyf polyf_derivative (  
    polyf P )
```

Computes the derivative of P .

Parameters

P	the polynomial
-----	----------------

Returns

the resulted polynomial, `NULL` if some error occurred.

5.17.3.2 polyf_diff()

```
polyf polyf_diff (  
    polyf P,  
    polyf Q )
```

Computes $P-Q$.

Parameters

P	a polynomial
Q	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.17.3.3 `polyf_eval()`

```
bool polyf_eval (
    compf res,
    polyf P,
    compf z )
```

Evaluates $P(z)$ using Horner's method.

Parameters

<i>res</i>	the result
P	the polynomial
z	the argument

Returns

`true` if successfull, `false` otherwise.

5.17.3.4 `polyf_eval_r()`

```
bool polyf_eval_r (
    compf res,
    polyf P,
    coeff_t x )
```

Evaluates $P(x)$ using Horner's method.

Parameters

<i>res</i>	the result
P	the polynomial
x	the real argument

Returns

`true` if successfull, `false` otherwise.

5.17.3.5 polyf_free()

```
bool polyf_free (
    polyf P )
```

Frees all the memory used by the polynomial `P`, assuming the struct has been allocated with `malloc()`, for example with `polyr_new()`.

Parameters

<i>P</i>	the polynomial
----------	----------------

Returns

`true` if successfull, `false` otherwise.

5.17.3.6 polyf_from_roots()

```
polyf polyf_from_roots (
    compf_ptr roots,
    deg_t degree )
```

Returns a new complex polynomial given the list of its `roots`, with machine floating point coefficients.

Parameters

<i>roots</i>	the roots of the polynomial
<i>degree</i>	the degree of the polynomial

Returns

the new polynomial, `NULL` if some error occurred.

5.17.3.7 polyf_new()

```
polyf polyf_new (
    deg_t degree )
```

Returns a new complex polynomial of given `degree`, with machine floating point coefficients.

Parameters

<i>degree</i>	the degree of the polynomial
---------------	------------------------------

Returns

the new polynomial, `NULL` if the degree is larger than `MAX_DEG`.

5.17.3.8 `polyf_prod()`

```
polyf polyf_prod (
    polyf P,
    polyf Q )
```

Computes $P*Q$.

Parameters

<i>P</i>	a polynomial
<i>Q</i>	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.17.3.9 `polyf_set()`

```
bool polyf_set (
    polyf P,
    compf coeff,
    deg_t ind )
```

Sets the coefficient of the polynomial `P` corresponding to the power `ind` to `coeff`.

Parameters

<i>P</i>	the polynomial
<i>coeff</i>	the coefficient
<i>ind</i>	the index

Returns

`true` if successfull, `false` otherwise.

5.17.3.10 polyf_sqr()

```
polyf polyf_sqr (
    polyf P )
```

Computes the square of P .

Parameters

P	the polynomial
-----	----------------

Returns

the resulted polynomial, `NULL` if some error occurred.

5.17.3.11 polyf_sum()

```
polyf polyf_sum (
    polyf P,
    polyf Q )
```

Computes $P+Q$.

Parameters

P	a polynomial
Q	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.18 /home/runner/work/FastPolyEval/FastPolyEval/code/poly/polyfr.h File Reference

Definition of real polynomials with machine floating point coefficients.

Data Structures

- struct `polyfr_struct`

Polynomial with machine floating point real coefficients.

Typedefs

- typedef `polyfr_struct polyfr_t[1]`
Practical wrapper for `polyfr_struct`.
- typedef `polyfr_struct * polyfr`
Convenience pointer to `polyfr_struct`.

Functions

- `polyfr polyfr_new (deg_t degree)`
Returns a new real polynomial of given `degree`, with machine floating point coefficients.
- `bool polyfr_free (polyfr P)`
Frees all the memory used by the polynomial `P`, assuming the struct has been allocated with `malloc()`, for example with `polyfr_new()`.
- `bool polyfr_set (polyfr P, coeff_t coeff, deg_t ind)`
Sets the coefficient corresponding to the power `ind` to `coeff`.
- `coeff_t polyfr_get (polyfr P, deg_t ind)`
Returns the coefficient corresponding to the power `ind`.
- `bool polyfr_eval_c (compf res, polyfr P, compf z)`
Evaluates $P(z)$ using Horner's method.
- `coeff_t polyfr_eval (polyfr P, coeff_t x)`
Evaluates $P(x)$ using Horner's method.
- `polyf polyfr_comp (polyfr P)`
Return a complex version of the real polynomial `P`.
- `polyfr polyfr_derivative (polyfr P)`
Computes the derivative of `P`.
- `polyfr polyfr_sum (polyfr P, polyfr Q)`
Computes $P+Q$.
- `polyfr polyfr_diff (polyfr P, polyfr Q)`
Computes $P-Q$.
- `polyfr polyfr_prod (polyfr P, polyfr Q)`
*Computes $P*Q$.*
- `polyfr polyfr_sqr (polyfr P)`
Computes the square of `P`.
- `polyfr polyf_hyp (int n)`
Computes the n -th hyperbolic polynomial, the n -th image of 0 under the iteration of $z \rightarrow z^2 + c$. It is a polynomial of degree 2^{n-1} in c .
- `polyfr polyf_cheb (int n)`
Computes the Chebyshev polynomial of degree n .
- `polyfr polyf_leg (int n)`
Computes the Legendre polynomial of degree n .
- `polyfr polyf_her (int n)`
Computes the Hermite polynomial of degree n .
- `polyfr polyf_lag (int n)`
Computes the Laguerre polynomial of degree n .

5.18.1 Detailed Description

Definition of real polynomials with machine floating point coefficients.

5.18.2 Typedef Documentation

5.18.2.1 polyfr_t

