

---

**baryrat**

**Mar 01, 2022**



---

## Contents:

---

<b>1</b>	<b>Indices and tables</b>	<b>7</b>
	<b>Python Module Index</b>	<b>9</b>
	<b>Index</b>	<b>11</b>



A Python package for barycentric rational approximation.

**class** `baryrat.BarycentricRational` ( $z, f, w$ )

A class representing a rational function in barycentric representation.

#### Parameters

- $z$  (*array*) – the interpolation nodes
- $f$  (*array*) – the values at the interpolation nodes
- $w$  (*array*) – the weights

The rational function has the interpolation property  $r(z_j) = f_j$  at all nodes where  $w_j \neq 0$ .

**\_\_call\_\_** ( $x$ )

Evaluate rational function at all points of  $x$ .

**degree** ( $tol=1e-12$ )

Compute the pair  $(m, n)$  of true degrees of the numerator and denominator.

**degree\_denom** ( $tol=1e-12$ )

Compute the true degree of the denominator polynomial.

Uses a result from [Berrut, Mittelmann 1997].

**degree\_numer** ( $tol=1e-12$ )

Compute the true degree of the numerator polynomial.

Uses a result from [Berrut, Mittelmann 1997].

**denominator** ()

Return a new *BarycentricRational* which represents the denominator polynomial.

**eval\_deriv** ( $x, k=1$ )

Evaluate the  $k$ -th derivative of this rational function at a scalar node  $x$ , or at each point of an array  $x$ . Only the cases  $k \leq 2$  are currently implemented.

Note that this function may incur significant numerical error if  $x$  is very close (but not exactly equal) to a node of the barycentric rational function.

#### References

<https://doi.org/10.1090/S0025-5718-1986-0842136-8> (C. Schneider and W. Werner, 1986)

**gain** ()

The gain in a poles-zeros-gain representation of the rational function, or equivalently, the value at infinity.

**jacobians** ( $x$ )

Compute the Jacobians of  $r(x)$ , where  $x$  may be a vector of evaluation points, with respect to the node, value, and weight vectors.

The evaluation points  $x$  may not lie on any of the barycentric nodes (unimplemented).

**Returns** A triple of arrays with as many rows as  $x$  has entries and as many columns as the barycentric function has nodes, representing the Jacobians with respect to `self.nodes`, `self.values`, and `self.weights`, respectively.

**numerator** ()

Return a new *BarycentricRational* which represents the numerator polynomial.

**order**

The order of the barycentric rational function, that is, the maximum degree that its numerator and denominator may have, or the number of interpolation nodes minus one.

**poles** (*use\_mp=False*)

Return the poles of the rational function.

If *use\_mp* is *True*, uses the *flamp* multiple precision package to compute the result. This option is automatically enabled if *uses\_mp()* is *True*.

**polres** (*use\_mp=False*)

Return the poles and residues of the rational function.

If *use\_mp* is *True*, uses the *flamp* multiple precision package to compute the result. This option is automatically enabled if *uses\_mp()* is *True*.

**reciprocal** ()

Return a new *BarycentricRational* which is the reciprocal of this one.

**reduce\_order** ()

Return a new *BarycentricRational* which represents the same rational function as this one, but with minimal possible order.

See (Ionita 2013), PhD thesis.

**uses\_mp** ()

Checks whether any of the data of this rational function uses extended precision.

**zeros** (*use\_mp=False*)

Return the zeros of the rational function.

If *use\_mp* is *True*, uses the *flamp* multiple precision package to compute the result. This option is automatically enabled if *uses\_mp()* is *True*.

`baryrat.aaa(Z, F, tol=1e-13, mmax=100, return_errors=False)`

Compute a rational approximation of *F* over the points *Z* using the AAA algorithm.

