pPXF: Full Spectrum and SED Fitting of Galactic and Stellar Spectra

(Package v8.1, June 2022)

This `pPXF` package contains a Python implementation of the Penalized PiXel-Fitting (`pPXF`) method to fit the stellar and gas kinematics, as well as the stellar population and the photometry (SED) of galaxies. The method was originally described in Cappellari & Emsellem (2004) and was substantially upgraded in subsequent years and particularly in Cappellari (2017).

Attribution

If you use this software for your research, please cite at least Cappellari (2017), or both `pPXF` papers above. The BibTeX entry for the paper is:

```
@ARTICLE{Cappellari2017,
    author = {{Cappellari}, M.},
    title = "{Improving the full spectrum fitting method:
        accurate convolution with Gauss-Hermite functions}",
    journal = {MNRAS},
    eprint = {1607.08538},
    year = 2017,
    volume = 466,
    pages = {798-811},
    doi = {10.1093/mnras/stw3020}
}
```

Installation

install with:

```
pip install ppxf
```

Without write access to the global `site-packages` directory, use:

```
pip install --user ppxf
```

To upgrade `pPXF` to the latest version use:

```
pip install --upgrade ppxf
```

Usage Examples

To learn how to use the `pPXF` package, copy, modify and run the example programs in the `ppxf/examples` directory. It can be found within the main `ppxf` package installation folder inside site-packages. The detailed documentation is contained in the docstring of the file `ppxf.py`, or on PyPi or as PDF from https://purl.org/cappellari/software.

Examples as Jupyter Notebooks are also available on my GitHub repository.

---

pPXF Purpose

Extract the galaxy stellar and gas kinematics, stellar population and gas emission by fitting a set of templates to an observed spectrum, or to a combination of a spectrum and photometry (SED), via full-spectrum fitting, using the Penalized PiXel-Fitting (`pPXF`) method originally described in

Cappellari & Emsellem (2004)

and substantially upgraded in subsequent years and particularly in

Cappellari (2017).

The following key optional features are also available:

1) An optimal template, positive linear combination of different input templates, can be fitted together with the kinematics.
2) One can enforce smoothness on the template weights during the fit. This is useful to attach a physical meaning to the weights e.g. in terms of the star formation history of a galaxy.
3) One can fit multiple kinematic components for both the stars and the gas emission lines. Both the stellar and gas LOSVD can be penalized and can be described by a general Gauss-Hermite series.
4) One can fit simultaneously a spectrum and a set of photometric measurements (SED fitting).
5) Any parameter of the LOSVD (e.g. sigma) for any kinematic component can either be fitted or held fixed to a given value, while other parameters are fitted. Alternatively, parameters can be constrained to lie within given limits or tied by nonlinear equalities to other parameters.
6) One can enforce linear equality/inequality constraints on either the template weights or the kinematic parameters.
7) Additive and/or multiplicative polynomials can be included to adjust the continuum shape of the template to the observed spectrum.
8) Iterative sigma clipping can be used to clean the spectrum.
9) It is possible to fit a mirror-symmetric LOSVD to two spectra at the same time. This is useful for spectra taken at point-symmetric spatial positions with respect to the center of an equilibrium stellar system.
10) One can include sky spectra in the fit, to deal with cases where the sky dominates the observed spectrum and an accurate sky subtraction is critical.
11) One can derive an estimate of the reddening in the spectrum. This can be done independently for the stellar spectrum or the gas emission lines.
12) The covariance matrix can be input instead of the error spectrum, to account for correlated errors in the spectral pixels.

13) One can specify the weights fraction between two kinematics components, e.g. to model bulge and disk contributions.
14) One can use templates with higher resolution than the galaxy, to improve the accuracy of the LOSVD extraction at low dispersion.

Calling Sequence

```
from ppxf.ppxf import ppxf

pp = ppxf(templates, galaxy, noise, velscale, start,
          bias=None, bounds=None, clean=False, component=0, constr_templ={},
          constr_kinem={}, degree=4, fixed=None, fraction=None, ftol=1e-4,
          gas_component=None, gas_names=None, gas_reddening=None,
          global_search=False, goodpixels=None, lam=None, lam_temp=None,
          linear=False, linear_method='lsq_box', mask=None, method='capfit',
          mdegree=0, moments=2, phot={}, plot=False, quiet=False,
          reddening=None, reddening_func=None, reg_dim=None, reg_ord=2,
          regul=0, sigma_diff=0, sky=None, templates_rfft=None, tied=None,
          trig=False, velscale_ratio=1, vsyst=0, x0=None)

print(pp.sol)  # print best-fitting kinematics (V, sigma, h3, h4)
pp.plot()      # Plot best fit with gas lines and photometry
```

Example programs are in the `ppxf/examples` directory. It can be found within the main `ppxf` package installation folder inside site-packages.

Examples as Jupyter Notebooks are also available on my GitHub repository.

Input Parameters

templates: array_like with shape (n_pixels_temp, n_templates) Vector containing a single optimized spectral template, or an array of dimensions `templates[n_pixels_temp, n_templates]` containing different stellar or gas emission spectral templates to be optimized during the fit of the `galaxy` spectrum. It has to be `n_pixels_temp >= galaxy.size`.

To apply linear regularization to the `weights` via the keyword `regul`, `templates` should be an array of

- 2-dim: `templates[n_pixels_temp, n_age]`,
- 3-dim: `templates[n_pixels_temp, n_age, n_metal]`
- 4-dim: `templates[n_pixels_temp, n_age, n_metal, n_alpha]`

depending on the number of population variables one wants to study. This can be useful to try to attach a physical meaning to the output `weights`, in term of the galaxy star formation history and chemical composition distribution. In that case the templates may represent single stellar population SSP models and should be arranged in sequence of increasing age, metallicity or alpha (or alternative population parameters) along the second, third or fourth dimension of the array respectively.

IMPORTANT: The templates must be normalized to unity order of magnitude, to avoid numerical instabilities.