```
typedef polyfr_struct polyfr_t[1]
```

Practical wrapper for `polyfr_struct`.

To avoid the constant use `*` and `&` the type `polyfr_t` is a pointer.

Definition at line 40 of file `polyfr.h`.

5.18.3 Function Documentation

5.18.3.1 polyf_cheb()

```
polyfr polyf_cheb (  
    int n )
```

Computes the Chebyshev polynomial of degree `n`.

Parameters

<code>n</code>	the degree of the polynomial
----------------	------------------------------

Returns

the Chebyshev polynomial, `NULL` if some error occurred.

5.18.3.2 polyf_her()

```
polyfr polyf_her (  
    int n )
```

Computes the Hermite polynomial of degree `n`.

Parameters

<code>n</code>	the degree of the polynomial
----------------	------------------------------

Returns

the Hermite polynomial, `NULL` if some error occurred.

5.18.3.3 polyf_hyp()

```
polyfr polyf_hyp (  
    int n )
```

Computes the n -th hyperbolic polynomial, the n -th image of 0 under the iteration of $z \rightarrow z^2 + c$. It is a polynomial of degree 2^{n-1} in c .

Parameters

n	the order of the polynomial
-----	-----------------------------

Returns

the hyperbolic polynomial, `NULL` if some error occurred.

5.18.3.4 polyf_lag()

```
polyfr polyf_lag (  
    int n )
```

Computes the Laguerre polynomial of degree n .

Parameters

n	the degree of the polynomial
-----	------------------------------

Returns

the Laguerre polynomial, `NULL` if some error occurred.

5.18.3.5 polyf_leg()

```
polyfr polyf_leg (  
    int n )
```

Computes the Legendre polynomial of degree n .

Parameters

n	the degree of the polynomial
-----	------------------------------

Returns

the Legendre polynomial, `NULL` if some error occurred.

5.18.3.6 polyfr_comp()

```
polyf polyfr_comp (  
    polyfr P )
```

Return a complex version of the real polynomial P .

Parameters

P	a polynomial
-----	--------------

Returns

the resulted polynomial, `NULL` if some error occurred.

5.18.3.7 polyfr_derivative()

```
polyfr polyfr_derivative (  
    polyfr P )
```

Computes the derivative of P .

Parameters

P	the polynomial
-----	----------------

Returns

the resulted polynomial, `NULL` if some error occurred.

5.18.3.8 polyfr_diff()

```
polyfr polyfr_diff (  
    polyfr P,  
    polyfr Q )
```

Computes $P-Q$.

Parameters

P	a polynomial
Q	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.18.3.9 `polyfr_eval()`

```
coeff_t polyfr_eval (
    polyfr P,
    coeff_t x )
```

Evaluates $P(x)$ using Horner's method.

Parameters

P	the polynomial
x	the argument

Returns

the result, `NaN` if some error occurred.

5.18.3.10 `polyfr_eval_c()`

```
bool polyfr_eval_c (
    compf res,
    polyfr P,
    compf z )
```

Evaluates $P(z)$ using Horner's method.

Parameters

res	the result
P	the polynomial
z	the argument

Returns

`true` if successfull, `false` otherwise.

5.18.3.11 polyfr_free()

```
bool polyfr_free (
    polyfr P )
```

Frees all the memory used by the polynomial `P`, assuming the struct has been allocated with `malloc()`, for example with `polyfr_new()`.

Parameters

<i>P</i>	the polynomial
----------	----------------

Returns

`true` if successfull, `false` otherwise.

5.18.3.12 polyfr_get()

```
coeff_t polyfr_get (
    polyfr P,
    deg_t ind )
```

Returns the coefficient corresponding to the power `ind`.

Parameters

<i>P</i>	the polynomial
<i>ind</i>	the index

Returns

the coefficient corresponding to the power `ind`.

5.18.3.13 polyfr_new()