#### Parameters

- **Z** (*array*) – the sampling points of the function. Unlike for interpolation algorithms, where a small number of nodes is preferred, since the AAA algorithm chooses its support points adaptively, it is better to provide a finer mesh over the support.
- **F** – the function to be approximated; can be given as a function or as an array of function values over *Z*.
- **tol** – the approximation tolerance
- **mmax** – the maximum number of iterations/degree of the resulting approximant
- **return\_errors** – if *True*, also return the history of the errors over all iterations

**Returns** an object which can be called to evaluate the rational function, and can be queried for the poles, residues, and zeros of the function.

**Return type** *BarycentricRational*

For more information, see the paper

The AAA Algorithm for Rational Approximation

Yuji Nakatsukasa, Olivier Sete, and Lloyd N. Trefethen

SIAM Journal on Scientific Computing 2018 40:3, A1494-A1522

<https://doi.org/10.1137/16M1106122>

as well as the Chebfun package <<http://www.chebfun.org>>. This code is an almost direct port of the Chebfun implementation of *aaa* to Python.

`baryrat.brasil` (*f*, *interval*, *deg*, *tol*=0.0001, *maxiter*=1000, *max\_step\_size*=0.1, *step\_factor*=0.1, *npi*=-30, *init\_steps*=100, *info*=False)

Best Rational Approximation by Successive Interval Length adjustment.

Computes best rational or polynomial approximations in the maximum norm by the BRASIL algorithm (see reference below).

## References

<https://doi.org/10.1007/s11075-020-01042-0>

### Parameters

- **f** – the scalar function to be approximated. Must be able to operate on arrays of arguments.
- **interval** – the bounds (*a*, *b*) of the approximation interval
- **deg** – the degree of the numerator *m* and denominator *n* of the rational approximation; either an integer (*m*=*n*) or a pair (*m*, *n*). If *n* = 0, a polynomial best approximation is computed.
- **tol** – the maximum allowed deviation from equioscillation
- **maxiter** – the maximum number of iterations
- **max\_step\_size** – the maximum allowed step size
- **step\_factor** – factor for adaptive step size choice
- **npi** – points per interval for error calculation. If *npi* < 0, golden section search with *-npi* iterations is used instead of sampling. For high-accuracy results, *npi*=-30 is typically a good choice.
- **init\_steps** – how many steps of the initialization iteration to run
- **info** – whether to return an additional object with details

**Returns** the computed rational approximation. If *info* is True, instead returns a pair containing the approximation and an object with additional information (see below).

**Return type** *BarycentricRational*

The *info* object returned along with the approximation if *info*=True has the following members:

- **converged** (bool): whether the method converged to the desired tolerance **tol**
- **error** (float): the maximum error of the approximation
- **deviation** (float): the relative error between the smallest and the largest equioscillation peak. The convergence criterion is **deviation** <= **tol**.
- **nodes** (array): the abscissae of the interpolation nodes (2\*deg + 1)
- **iterations** (int): the number of iterations used, including the initialization phase
- **errors** (array): the history of the maximum error over all iterations
- **deviations** (array): the history of the deviation over all iterations
- **stepsizes** (array): the history of the adaptive step size over all iterations

Additional information about the resulting rational function, such as poles, residues and zeroes, can be queried from the *BarycentricRational* object itself.

---

**Note:** This function supports `gmpy2` for extended precision. To enable this, specify the interval  $(a, b)$  as `mpfr` numbers, e.g., `interval=(mpfr(0), mpfr(1))`. Also make sure that the function  $f$  consumes and outputs arrays of `mpfr` numbers; the Numpy function `numpy.vectorize()` may help with this.

---

`baryrat.chebyshev_nodes(num_nodes, interval=(-1.0, 1.0))`

Compute `num_nodes` Chebyshev nodes of the first kind in the given interval.

`baryrat.floater_hormann(nodes, values, blending)`

Compute the Floater-Hormann rational interpolant for the given nodes and values. See (Floater, Hormann 2007), DOI 10.1007/s00211-007-0093-y.