When studying stellar population, the relative fluxes of the templates are important. For this reason one must scale all templates by a scalar. This can be done with a command like:

```
templates /= np.median(templates)
```

When using individual stars as templates, the relative fluxes are generally irrelevant and one can normalize each template independently. This can be done with a command like:

```
templates /= np.median(templates, 0)
```

galaxy: array_like with shape (n_pixels,) Vector containing the spectrum of the galaxy to be measured. The star and the galaxy spectra have to be logarithmically rebinned but the continuum should *not* be subtracted. The rebinning may be performed with the `log_rebin` routine in `ppxf.ppxf_util`. The units of the spectrum flux are arbitrary. One can use e.g. `erg/(s cm^2 A)` or `erg/(s cm^2 pixel)` as long as the same are used for `templates`. But see the note at the end of this section.

For high redshift galaxies, it is generally easier to bring the spectra close to the restframe wavelength, before doing the `pPXF` fit. This can be done by dividing the observed wavelength by `(1 + z)`, where `z` is a rough estimate of the galaxy redshift. There is no need to modify the spectrum in any way, given that a red shift corresponds to a linear shift of the log-rebinned spectrum. One just needs to compute the wavelength range in the rest-frame and adjust the instrumental resolution of the galaxy observations. See Section 2.4 of Cappellari (2017) for details.

`galaxy` can also be an array of dimensions `galaxy[n_pixels, 2]` containing two spectra to be fitted, at the same time, with a reflection-symmetric LOSVD. This is useful for spectra taken at point-symmetric spatial positions with respect to the center of an equilibrium stellar system. For a discussion of the usefulness of this two-sided fitting see e.g. Section 3.6 of Rix & White (1992).

IMPORTANT: (1) For the two-sided fitting the `vsyst` keyword has to be used. (2) Make sure the spectra are rescaled to be not too many order of magnitude different from unity, to avoid numerical instability. E.g. units of `erg/(s cm^2 A)` may cause problems!

noise: array_like with shape (n_pixels,) Vector containing the `1*sigma` uncertainty (per spectral pixel) in the `galaxy` spectrum, or covariance matrix describing the correlated uncertainties in the galaxy spectrum. Of course this vector/matrix must have the same units as the galaxy spectrum.

The overall normalization of the `noise` does not affect the location of the `chi2` minimum. For this reason one can measure reliable kinematics even when the noise is not accurately know.

If `galaxy` is a `n_pixels*2` array, `noise` has to be an array with the same dimensions.

When `noise` has dimensions `n_pixels*n_pixels` it is assumed to contain the covariance matrix with elements `cov(i, j)`. When the errors in the spectrum are uncorrelated it is mathematically equivalent to input in `pPXF` an error vector `noise=errvec` or a `n_pixels*n_pixels` diagonal matrix `noise = np.diag(errvec**2)` (note squared!).

IMPORTANT: the penalty term of the `pPXF` method is based on the *relative* change of the fit residuals. For this reason, the penalty will work as expected even if the normalization of the `noise` is arbitrary. See Cappellari & Emsellem (2004) for details. If no reliable noise is available this keyword can just be set to:

```
noise = np.ones_like(galaxy)  # Same uncertainty for all pixels
```

velscale: float Velocity scale of the spectra in km/s per pixel. It has to be the same for both the galaxy and the template spectra. An exception is when the `velscale_ratio` keyword is used, in which case one can input `templates` with smaller `velscale` than `galaxy`.

`velscale` is precisely *defined* in `pPXF` by `velscale = c*np.diff(np.log(lambda))`, which is approximately `velscale ~ c*np.diff(lambda)/lambda`. See Section 2.3 of Cappellari (2017) for details.

start: Vector, or list/array of vectors `[start1, start2, ...]`, with the initial estimate for the LOSVD parameters.

When LOSVD parameters are not held fixed, each vector only needs to contain `start = [velStart, sigmaStart]` the initial guess for the velocity and the velocity dispersion in km/s. The starting values for h3-h6 (if they are fitted) are all set to zero by default. In other words, when `moments=4`:

```
start = [velStart, sigmaStart]
```

is interpreted as:

```
start = [velStart, sigmaStart, 0, 0]
```

When the LOSVD for some kinematic components is held fixed (see `fixed` keyword), all values for `[Vel, Sigma, h3, h4,...]` can be provided.

Unless a good initial guess is available, it is recommended to set the starting `sigma >= 3*velscale` in km/s (i.e. 3 pixels). In fact, when the sigma is very low, and far from the true solution, the `chi^2` of the fit becomes weakly sensitive to small variations in sigma (see `pPXF` paper). In some instances, the near-constancy of `chi^2` may cause premature convergence of the optimization.

In the case of two-sided fitting a good starting value for the velocity is `velStart = 0.0` (in this case `vsyst` will generally be nonzero). Alternatively on should keep in mind that `velStart` refers to the first input galaxy spectrum, while the second will have velocity `-velStart`.

With multiple kinematic components `start` must be a list of starting values, one for each different component.

EXAMPLE: We want to fit two kinematic components. We fit 4 moments for the first component and 2 moments for the second one as follows:

```
component = [0, 0, ... 0, 1, 1, ... 1]
moments = [4, 2]
start = [[V1, sigma1], [V2, sigma2]]
```

Optional Keywords

bias: float, optional When `moments > 2`, this parameter biases the `(h3, h4, ...)` measurements towards zero (Gaussian LOSVD) unless their inclusion significantly decreases the error in the fit. Set this to `bias=0` not to bias the fit: the solution (including `[V, sigma]`) will be noisier in that case. This parameter is ignored if `moments <= 2`. The default `bias` should provide acceptable results in most cases, but it would be safe to test it with Monte

Carlo simulations as described in the section "How to Set the Kinematic Penalty Keyword" near the end of the documentation. This keyword precisely corresponds to the parameter `lambda` in the Cappellari & Emsellem (2004) paper. Note that the penalty depends on the *relative* change of the fit residuals, so it is insensitive to proper scaling of the `noise` vector. A nonzero `bias` can be safely used even without a reliable `noise` spectrum, or with equal weighting for all pixels.

bounds: Lower and upper bounds for every kinematic parameter. This is an array, or list of arrays, with the same dimensions as `start`, except for the last dimension, which is 2. In practice, for every element of `start` one needs to specify a pair of values `[lower, upper]`.