```
polyfr polyfr_new (
    deg_t degree )
```

Returns a new real polynomial of given `degree`, with machine floating point coefficients.

Parameters

<i>degree</i>	the degree of the polynomial
---------------	------------------------------

Returns

the new polynomial, `NULL` if the degree is larger than `MAX_DEG`.

5.18.3.14 polyfr_prod()

```
polyfr polyfr_prod (
    polyfr P,
    polyfr Q )
```

Computes $P*Q$.

Parameters

<i>P</i>	a polynomial
<i>Q</i>	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.18.3.15 polyfr_set()

```
bool polyfr_set (
    polyfr P,
    coeff_t coeff,
    deg_t ind )
```

Sets the coefficient corresponding to the power `ind` to `coeff`.

Parameters

<i>P</i>	the polynomial
<i>coeff</i>	the coefficient
<i>ind</i>	the index

Returns

`true` if successfull, `false` otherwise.

5.18.3.16 polyfr_sqr()

```
polyfr polyfr_sqr (
    polyfr P )
```

Computes the square of P .

Parameters

P	the polynomial
-----	----------------

Returns

the resulted polynomial, `NULL` if some error occurred.

5.18.3.17 polyfr_sum()

```
polyfr polyfr_sum (
    polyfr P,
    polyfr Q )
```

Computes $P+Q$.

Parameters

P	a polynomial
Q	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.19 /home/runner/work/FastPolyEval/FastPolyEval/code/poly/polyr.h File Reference

Definition of real polynomials with arbitrary precision coefficients.

Data Structures

- struct `polyr_struct`

Polynomial with multi-precision floating point complex coefficients.

Typedefs

- typedef `polyr_struct` `polyr_t`[1]
Practical wrapper for `polyr_struct`.
- typedef `polyr_struct` * `polyr`
Convenience pointer to `polyr_struct`.

Functions

- `polyr` `polyr_new` (`deg_t` degree, `prec_t` prec)
Returns a new real polynomial of given `degree`, with coefficients of precision `prec`.
- `bool` `polyr_free` (`polyr` P)
Frees all the memory used by the polynomial `P`, assuming the struct has been allocated with `malloc()`, for example with `polyr_new()`.
- `bool` `polyr_set` (`polyr` P, `mpfr_t` coeff, `deg_t` ind)
Sets the coefficient of the polynomial `P` corresponding to the power `ind` to `coeff`.
- `bool` `polyr_seti` (`polyr` P, long coeff, `deg_t` ind)
Sets the coefficient of the polynomial `P` corresponding to the power `ind` to `coeff`.
- `bool` `polyr_eval_c` (`comp` res, `polyr` P, `comp` z)
Evaluates $P(z)$ using Horner's method.
- `bool` `polyr_eval` (`mpfr_t` res, `polyr` P, `mpfr_t` x)
Evaluates $P(x)$ using Horner's method.
- `polyr` `polyr_derivative` (`polyr` P)
Computes the derivative of `P`.
- `polyr` `polyr_sum` (`polyr` P, `polyr` Q)
Computes $P+Q$.
- `polyr` `polyr_diff` (`polyr` P, `polyr` Q)
Computes $P-Q$.
- `polyr` `polyr_prod` (`polyr` P, `polyr` Q)
*Computes $P*Q$.*
- `polyr` `polyr_sqr` (`polyr` P)
Computes the square of `P`.
- `polyr` `poly_hyp` (int n, `prec_t` prec)
Computes the n -th hyperbolic polynomial, the n -th image of 0 under the iteration of $z \rightarrow z^2 + c$. It is a polynomial of degree 2^{n-1} in `c`.
- `polyr` `poly_cheb` (int n, `prec_t` prec)
Computes the Chebyshev polynomial of degree n .
- `polyr` `poly_leg` (int n, `prec_t` prec)
Computes the Legendre polynomial of degree n .
- `polyr` `poly_her` (int n, `prec_t` prec)
Computes the Hermite polynomial of degree n .
- `polyr` `poly_lag` (int n, `prec_t` prec)
Computes the Laguerre polynomial of degree n .

5.19.1 Detailed Description

Definition of real polynomials with arbitrary precision coefficients.

5.19.2 Typedef Documentation

5.19.2.1 `polyr_t`

```
typedef polyr_struct polyr_t[1]
```

Practical wrapper for `polyr_struct`.

To avoid the constant use `*` and `&` the type `polyr_t` is a pointer.

Definition at line 43 of file `polyr.h`.

5.19.3 Function Documentation

5.19.3.1 `poly_cheb()`

```
polyr poly_cheb (  
    int n,  
    prec_t prec )
```

Computes the Chebyshev polynomial of degree *n*.

Parameters

<i>n</i>	the degree of the polynomial
<i>prec</i>	the precision of the coefficients

Returns

the Chebyshev polynomial, `NULL` if some error occurred.

5.19.3.2 `poly_her()`