The blending parameter (usually called  $d$  in the literature) is an integer between 0 and  $n$  (inclusive), where  $n+1$  is the number of interpolation nodes. For functions with higher smoothness, the blending parameter may be chosen higher. For  $d=n$ , the result is the polynomial interpolant.

Returns an instance of *BarycentricRational*.

`baryrat.interpolate_poly(nodes, values)`

Compute the interpolating polynomial for the given nodes and values in barycentric form.

`baryrat.interpolate_rat(nodes, values, use_mp=False)`

Compute a rational function which interpolates the given nodes/values.

#### Parameters

- **nodes** (*array*) – the interpolation nodes; must have odd length and be passed in strictly increasing or decreasing order
- **values** (*array*) – the values at the interpolation nodes
- **use\_mp** (*bool*) – whether to use `gmpy2` for extended precision. Is automatically enabled if `nodes` or `values` use `gmpy2`.

**Returns** the rational interpolant. If there are  $2n + 1$  nodes, both the numerator and denominator have degree at most  $n$ .

**Return type** *BarycentricRational*

## References

<https://doi.org/10.1109/LSP.2007.913583>

`baryrat.interpolate_with_degree(nodes, values, deg, use_mp=False)`

Compute a rational function which interpolates the given nodes/values with given degree  $m$  of the numerator and  $n$  of the denominator.

#### Parameters

- **nodes** (*array*) – the interpolation nodes
- **values** (*array*) – the values at the interpolation nodes
- **deg** – a pair  $(m, n)$  of the degrees of the interpolating rational function. The number of interpolation nodes must be  $m + n + 1$ .
- **use\_mp** (*bool*) – whether to use `gmpy2` for extended precision. Is automatically enabled if `nodes` or `values` use `gmpy2`.

**Returns** the rational interpolant

**Return type** *BarycentricRational*



## References

[https://doi.org/10.1016/S0377-0427\(96\)00163-X](https://doi.org/10.1016/S0377-0427(96)00163-X)

`baryrat.interpolate_with_poles` (*nodes*, *values*, *poles*, *use\_mp=False*)

Compute a rational function which interpolates the given values at the given nodes and which has the given poles.

The arrays `nodes` and `values` should have length  $n$ , and `poles` should have length  $n - 1$ .



# CHAPTER 1

---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`



**b**

baryrat, [1](#)



## Symbols

`__call__()` (*baryrat.BarycentricRational method*), 1

## A

`aaa()` (*in module baryrat*), 2

## B

`BarycentricRational` (*class in baryrat*), 1

`baryrat` (*module*), 1

`brasil()` (*in module baryrat*), 2

## C

`chebyshev_nodes()` (*in module baryrat*), 4

## D

`degree()` (*baryrat.BarycentricRational method*), 1

`degree_denom()` (*baryrat.BarycentricRational method*), 1

`degree_numer()` (*baryrat.BarycentricRational method*), 1

`denominator()` (*baryrat.BarycentricRational method*), 1

## E

`eval_deriv()` (*baryrat.BarycentricRational method*), 1

## F

`floater_hormann()` (*in module baryrat*), 4

## G

`gain()` (*baryrat.BarycentricRational method*), 1

## I

`interpolate_poly()` (*in module baryrat*), 4

`interpolate_rat()` (*in module baryrat*), 4

`interpolate_with_degree()` (*in module baryrat*), 4

`interpolate_with_poles()` (*in module baryrat*), 5

## J

`jacobians()` (*baryrat.BarycentricRational method*), 1

## N

`numerator()` (*baryrat.BarycentricRational method*), 1

## O

`order` (*baryrat.BarycentricRational attribute*), 1

## P

`poles()` (*baryrat.BarycentricRational method*), 2

`polres()` (*baryrat.BarycentricRational method*), 2

## R

`reciprocal()` (*baryrat.BarycentricRational method*), 2

`reduce_order()` (*baryrat.BarycentricRational method*), 2

## U

`uses_mp()` (*baryrat.BarycentricRational method*), 2

## Z

`zeros()` (*baryrat.BarycentricRational method*), 2