EXAMPLE: We want to fit two kinematic components, with 4 moments for the first component and 2 for the second (e.g. stars and gas). In this case:

```
moments = [4, 2]
start_stars = [V1, sigma1, 0, 0]
start_gas = [V2, sigma2]
start = [start_stars, start_gas]
```

then we can specify boundaries for each kinematic parameter as:

```
bounds_stars = [[V1_lo, V1_up], [sigma1_lo, sigma1_up],
                [-0.3, 0.3], [-0.3, 0.3]]
bounds_gas = [[V2_lo, V2_up], [sigma2_lo, sigma2_up]]
bounds = [bounds_stars, bounds_gas]
```

component: When fitting more than one kinematic component, this keyword should contain the component number of each input template. In principle, every template can belong to a different kinematic component.

EXAMPLE: We want to fit the first 50 templates to component 0 and the last 10 templates to component 1. In this case:

```
component = [0]*50 + [1]*10
```

which, in Python syntax, is equivalent to:

```
component = [0, 0, ... 0, 1, 1, ... 1]
```

This keyword is especially useful when fitting both emissions (gas) and absorption (stars) templates simultaneously (see the example for the `moments` keyword).

constr_kinem: dictionary, optional It enforces linear constraints on the kinematic parameters during the fit. This is specified by the following dictionary, where `A_ineq` and `A_eq` are arrays (have `A.ndim = 2`), while `b_ineq` and `b_eq` are vectors (have `b.ndim = 1`). Either the `_eq` or the `_ineq` keys can be omitted if not needed:

```
constr_kinem = {"A_ineq": A_ineq, "b_ineq": b_ineq, "A_eq": A_eq, "b_eq": b_eq}
```

The resulting pPXF kinematics solution will satisfy the following linear matrix inequalities and/or equalities:

```
params = np.ravel(pp.sol)  # Unravel for multiple components
A_ineq @ params <= b_ineq
A_eq @ params == b_eq
```

IMPORTANT: the starting guess `start` must satisfy the constraints, or in other words, it must lie in the feasible region.

Inequalities can be used e.g. to force one kinematic component to have larger velocity or dispersion than another one. This is useful e.g. when extracting two stellar kinematic components or when fitting both narrow and broad components of gas emission lines.

EXAMPLES: We want to fit two kinematic components, with two moments for both the first and second component. In this case:

```
moments = [2, 2]
start = [[V1, sigma1], [V2, sigma2]]
```

then we can set the constraint `sigma1 >= 3*sigma2` as follows:

```
A_ineq = [[0, -1, 0, 3]]  # 0*V1 - 1*sigma1 + 0*V2 + 3*sigma2 <= 0
b_ineq = [0]
constr_kinem = {"A_ineq": A_ineq, "b_ineq": b_ineq}
```

We can set the constraint `sigma1 >= sigma2 + 2*velscale` as follows:

```
A_ineq = [[0, -1, 0, 1]]  # -sigma1 + sigma2 <= -2*velscale
b_ineq = [-2]             # kinem. in pixels (-2 --> -2*velscale)!
constr_kinem =  {"A_ineq": A_ineq, "b_ineq": b_ineq}
```

We can set both the constraints `V1 >= V2` and `sigma1 >= sigma2 + 2*velscale` as follows:

```
A_ineq = [[-1, 0, 1, 0],   # -V1 + V2 <= 0
          [0, -1, 0, 1]]   # -sigma1 + sigma2 <= -2*velscale
b_ineq = [0, -2]           # kinem. in pixels (-2 --> -2*velscale)!
constr_kinem =  {"A_ineq": A_ineq, "b_ineq": b_ineq}
```

We can constrain the velocity dispersion of the second kinematic component to differ less than 10% from that of the first component `sigma1/1.1 <= sigma2 <= sigma1*1.1` as follows:

```
A_ineq = [[0, 1/1.1, 0, -1],   # +sigma1/1.1 - sigma2 <= 0
          [0, -1.1,  0,  1]]   # -sigma1*1.1 + sigma2 <= 0
b_ineq = [0, 0]
constr_kinem =  {"A_ineq": A_ineq, "b_ineq": b_ineq}
```

EXAMPLE: We want to fit three kinematic components, with four moments for the first and two for the rest. In this case:

```
moments = [4, 2, 2]
start = [[V1, sigma1, 0, 0], [V2, sigma2], [V3, sigma3]]
```

then we can set the constraints `sigma3 >= sigma1 + 2*velscale` and `V1 <= V2 <= V3` as follows:

```
A_ineq = [[0, 1, 0, 0,  0, 0,  0, -1],  # sigma1 - sigma3 <= -2*velscale
          [1, 0, 0, 0, -1, 0,  0,  0],  # V1 - V2 <= 0
          [0, 0, 0, 0,  1, 0, -1,  0]]  # V2 - V3 <= 0
b_ineq = [-2, 0, 0]           # kinem. in pixels (-2 --> -2*velscale)!
constr_kinem =  {"A_ineq": A_ineq, "b_ineq": b_ineq}
```

NOTE: When possible, it is more efficient to set equality constraints using the `tied` keyword, instead of setting `A_eq` and `b_eq` in `constr_kinem`.

constr_templ: dictionary, optional It enforces linear constraints on the template weights during the fit. This is specified by the following dictionary, where `A_ineq` and `A_eq` are arrays (have `A.ndim = 2`), while `b_ineq` and `b_eq` are vectors (have `b.ndim = 1`). Either the `_eq` or the `_ineq` keys can be omitted if not needed:

```
constr_templ = {"A_ineq": A_ineq, "b_ineq": b_ineq, "A_eq": A_eq, "b_eq": b_eq}
```

The resulting pPXF solution will satisfy the following linear matrix inequalities and/or equalities:

```
A_ineq @ pp.weights <= b_ineq
A_eq @ pp.weights == b_eq
```

Inequality can be used e.g. to constrain the fluxes of emission lines to lie within prescribed ranges. Equalities can be used e.g. to force the weights for different kinematic components to contain prescribed fractions of the total weights.