```
polyr poly_her (  
    int n,  
    prec_t prec )
```

Computes the Hermite polynomial of degree *n*.

Parameters

<i>n</i>	the degree of the polynomial
<i>prec</i>	the precision of the coefficients

Returns

the Hermite polynomial, `NULL` if some error occurred.

5.19.3.3 poly_hyp()

```
polyr poly_hyp (
    int n,
    prec_t prec )
```

Computes the *n*-th hyperbolic polynomial, the *n*-th image of 0 under the iteration of $z \rightarrow z^2 + c$. It is a polynomial of degree 2^{n-1} in *c*.

Parameters

<i>n</i>	the order of the polynomial
<i>prec</i>	the precision of the coefficients

Returns

the hyperbolic polynomial, `NULL` if some error occurred.

5.19.3.4 poly_lag()

```
polyr poly_lag (
    int n,
    prec_t prec )
```

Computes the Laguerre polynomial of degree *n*.

Parameters

<i>n</i>	the degree of the polynomial
<i>prec</i>	the precision of the coefficients

Returns

the Laguerre polynomial, `NULL` if some error occurred.

5.19.3.5 poly_leg()

```
polyr poly_leg (
    int n,
    prec_t prec )
```

Computes the Legendre polynomial of degree n .

Parameters

n	the degree of the polynomial
$prec$	the precision of the coefficients

Returns

the Legendre polynomial, `NULL` if some error occurred.

5.19.3.6 polyr_derivative()

```
polyr polyr_derivative (
    polyr P )
```

Computes the derivative of P .

Parameters

P	the polynomial
-----	----------------

Returns

the resulted polynomial, `NULL` if some error occurred.

5.19.3.7 polyr_diff()

```
polyr polyr_diff (
    polyr P,
    polyr Q )
```

Computes $P - Q$.

Parameters

P	a polynomial
Q	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.19.3.8 polyr_eval()

```
bool polyr_eval (
    mpfr_t res,
    polyr P,
    mpfr_t x )
```

Evaluates $P(x)$ using Horner's method.

Parameters

<i>res</i>	the result
<i>P</i>	the polynomial
<i>x</i>	the real argument

Returns

`true` if successfull, `false` otherwise.

5.19.3.9 polyr_eval_c()

```
bool polyr_eval_c (
    comp res,
    polyr P,
    comp z )
```

Evaluates $P(z)$ using Horner's method.

Parameters

<i>res</i>	the result
<i>P</i>	the polynomial
<i>z</i>	the argument

Returns

`true` if successfull, `false` otherwise.

5.19.3.10 `polyr_free()`

```
bool polyr_free (
    polyr P )
```

Frees all the memory used by the polynomial `P`, assuming the struct has been allocated with `malloc()`, for example with `polyr_new()`.

Parameters

<i>P</i>	the polynomial
----------	----------------

Returns

`true` if successfull, `false` otherwise.

5.19.3.11 `polyr_new()`

```
polyr polyr_new (
    deg_t degree,
    prec_t prec )
```

Returns a new real polynomial of given `degree`, with coefficients of precision `prec`.

Warning

`prec` must be at least `prec_f` and the degree at most `MAX_DEG`.

Parameters

<i>degree</i>	the degree of the polynomial
<i>prec</i>	the precision of the coefficients, in bits

Returns

the new buffer, `NULL` if some error occurred.

5.19.3.12 `polyr_prod()`

```
polyr polyr_prod (
    polyr P,
    polyr Q )
```

Computes $P \cdot Q$.

Parameters

P	a polynomial
Q	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.19.3.13 `polyr_set()`

```
bool polyr_set (
    polyr P,
    mpfr_t coeff,
    deg_t ind )
```

Sets the coefficient of the polynomial P corresponding to the power `ind` to `coeff`.

Parameters

P	the polynomial
<code>coeff</code>	the coefficient
<code>ind</code>	the index

Returns

`true` if successfull, `false` otherwise.

5.19.3.14 `polyr_seti()`

```
bool polyr_seti (
    polyr P,
    long coeff,
    deg_t ind )
```

Sets the coefficient of the polynomial P corresponding to the power `ind` to `coeff`.

Parameters

P	the polynomial
<code>coeff</code>	the coefficient
<code>ind</code>	the index

Returns

`true` if successfull, `false` otherwise.

5.19.3.15 polyr_sqr()

```
polyr polyr_sqr (
    polyr P )
```

Computes the square of P .

Parameters

P	the polynomial
-----	----------------

Returns

the resulted polynomial, `NULL` if some error occurred.

5.19.3.16 polyr_sum()

```
polyr polyr_sum (
    polyr P,
    polyr Q )
```

Computes $P+Q$.

Parameters

P	a polynomial
Q	another polynomial

Returns

the resulted polynomial, `NULL` if some error occurred.

5.20 /home/runner/work/FastPolyEval/FastPolyEval/code/tools/array.h File Reference

Definition of a variable length array of arbitrary precision complex numbers that are based on `mpfr`.

Data Structures

- struct `array_struct`

A variable length array of machine floating point complex numbers.

Macros

- `#define ARRAY_MIN_SIZE 100`
- `#define ARRAY_SIZE_INCREASE 1.25`

Typedefs

- `typedef array_struct array_t[1]`
Practical wrapper for `array_struct`.
- `typedef array_struct * array`
Convenience pointer to `array_struct`.

Functions

- `array array_new (ulong size)`
Returns a new empty array with memory size at least `size`.
- `array array_new_polyr (polyr P, prec_t prec)`
Returns the array of coefficients of the real polynomial `P`.
- `array array_new_poly (poly P, prec_t prec)`
Returns the array of coefficients of the polynomial `P`.
- `bool array_free (array l)`
Frees all the memory used by the array `l`, assuming the struct has been allocated with `malloc()`, for example with `array_new()`.
- `bool array_add (array l, comp z, prec_t prec)`
Adds the complex number `z` at the end of the array. The size of the array is automatically increased, if needed.
- `comp_ptr array_get (array l, ulong pos)`
Returns the pointer of the element on position `pos` in the array `l`.
- `bool array_is_real (array l)`
Checks if all complex numbers in the array `l` are real.
- `long array_first_inf (array l)`
Checks if there are infinite values in the list `l`.
- `long array_first_nan (array l)`
Checks if all complex numbers in the list `l` are well defined.
- `polyr array_polyr (array l, prec_t prec)`
Returns the real polynomial with the coefficients the real parts of the complex numbers in the array `l`.
- `poly array_poly (array l, prec_t prec)`
Returns the polynomial with the coefficients in the array `l`.
- `bool array_write (array l, char *fileName, int digits, bool verbose)`
Writes the array `l` to a CSV file with the given path `fileName`.
- `bool array_append (array l, char *fileName, int digits, bool verbose)`
Writes the array `l` to the end of a CSV file with the given path `fileName`.
- `array array_read (char *fileName, prec_t prec, bool verbose)`
Reads a array of complex numbers of precision `prec` from the CSV file with path `fileName`.

5.20.1 Detailed Description

Definition of a variable length array of arbitrary precision complex numbers that are based on `mpfr`.

5.20.2 Typedef Documentation

5.20.2.1 array_t

```
typedef array_struct array_t[1]
```

Practical wrapper for `array_struct`.

To avoid the constant use `*` and `&` the type `array_t` is a pointer.

Definition at line 46 of file `array.h`.

5.20.3 Function Documentation

5.20.3.1 array_add()

```
bool array_add (
    array l,
    comp z,
    prec_t prec )
```

Adds the complex number `z` at the end of the array. The size of the array is automatically increased, if needed.

Parameters

<i>l</i>	the array
<i>z</i>	the new element to add to the array
<i>prec</i>	the precision of the number in the array

Returns

`true` if successfull, `false` otherwise.

5.20.3.2 array_append()

```
bool array_append (
    array l,
    char * fileName,
    int digits,
    bool verbose )
```

Writes the array `l` to the end of a CSV file with the given path `fileName`.

Parameters

<i>l</i>	the array
<i>fileName</i>	the path of the CSV file
<i>digits</i>	the number of digits after the decimal point of the numbers in the file
<i>verbose</i>	<code>true</code> to print information about the error, if any

Returns

`true` if successfull, `false` otherwise.

5.20.3.3 array_first_inf()

```
long array_first_inf (
    array l )
```

Checks if there are infinite values in the list `l`.

Parameters

<i>l</i>	the list
----------	----------

Returns

the position of the first infinite value in `l`, `-1` if there is none, `LONG_MAX` if some error occurred.

5.20.3.4 array_first_nan()

```
long array_first_nan (
    array l )
```

Checks if all complex numbers in the list `l` are well defined.

Parameters

<i>l</i>	the list
----------	----------

Returns

the position of the first undefined value in `l`, `-1` if there is none, `LONG_MAX` if some error occurred.

5.20.3.5 array_free()

```
bool array_free (
    array l )
```

Frees all the memory used by the array `l`, assuming the struct has been allocated with `malloc()`, for example with `array_new()`.

Parameters

<code>l</code>	the array
----------------	-----------

Returns

`true` if successfull, `false` otherwise.

5.20.3.6 array_get()

```
comp_ptr array_get (
    array l,
    ulong pos )
```

Returns the pointer of the element on position `pos` in the array `l`.

Parameters

<code>l</code>	the array
<code>pos</code>	the position of the element

Returns

the pointer to the element on position `pos` in `l`, `NULL` if some error occurred.

5.20.3.7 array_is_real()