EXAMPLES: We are fitting a spectrum using four templates, the first two templates belong to one kinematic component and the rest to the other. (NOTE: This 4-templates example is for illustration, but in real applications one will use many more than two templates per component!) This implies we have:

```
component=[0, 0, 1, 1]
```

then we can set the equality constraint that the sum of the weights of the first kinematic component is a given `fraction` of the total:

```
pp.weights[component == 0].sum()/pp.weights.sum() == fraction
```

as follows [see equation 30 of Cappellari (2017)]:

```
A_eq = [[fraction - 1, fraction - 1, fraction, fraction]]
b_eq = [0]
constr_templ = {"A_eq": A_eq, "b_eq": b_eq}
```

An identical result can be obtained in this case using the legacy `fraction` keyword, but `constr_templ` additionally allows for general linear constraints for multiple kinematic components.

Similarly, we can set the inequality constraint that the total weights of each of the two kinematic components is larger than `fraction`:

```
fraction <= pp.weights[component == 0].sum()/pp.weights.sum()
fraction <= pp.weights[component == 1].sum()/pp.weights.sum()
```

as follows:

```
A_ineq = [[fraction - 1, fraction - 1, fraction, fraction],
          [fraction, fraction, fraction - 1, fraction - 1]]
b_ineq = [0, 0]
constr_templ = {"A_ineq": A_ineq, "b_ineq": b_ineq}
```

We can constrain the ratio of the first two templates weights to lie in the interval `ratio_min <= w[0]/w[1] <= ratio_max` as follows:

```
A_ineq = [[-1, ratio_min, 0, 0],    # -w[0] + ratio_min*w[1] <= 0
          [1, -ratio_max, 0, 0]]    # +w[0] - ratio_max*w[1] <= 0
b_ineq = [0, 0]
constr_templ = {"A_ineq": A_ineq, "b_ineq": b_ineq}
```

If we have six templates for three kinematics components:

```
component=[0, 0, 1, 1, 2, 2]
```

we can set the fractions for the first two components to be `fraction1` and `fraction2` (of the total weights) respectively as follows (the third components will be `1 - fraction1 - fraction2`):

```
A_eq = [[fraction1 - 1, fraction1 - 1, fraction1, fraction1, fraction1, fraction1],
        [fraction2, fraction2, fraction2 - 1, fraction2 - 1, fraction2, fraction2]]
b_eq = [0, 0]
constr_templ = {"A_eq": A_eq, "b_eq": b_eq}
```

clean: bool, optional Set this keyword to use the iterative sigma clipping method described in Section 2.1 of Cappellari et al. (2002). This is useful to remove from the fit unmasked bad pixels, residual gas emissions or cosmic rays.

IMPORTANT: This is recommended *only* if a reliable estimate of the `noise` spectrum is available. See also note below for `.chi2`.

degree: int, optional Degree of the *additive* Legendre polynomial used to correct the template continuum shape during the fit (default: 4). This uses the standard mathematical definition where e.g. `degree=2` is a quadratic polynomial. Set `degree=-1` not to include any additive polynomial.

fixed: Boolean vector set to `True` where a given kinematic parameter has to be held fixed with the value given in `start`. This is an array, or list, with the same dimensions as `start`.

EXAMPLE: We want to fit two kinematic components, with 4 moments for the first component and 2 for the second. In this case:

```
moments = [4, 2]
start = [[V1, sigma1, h3, h4], [V2, sigma2]]
```

then we can held fixed e.g. the sigma (only) of both components using:

```
fixed = [[0, 1, 0, 0], [0, 1]]
```

NOTE: Setting a negative `moments` for a kinematic component is entirely equivalent to setting `fixed = 1` for all parameters of the given kinematic component. In other words:

```
moments = [-4, 2]
```

is equivalent to:

```
moments = [4, 2]
fixed = [[1, 1, 1, 1], [0, 0]]
```

fraction: float, optional This keyword allows one to fix the ratio between the first two kinematic components. This is a scalar defined as follows:

```
fraction = np.sum(weights[component == 0])
         / np.sum(weights[component < 2])
```

This is useful e.g. to try to kinematically decompose bulge and disk.

The remaining kinematic components (`component > 1`) are left free, and this allows, for example, to still include gas emission line components. More general linear constraints, for multiple kinematic components at the same time, can be specified using the more general and flexible `constr_templ` keyword.

ftol: float, optional Fractional tolerance for stopping the non-linear minimization (default 1e-4).

gas_component: Boolean vector, of the same size as `component`, set to `True` where the given `component` describes a gas emission line. If given, `pPXF` provides the `pp.gas_flux` and `pp.gas_flux_error` in output.

EXAMPLE: In the common situation where `component = 0` are stellar templates and the rest are gas emission lines, one will set:

```
gas_component = component > 0
```

This keyword is also used to plot the gas lines with a different color.

gas_names: String array specifying the names of the emission lines (e.g. `gas_names=["Hbeta", "[OIII]",...]`, one per gas line. The length of this vector must match the number of nonzero elements in `gas_component`. This vector is only used by `pPXF` to print the line names on the console.

gas_reddening: float, optional Set this keyword to an initial estimate of the gas reddening `E(B-V) >= 0` to fit a positive gas reddening together with the kinematics and the templates. This reddening is applied only to the gas templates, namely to the templates with the corresponding element of `gas_component=True`. The typical use of this keyword is when using a single template for all the Balmer lines, with assumed intrinsic ratios for the lines. In this way the gas fit becomes sensitive to redening. The fit assumes by default the extinction curve of Calzetti et al. (2000) but any other prescription can be passed via the `reddening_func` keyword. By default `gas_reddening=None` and this parameter is not fitted.

global_search: bool or dictionary, optional Set to `True` to perform a global optimization of the nonlinear parameters (kinematics) before starting the usual local optimizer. Alternatively, one can pass via this keyword a dictionary of options for the function scipy.optimize.differential_evolution. Default options are `global_search={'tol': 0.1, 'disp': 1}`.

The `fixed` and `tied` keywords, as well as `constr_kinem` are properly supported when using `global_search` and one is encouraged to use them to reduce parameters degeneracies.

NOTE: This option is computationally intensive and completely unnecessary in most situations. It should *only* be used in special situations where there are obvious multiple local `chi2` minima. An example is when fitting multiple stellar or gas kinematic components with well-resolved velocity differences.

IMPORTANT: when using this keyword it is recommended *not* to use multiplicative polynomials but only additive ones to avoid unnecessarily long computation times. After converging to a global solution, if desired one can repeat the `pPXF` fit with multiplicative polynomials but without setting `global_search`.

goodpixels: array_like of int with shape (n_pixels,), optional Integer vector containing the indices of the good pixels in the `galaxy` spectrum (in increasing order). Only these spectral pixels are included in the fit.

lam: array_like with shape (n_pixels,), optional Vector with the *restframe* wavelength in Angstrom of every pixel in the input `galaxy` spectrum. This keyword is required when using the keyword `reddening` or `gas_reddening`.

If one uses my `ppxf_util.log_rebin` routine to rebin the spectrum before the `pPXF` fit, the wavelength can be obtained as `lam = np.exp(ln_lam)` below:

```
from ppxf.ppxf_util import log_rebin
specNew, ln_lam, velscale = log_rebin(lamRange, galaxy)
```

When `lam` is given, the wavelength is shown in the best-fitting plot, instead of the pixels.

lam_temp: array_like with shape (n_pixels_temp,), optional Vector with the *restframe* wavelength in Angstrom of every pixel in the input `templates` spectra.

When both the wavelength of the templates `lam_temp` and of the galaxy `lam` are given, the templates are automatically truncated to the minimal range required, for the adopted input velocity guess. In this case it is unnecessary to use the `vsyst` keyword.

If `phot` is also given, the final plot will include a best fitting spectrum estimated using the full `template`, before truncation, together with the photometric values and the truncated best fit to the `galaxy` spectrum. This is useful to see the underlying best fitting spectrum, in the wavelength range where only photometry (SED) was fitted.

linear: bool, optional Set to `True` to keep *all* nonlinear parameters fixed and *only* perform a linear fit for the templates and additive polynomials weights. The output solution is a copy of the input one and the errors are zero.

linear_method: {'nnls', 'lsq_box', 'lsq_lin', 'cvxopt'} optional Method used for the solution of the linear least-squares subproblem to fit for the templates weights (default 'lsq_box' fast box-constrained).

The computational speed of the four alternative linear methods depends on the size of the problem, with the default 'lsq_box' generally being the fastest without linear inequality constraints. Note that 'lsq_lin' is included in `ppxf`, while 'cvxopt' is an optional external package. The 'nnls' option (the only one before v7.0) is generally slower and for this reason is now deprecated.

The inequality constraints in `constr_templ` are only supported with `linear_method='lsq_lin'` or `linear_method='cvxopt'`.

mask: array_like of bool with shape (n_pixels,), optional Boolean vector of length `galaxy.size` specifying with `True` the pixels that should be included in the fit. This keyword is just an alternative way of specifying the `goodpixels`.

mdegree: int, optional Degree of the *multiplicative* Legendre polynomial (with a mean of 1) used to correct the continuum shape during the fit (default: 0). The zero degree multiplicative polynomial (i.e. constant) is always included in the fit as it corresponds to the multiplicative weights assigned to the templates. Note that the computation time is longer with multiplicative polynomials than with the same `degree` of additive polynomials.

method: {'capfit', 'trf', 'dogbox', 'lm'}, optional. Algorithm to perform the non-linear minimization step. The default 'capfit' is a novel linearly-constrained non-linear least-squares optimization program, which combines the Sequential Quadratic Programming and the Levenberg-Marquardt methods. For a description of the other methods ('trf', 'dogbox', 'lm'), see the documentation of scipy.optimize.least_squares.

The use of linear constraints with `constr_kinem` is only supported with the default `method='capfit'`.

moments: Order of the Gauss-Hermite moments to fit. Set this keyword to 4 to fit `[h3, h4]` and to 6 to fit `[h3, h4, h5, h6]`. Note that in all cases the G-H moments are fitted (non-linearly) *together* with `[V, sigma]`.

If `moments=2` or `moments` is not set then only `[V, sigma]` are fitted.

If `moments` is negative then the kinematics of the given `component` are kept fixed to the input values. NOTE: Setting a negative `moments` for a kinematic component is entirely equivalent to setting `fixed = 1` for all parameters of the given kinematic component.

EXAMPLE: We want to keep fixed `component = 0`, which has a LOSVD described by `[V, sigma, h3, h4]` and is modelled with 100 spectral templates; At the same time, we fit `[V, sigma]` for `component = 1`, which is described by 5 templates (this situation may arise when fitting stellar templates with pre-determined stellar kinematics, while fitting the gas emission). We should give in input to `pPXF` the following parameters:

```
component = [0]*100 + [1]*5   # --> [0, 0, ... 0, 1, 1, 1, 1, 1]
moments = [-4, 2]
start = [[V, sigma, h3, h4], [V, sigma]]
```

phot: dictionary, optional Dictionary of parameters used to fit photometric data (SED fitting) together with a spectrum. This is defined as follows:

```
phot = {"templates": phot_templates, "galaxy": phot_galaxy,
        "noise": phot_noise, "lam": phot_lam}
```

The keys of this dictionary are analogue to the `pPXF` parameters `galaxy`, `templates`, `noise` and `lam` for the spectra. However, the ones in this dictionary contain photometric data instead of spectra and will generally consist just a few values (one per photometric band) instead of thousands of elements like the spectra. Specifically:

- `phot_templates`: array_like with shape (n_phot, n_templates) -Mean flux of the templates in the observed photometric bands. This array has the same number of dimension as the `templates` input parameter. The same description applies. The only difference is that the first dimension is `n_phot` instead of `n_pixels_temp`. This array can have 2-4 dimensions and all dimensions must match those of the spectral `templates`, except for the first dimension. These templates must have the same units and normalization as the spectral `templates`. If the spectral templates cover the ranges of the photometric bands, and filter responses `resp` are available, the mean fluxes for each template can be computed as (e.g. equation A11 of Bessell & Murphy 2012):

  ```
  phot_template = Integrate[template*resp(lam)*lam, {lam, -inf, inf}]
                / Integrate[resp(lam)*lam, {lam, -inf, inf}]
  ```

One can use the function `ppxf_util.photometry_from_spectra` as an illustration of how to compute the `phot_templates`. This function can be easily modified to include any additional filter.