```
bool array_is_real (
    array l )
```

Checks if all complex numbers in the array `l` are real.

Parameters

<code>l</code>	the array
----------------	-----------

Returns

`true` if all elements in the array are real, `false` otherwise.

5.20.3.8 array_new()

```
array array_new (
    ulong size )
```

Returns a new empty array with memory size at least `size`.

Parameters

<code>size</code>	the number of elements to allocate memory for
-------------------	---

Returns

the array, `NULL` if not enough memory is available

5.20.3.9 array_new_poly()

```
array array_new_poly (
    poly P,
    prec_t prec )
```

Returns the array of coefficients of the polynomial `P`.

Parameters

<code>P</code>	the polynomial
<code>prec</code>	the precision of the numbers in the array

Returns

the array, `NULL` if some error occurred.

5.20.3.10 array_new_polyr()

```
array array_new_polyr (
    polyr P,
    prec_t prec )
```

Returns the array of coefficients of the real polynomial `P`.

Parameters

<i>P</i>	the real polynomial
<i>prec</i>	the precision of the numbers in the array

Returns

the array, `NULL` if some error occurred.

5.20.3.11 array_poly()

```
poly array_poly (
    array l,
    prec_t prec )
```

Returns the polynomial with the coefficients in the array `l`.

Parameters

<i>l</i>	the array
<i>prec</i>	the precision of the coefficients of the polynomial

Returns

the polynomial, `NULL` if some error occurred.

5.20.3.12 array_polyr()

```
polyr array_polyr (
    array l,
    prec_t prec )
```

Returns the real polynomial with the coefficients the real parts of the complex numbers in the array `l`.

Parameters

<i>l</i>	the array
<i>prec</i>	the precision of the coefficients of the polynomial

Returns

the polynomial, `NULL` if some coefficients are not real or some other error occurred.

5.20.3.13 `array_read()`

```
array array_read (
    char * fileName,
    prec_t prec,
    bool verbose )
```

Reads a array of complex numbers of precision `prec` from the CSV file with path `fileName`.

Parameters

<i>fileName</i>	the path of the CSV file
<i>prec</i>	the precision
<i>verbose</i>	<code>true</code> to print information about the file and the error, if any

Returns

the array of complex numbers in the CSV file, `NULL` if some error occurred.

5.20.3.14 `array_write()`

```
bool array_write (
    array l,
    char * fileName,
    int digits,
    bool verbose )
```

Writes the array `l` to a CSV file with the given path `fileName`.

Parameters

<i>l</i>	the array
<i>fileName</i>	the path of the CSV file
<i>digits</i>	the number of digits after the decimal point of the numbers in the file
<i>verbose</i>	<code>true</code> to print information about the error, if any

Returns

`true` if successfull, `false` otherwise.

5.21 `/home/runner/work/FastPolyEval/FastPolyEval/code/tools/arrayf.h` File Reference

Definition of a variable length array of machine floating point complex numbers.

Data Structures

- struct [arrayf_struct](#)
A variable length list of machine floating point complex numbers.

Macros

- `#define LISTF_MIN_SIZE 100`
- `#define LISTF_SIZE_INCREASE 1.25`

Typedefs

- typedef [arrayf_struct](#) [arrayf_t](#)[1]
Practical wrapper for [arrayf_struct](#).
- typedef [arrayf_struct](#) * [arrayf](#)
Convenience pointer to [arrayf_struct](#).

Functions

- [arrayf](#) [arrayf_new](#) (ulong size)
Returns a new empty list with memory size at least `size`.
- [arrayf](#) [arrayf_new_polyfr](#) (polyfr P)
Returns the list of coefficients of the real polynomial `P`.
- [arrayf](#) [arrayf_new_polyf](#) (polyf P)
Returns the list of coefficients of the polynomial `P`.
- bool [arrayf_free](#) ([arrayf](#) l)
Frees all the memory used by the list `l`, assuming the struct has been allocated with `malloc()`, for example with `list_new()`.
- bool [arrayf_add](#) ([arrayf](#) l, compf z)
Adds the complex number `z` at the end of the list. The size of the list is automatically increased, if needed.
- compf_ptr [arrayf_get](#) ([arrayf](#) l, ulong pos)
Returns the pointer of the element on position `pos` in the list `l`.
- bool [arrayf_is_real](#) ([arrayf](#) l)
Checks if all complex numbers in the list `l` are real.
- long [arrayf_first_inf](#) ([arrayf](#) l)
Checks if there are infinite values in the list `l`.
- long [arrayf_first_nan](#) ([arrayf](#) l)
Checks if all complex numbers in the list `l` are well defined.
- polyfr [arrayf_polyfr](#) ([arrayf](#) l)
Returns the real polynomial with the coefficients the real parts of the complex numbers in the list `l`.
- polyf [arrayf_polyf](#) ([arrayf](#) l)
Returns the polynomial with the coefficients in the list `l`.
- bool [arrayf_write](#) ([arrayf](#) l, char *fileName, bool verbose)
Writes the list `l` to a CSV file with the given path `fileName`.
- bool [arrayf_append](#) ([arrayf](#) l, char *fileName, bool verbose)
Writes the list `l` to the end of a CSV file with the given path `fileName`.
- [arrayf](#) [arrayf_read](#) (char *fileName, bool verbose)
Reads a list of complex numbers from the CSV file with path `fileName`.

5.21.1 Detailed Description

Definition of a variable length array of machine floating point complex numbers.

5.21.2 Typedef Documentation

5.21.2.1 `arrayf_t`