Alternatively, the fluxes may be tabulated by the authors of the SSP models, for the same model parameters as the spectral SSP templates. However, this can only be used for redshift `z ~ 0`.

- `phot_galaxy`: array_like with shape (n_phot) - Observed photometric measurements for the galaxy in linear flux units. These values must be matched to the same spatial aperture used for the spectra and they must have the same units (e.g. `erg/(s cm^2 A)`). This means that these values must be like the average fluxes one would measure on the fitted galaxy spectrum if it was sufficiently extended. One can think of these photometric values as some special extra pixels to be added to the spectrum. The difference is that they are not affected by the polynomials nor by the kinematics.
- `phot_noise`: array_like with shape (n_phot) -Vector containing the `1*sigma` uncertainty of each photometric measurement in `phot_galaxy`. One can change the normalization of these uncertainties to vary the relative influence of the photometric measurements versus the spectral fits.
- `phot_lam`: array_like with shape (n_phot) or (n_phot, n_templates)

  - Mean *restframe* wavelength for each photometric band in `phot_galaxy`. This is only used to estimate reddening of each band and to produce the plots. It can be computed from the system response function `resp` as (e.g. equation A17 of Bessell & Murphy 2012):

  ```
  phot_lam = Integrate[resp(lam)*lam^2, {lam, -inf, inf}]
           / Integrate[resp(lam)*lam, {lam, -inf, inf}]
  ```

  If spectral templates are available over the full extent of the photometric bands, then one can compute a more accurate effective wavelength for each template separately. In this case `phot_lam` must have the same dimensions as `phot_templates`. For each templates the effective wavelength can be computed as (e.g. equation A21 of Bessell & Murphy 2012):

  ```
  phot_lam = Integrate[template*resp(lam)*lam^2, {lam, -inf, inf}]
           / Integrate[template*resp(lam)*lam, {lam, -inf, inf}]
  ```

plot: bool, optional Set this keyword to plot the best fitting solution and the residuals at the end of the fit.

One can also call separately the class function `pp.plot()` after the call to `pp = ppxf(...)`.

quiet: bool, optional Set this keyword to suppress verbose output of the best fitting parameters at the end of the fit.

reddening: float, optional Set this keyword to an initial estimate of the stellar reddening `E(B-V) >= 0` to fit a positive stellar reddening together with the kinematics and the templates. This reddening is applied only to the stellar templates (both spectral and photometric ones), namely to the templates with the corresponding element of `gas_component=False`, or to all templates, if `gas_component` is not set. The fit assumes by default the extinction curve of Calzetti et al. (2000) but any other prescription can be passed via the `reddening_func` keyword. By default `reddening=None` and this parameter is not fitted.

regul: float, optional If this keyword is nonzero, the program applies first or second-order linear regularization to the `weights` during the `pPXF` fit. Regularization is done in one, two or three dimensions depending on whether the array of `templates` has two, three or four dimensions respectively. Large `regul` values correspond to smoother `weights` output. When this keyword is nonzero the solution will be a trade-off between the smoothness of `weights` and goodness of fit.

Section 3.5 of Cappellari (2017) gives a description of regularization.

When fitting multiple kinematic `component` the regularization is applied only to the first `component = 0`, while additional components are not regularized. This is useful when fitting stellar population together with gas emission lines. In that case, the SSP spectral templates must be given first and the gas emission templates are given last. In this situation, one has to use the `reg_dim` keyword (below), to give `pPXF` the dimensions of the population parameters (e.g. `n_age`, `n_metal`, `n_alpha`). A usage example is given in the file `ppxf_example_population_gas_sdss.py`.

The effect of the regularization scheme is the following:

- With `reg_ord=1` it enforces the numerical first derivatives between neighbouring weights (in the 1-dim case) to be equal to `w[j] - w[j+1] = 0` with an error `Delta = 1/regul`.
- With `reg_ord=2` it enforces the numerical second derivatives between neighboring weights (in the 1-dim case) to be equal to `w[j-1] - 2*w[j] + w[j+1] = 0` with an error `Delta = 1/regul`.

It may be helpful to define `regul = 1/Delta` and think of `Delta` as the regularization error.

IMPORTANT: `Delta` needs to be smaller but of the same order of magnitude of the typical `weights` to play an effect on the regularization. One quick way to achieve this is:

1. Divide the full `templates` array by a scalar in such a way that the typical template has a median of one:

   ```
   templates /= np.median(templates)
   ```

2. Do the same for the input galaxy spectrum:

   ```
   galaxy /= np.median(galaxy)
   ```

3. In this situation, a sensible guess for `Delta` will be a few percent (e.g. `Delta=0.01 --> regul=100`).

Alternatively, for a more rigorous definition of the parameter `regul`:

A. Perform an un-regularized fit (`regul=0`) and then rescale the input `noise` spectrum so that:

   ```
   Chi^2/DOF = Chi^2/goodPixels.size = 1.
   ```

   This is achieved by rescaling the input `noise` spectrum as:

   ```
   noise = noise*np.sqrt(Chi**2/DOF) = noise*np.sqrt(pp.chi2);
   ```

B. Increase `regul` and iteratively redo the `pPXF` fit until the `Chi^2` increases from the unregularized value `Chi^2 = goodPixels.size` by `DeltaChi^2 = np.sqrt(2*goodPixels.size)`.