```
typedef arrayf_struct arrayf_t[1]
```

Practical wrapper for `arrayf_struct`.

To avoid the constant use `*` and `&` the type `arrayf_t` is a pointer.

Definition at line 47 of file `arrayf.h`.

5.21.3 Function Documentation

5.21.3.1 `arrayf_add()`

```
bool arrayf_add (  
    arrayf l,  
    compf z )
```

Adds the complex number `z` at the end of the list. The size of the list is automatically increased, if needed.

Parameters

<code>l</code>	the list
<code>z</code>	the new element to add to the list

Returns

`true` if successfull, `false` otherwise.

5.21.3.2 `arrayf_append()`

```
bool arrayf_append (  
    arrayf l,
```

```
char * fileName,  
bool verbose )
```

Writes the list `l` to the end of a CSV file with the given path `fileName`.

Parameters

<i>l</i>	the list
<i>fileName</i>	the path of the CSV file
<i>verbose</i>	<code>true</code> to print information about the error, if any

Returns

`true` if successfull, `false` otherwise.

5.21.3.3 arrayf_first_inf()

```
long arrayf_first_inf (
    arrayf l )
```

Checks if there are infinite values in the list `l`.

Parameters

<i>l</i>	the list
----------	----------

Returns

the position of the first infinite value in `l`, `-1` if there is none, `LONG_MAX` if some error occurred.

5.21.3.4 arrayf_first_nan()

```
long arrayf_first_nan (
    arrayf l )
```

Checks if all complex numbers in the list `l` are well defined.

Parameters

<i>l</i>	the list
----------	----------

Returns

the position of the first undefined value in `l`, `-1` if there is none, `LONG_MAX` if some error occurred.

5.21.3.5 arrayf_free()

```
bool arrayf_free (
    arrayf l )
```

Frees all the memory used by the list `l`, assuming the struct has been allocated with `malloc()`, for example with `list_new()`.

Parameters

<code>l</code>	the list
----------------	----------

Returns

`true` if successfull, `false` otherwise.

5.21.3.6 arrayf_get()

```
compf_ptr arrayf_get (
    arrayf l,
    ulong pos )
```

Returns the pointer of the element on position `pos` in the list `l`.

Parameters

<code>l</code>	the list
<code>pos</code>	the position of the element

Returns

the pointer to the element on position `pos` in `l`, `NULL` if some error occurred.

5.21.3.7 arrayf_is_real()

```
bool arrayf_is_real (
    arrayf l )
```

Checks if all complex numbers in the list `l` are real.

Parameters

<code>l</code>	the list
----------------	----------

Returns

`true` if all elements in the list are real, `false` otherwise.

5.21.3.8 arrayf_new()

```
arrayf arrayf_new (
    ulong size )
```

Returns a new empty list with memory size at least `size`.

Parameters

<code>size</code>	the number of elements to allocate memory for
-------------------	---

Returns

the list, `NULL` if not enough memory is available

5.21.3.9 arrayf_new_polyf()

```
arrayf arrayf_new_polyf (
    polyf P )
```

Returns the list of coefficients of the polynomial `P`.

Parameters

<code>P</code>	the polynomial
----------------	----------------

Returns

the list, `NULL` if some error occurred.

5.21.3.10 arrayf_new_polyfr()

```
arrayf arrayf_new_polyfr (
    polyfr P )
```

Returns the list of coefficients of the real polynomial `P`.

Parameters

P	the real polynomial
-----	---------------------

Returns

the list, `NULL` if some error occurred.

5.21.3.11 arrayf_polyf()

```
polyf arrayf_polyf (
    arrayf l )
```

Returns the polynomial with the coefficients in the list `l`.

Parameters

<code>l</code>	the list
----------------	----------

Returns

the polynomial, `NULL` if some error occurred.

5.21.3.12 arrayf_polyfr()

```
polyfr arrayf_polyfr (
    arrayf l )
```

Returns the real polynomial with the coefficients the real parts of the complex numbers in the list `l`.

Parameters

<code>l</code>	the list
----------------	----------

Returns

the polynomial, `NULL` if some coefficients are not real or some other error occurred.

5.21.3.13 arrayf_read()