The derived regularization corresponds to the maximum one still consistent with the observations and the derived star formation history will be the smoothest (minimum curvature or minimum variation) that is still consistent with the observations.

reg_dim: tuple, optional When using regularization with more than one kinematic component (using the `component` keyword), the regularization is only applied to the first one (`component=0`). This is useful to fit the stellar population and gas emission together.

In this situation, one has to use the `reg_dim` keyword, to give `pPXF` the dimensions of the population parameters (e.g. `n_age`, `n_metal`, `n_alpha`). One should creates the initial array of population templates like e.g. `templates[n_pixels, n_age, n_metal, n_alpha]` and define:

```
reg_dim = templates.shape[1:]   # = [n_age, n_metal, n_alpha]
```

The array of stellar templates is then reshaped into a 2-dim array as:

```
templates = templates.reshape(templates.shape[0], -1)
```

and the gas emission templates are appended as extra columns at the end. An usage example is given in `ppxf_example_population_gas_sdss.py`.

When using regularization with a single component (the `component` keyword is not used, or contains identical values), the number of population templates along different dimensions (e.g. `n_age`, `n_metal`, `n_alpha`) is inferred from the dimensions of the `templates` array and this keyword is not necessary.

reg_ord: int, optional Order of the derivative that is minimized by the regularization. The following two rotationally-symmetric estimators are supported:

- `reg_ord=1`: minimizes the integral over the weights of the squared gradient:

  `Grad[w] @ Grad[w].`

- `reg_ord=2`: minimizes the integral over the weights of the squared curvature:

  `Laplacian[w]**2.`

sigma_diff: float, optional Quadratic difference in km/s defined as:

```
sigma_diff**2 = sigma_inst**2 - sigma_temp**2
```

between the instrumental dispersion of the galaxy spectrum and the instrumental dispersion of the template spectra.

This keyword is useful when the templates have higher resolution than the galaxy and they were not convolved to match the instrumental dispersion of the galaxy spectrum. In this situation, the convolution is done by `pPXF` with increased accuracy, using an analytic Fourier Transform.

sky: vector containing the spectrum of the sky to be included in the fit, or array of dimensions `sky[n_pixels, nSky]` containing different sky spectra to add to the model of the observed `galaxy` spectrum. The `sky` has to be log-rebinned as the `galaxy` spectrum and needs to have the same number of pixels.

The sky is generally subtracted from the data before the `pPXF` fit. However, for observations very heavily dominated by the sky spectrum, where a very accurate sky subtraction is critical, it may be useful *not* to subtract the sky from the spectrum, but to include it in the fit using this keyword.

templates_rfft: When calling `pPXF` many times with an identical set of templates, one can use this keyword to pass the real FFT of the templates, computed in a previous `pPXF` call, stored in the `pp.templates_rfft` attribute. This keyword mainly exists to show that there is no need for it...

IMPORTANT: Use this keyword only if you understand what you are doing!

tied: A list of string expressions. Each expression "ties" the parameter to other free or fixed parameters. Any expression involving constants and the parameter array `p[j]` are permitted. Since they are totally constrained, tied parameters are considered to be fixed; no errors are computed for them.

This is an array, or list of arrays, with the same dimensions as `start`. In practice, for every element of `start` one needs to specify either an empty string `''` implying that the parameter is free, or a string expression involving some of the variables `p[j]`, where `j` represents the index of the flattened list of kinematic parameters.

EXAMPLE: We want to fit three kinematic components, with 4 moments for the first component and 2 moments for the second and third (e.g. stars and two gas components). In this case:

```
moments = [4, 2, 2]
start = [[V1, sigma1, 0, 0], [V2, sigma2], [V3, sigma3]]
```

then we can force the equality constraint `V2 = V3` as follows:

```
tied = [['', '', '', ''], ['', ''], ['p[4]', '']]  # p[6] = p[4]
```

or we can force the equality constraint `sigma2 = sigma3` as follows:

```
tied = [['', '', '', ''], ['', ''], ['', 'p[5]']]  # p[7] = p[5]
```

One can also use more general formulas. For example one could constrain `V3 = (V1 + V2)/2` as well as `sigma1 = sigma2` as follows:

```
# p[5] = p[1]
# p[6] = (p[0] + p[4])/2
tied = [['', '', '', ''], ['', 'p[1]'], ['(p[0] + p[4])/2', '']]
```

NOTE: One could in principle use the `tied` keyword to completely tie the LOSVD of two kinematic components. However, this same effect is more efficient achieved by assigning them to the same kinematic component using the `component` keyword.

trig: Set `trig=True` to use trigonometric series as an alternative to Legendre polynomials, for both the additive and multiplicative polynomials. When `trig=True` the fitted series below has `N = degree/2` or `N = mdegree/2`:

```
poly = A_0 + sum_{n=1}^{N} [A_n*cos(n*th) + B_n*sin(n*th)]
```

IMPORTANT: The trigonometric series has periodic boundary conditions. This is sometimes a desirable property, but this expansion is not as flexible as the Legendre polynomials.

velscale_ratio: int, optional Integer. Gives the integer `ratio >= 1` between the `velscale` of the `galaxy` and the `templates`. When this keyword is used, the templates are convolved by the LOSVD at their native resolution, and only subsequently are integrated over the pixels and fitted to `galaxy`. This keyword is generally unnecessary and mostly useful for testing.

Note that in realistic situations the uncertainty in the knowledge and variations of the intrinsic line-spread function becomes the limiting factor in recovering the LOSVD well below `velscale`.

vsyst: float, optional Reference velocity in `km/s` (default 0). The input initial guess and the output velocities are measured with respect to this velocity. This keyword can be used to account for the difference in the starting wavelength of the templates and the galaxy spectrum as follows:

```
vsyst = c*np.log(wave_temp[0]/wave_gal[0])
```

As alternative to using this keyword, one can pass the wavelengths `lam` and `lam_temp` of both the `galaxy` and `templates` spectra. In that case `vsyst` is computed automatically and should not be given.