```
arrayf arrayf_read (
    char * fileName,
    bool verbose )
```

Reads a list of complex numbers from the CSV file with path `fileName`.

Parameters

<code>fileName</code>	the path of the CSV file
<code>verbose</code>	<code>true</code> to print information about the file and the error, if any

Returns

the list of complex numbers in the CSV file, `NULL` if some error occurred.

5.21.3.14 `arrayf_write()`

```
bool arrayf_write (
    arrayf l,
    char * fileName,
    bool verbose )
```

Writes the list `l` to a CSV file with the given path `fileName`.

Parameters

<code>l</code>	the list
<code>fileName</code>	the path of the CSV file
<code>verbose</code>	<code>true</code> to print information about the error, if any

Returns

`true` if successfull, `false` otherwise.

5.22 `/home/runner/work/FastPolyEval/FastPolyEval/code/tools/chrono.h` File Reference

Tools for precise time measuring.

Macros

- `#define BILLION 1000000000L`
One billion (nanoseconds in one second).
- `#define MILLION 1000000L`
One billion (milliseconds in one second).
- `#define MINUTE 60000L`
Miliseconds in one minute.
- `#define HOUR (60 * MINUTE)`

- Milliseconds in one hour.*
 - `#define DAY (24 * HOUR)`
 - Milliseconds in one day.*
 - `#define USED_CLOCK CLOCK_MONOTONIC`
 - The ID of the system clock to use.*

Typedefs

- `typedef struct timespec ptime`
 - Short type name for precise time.*

Functions

- `void millis (long ms, char *str)`
 - Formats the time duration in milliseconds `ms` into human readable string `str`.*
- `void nanos (long ns, char *str)`
 - Formats the time duration in nanoseconds `ns` into human readable string `str`.*
- `ulong lap (ptime *ts, char *str)`
 - Computes and prints the time lapse since `ts`.*
- `void print_time_res (void)`
 - Prints the timer resolution.*

5.22.1 Detailed Description

Tools for precise time measuring.

5.22.2 Function Documentation

5.22.2.1 lap()

```
ulong lap (  
    ptime * ts,  
    char * str )
```

Computes and prints the time lapse since `ts`.

Pretty prints the elapsed time into the string `str` and updates `ts` to the current time. Does not fail if `str==NULL`, can also be used as a starter for the chronometer.

Warning

The string `str` should be at least 90 bytes long.

Parameters

<i>ts</i>	old time stamp
<i>str</i>	human readable string for the time increment

Returns

the total time in nanos

5.22.2.2 millis()

```
void millis (
    long ms,
    char * str )
```

Formats the time duration in milliseconds *ms* into human readable string *str*.

Warning

The string *str* should be at least 90 bytes long.

Parameters

<i>ms</i>	the time lapse in milliseconds
<i>str</i>	the human readable print

5.22.2.3 nanos()

```
void nanos (
    long ns,
    char * str )
```

Formats the time duration in nanoseconds *ns* into human readable string *str*.

Warning

The string *str* should be at least 90 bytes long.

Parameters

<i>ns</i>	the time lapse in milliseconds
<i>str</i>	the human readable print

5.23 /home/runner/work/FastPolyEval/FastPolyEval/code/tools/debug.h File Reference

A few basic tool for easy debug when usinf MPFR.

Functions

- `char * pm (mpfr_t x, int dig)`
Return a string containing the value of `x` with `dig` digits.
- `char * pc (comp z, int dig)`
Return a string containing the value of `x` with `dig` digits.
- `char * pcf (compf z)`
Return a string containing the value of `z`.

5.23.1 Detailed Description

A few basic tool for easy debug when usinf MPFR.

5.23.2 Function Documentation

5.23.2.1 pc()

```
char* pc (
    comp z,
    int dig )
```

Return a string containing the value of `x` with `dig` digits.

The format is that of a CSV file line, the real and imaginary components are separated by a comma.

Warning

The user is responsible to free the memory containing the results.

Parameters

<code>z</code>	a complex number, with MPFR real and imaginary parts
<code>dig</code>	the number of digits to print

Returns

the string containing the value of `z`, `NULL` if some error occurred

5.23.2.2 pcf()

```
char* pcf (
    compf z )
```

Return a string containing the value of *z*.

The format is that of a CSV file line, the real and imaginary components are separated by a comma.

Warning

The user is responsible to free the memory containing the results.

Parameters

<i>z</i>	a complex number, with machine numbers real and imaginary parts
----------	---

Returns

the string containing the value of *z*, `NULL` if some error occurred

5.23.2.3 pm()

```
char* pm (
    mpfr_t x,
    int dig )
```

Return a string containing the value of *x* with *dig* digits.

Warning

The user is responsible to free the memory containing the results.

Parameters

<i>x</i>	a real number, MPFR
<i>dig</i>	the number of digits to print

Returns

the string containing the value of *z*, `NULL` if some error occurred