The value assigned to this keyword is *crucial* for the two-sided fitting. In this case `vsyst` can be determined from a previous normal one-sided fit to the galaxy velocity profile. After that initial fit, `vsyst` can be defined as the measured velocity at the galaxy center. More accurately `vsyst` is the value which has to be subtracted to obtain a nearly anti-symmetric velocity profile at the two opposite sides of the galaxy nucleus.

IMPORTANT: this value is generally *different* from the systemic velocity one can get from the literature. Do not try to use that!

Output Parameters

Stored as attributes of the `pPXF` class:

.apoly: Vector with the best fitting additive polynomial.

.bestfit: Vector with the best fitting model for the galaxy spectrum. This is a linear combination of the templates, convolved with the best fitting LOSVD, multiplied by the multiplicative polynomials and with subsequently added polynomial continuum terms or sky components.

.chi2: The reduced `chi^2` (namely `chi^2/DOF`) of the fit, where `DOF = pp.dof` (approximately `DOF ~ pp.goodpixels.size`).

IMPORTANT: if `Chi^2/DOF` is not ~1 it means that the errors are not properly estimated, or that the template is bad and it is *not* safe to set the `clean` keyword.

.error: This variable contains a vector of *formal* uncertainty (`1*sigma`) for the fitted parameters in the output vector `sol`. They are computed from the estimated covariance matrix of the standard errors in the fitted parameters assuming it is diagonal at the minimum. This option can be used when speed is essential, to obtain an order of magnitude estimate of the uncertainties, but we *strongly* recommend to run bootstrapping simulations to obtain more reliable errors. In fact, these errors can be severely underestimated in the region where the penalty effect is most important (`sigma < 2*velscale`).

These errors are meaningless unless `Chi^2/DOF ~ 1`. However if one *assumes* that the fit is good, a corrected estimate of the errors is:

`error_corr = error*sqrt(chi^2/DOF) = pp.error*sqrt(pp.chi2).`

IMPORTANT: when running Monte Carlo simulations to determine the error, the penalty (`bias`) should be set to zero, or better to a very small value. See Section 3.4 of Cappellari & Emsellem (2004) for an explanation.

.gas_bestfit: If `gas_component is not None`, this attribute returns the best-fitting gas emission-lines spectrum alone. The best-fitting stellar spectrum alone can be computed as `stars_bestfit = pp.bestfit - pp.gas_bestfit`

.gas_bestfit_templates: If `gas_component is not None`, this attribute returns the individual best-fitting gas emission-lines templates as columns of an array. Note that `pp.gas_bestfit = pp.gas_bestfit_templates.sum(1)`

.gas_flux: Vector with the integrated flux (in counts) of all lines set as `True` in the input `gas_component` keyword. This is the flux of individual gas templates, which may include multiple lines. This implies that, if a gas template describes a doublet, the flux is that of both lines. If the Balmer series is input as a single template, this is the flux of the entire series.

The returned fluxes are not corrected in any way and in particular, no reddening correction is applied. In other words, the returned `.gas_flux` should be unchanged, within the errors, regardless of whether reddening or multiplicative polynomials were fitted by `pPXF` or not.

IMPORTANT: `pPXF` makes no assumptions about the input flux units: The returned `.gas_flux` has the same units and values one would measure (with lower accuracy) by summing the pixels values, within the given gas lines, on the continuum-subtracted input galaxy spectrum. This implies that, if the spectrum is in units of `erg/(s cm^2 A)`, the `.gas_flux` returned by `pPXF` should be multiplied by the pixel size in Angstrom at the line wavelength to obtain the integrated line flux in units of `erg/(s cm^2)`.

NOTE: If there is no gas reddening and each input gas templates was normalized to `sum = 1`, then `pp.gas_flux = pp.weights[pp.gas_component]`.

When a gas template is identically zero within the fitted region, then `pp.gas_flux = pp.gas_flux_error = np.nan`. The corresponding components of `pp.gas_zero_template` are set to `True`. These `np.nan` values are set at the end of the calculation to flag the undefined values. These flags generally indicate that some of the gas templates passed to `pPXF` consist of gas emission lines that fall outside the fitted wavelength range or within a masked spectral region. These `np.nan` do *not* indicate numerical issues with the actual `pPXF` calculation and the rest of the `pPXF` output is reliable.

.gas_flux_error: *Formal* uncertainty (`1*sigma`) for the quantity `pp.gas_flux`, in the same units as the gas fluxes.

This error is approximate as it ignores the covariance between the gas flux and any non-linear parameter. Bootstrapping can be used for more accurate errors.

These errors are meaningless unless `Chi^2/DOF ~ 1`. However if one *assumes* that the fit is good, a corrected estimate of the errors is:

`gas_flux_error_corr = gas_flux_error*sqrt(chi^2/DOF)`

= pp.gas_flux_error*sqrt(pp.chi2).

.gas_mpoly: vector with the best-fitting gas reddening curve.

.gas_reddening: Best fitting `E(B-V)` value if the `gas_reddening` keyword is set. This is especially useful when the Balmer series is input as a single template with an assumed theoretically predicted decrement e.g. using `emission_lines(..., tie_balmer=True)` in `ppxf.ppxf_util` to compute the gas templates.

.gas_zero_template: vector of size `gas_component.sum()` set to `True` where the gas template was identically zero within the fitted region. For those gas components `pp.gas_flux = pp.gas_flux_error = np.nan`. These flags generally indicate that some of the gas templates passed to `pPXF` consist of gas emission lines that fall outside the fitted wavelength range or within a masked spectral region.

.goodpixels: Integer vector containing the indices of the good pixels in the fit. This vector is a copy of the input `goodpixels` if `clean = False` otherwise it will be updated by removing the detected outliers.

.matrix: Prediction `matrix[n_pixels, degree+n_templates]` of the linear system.

`pp.matrix[n_pixels, :degree]` contains the additive polynomials if `degree >= 0`.

`pp.matrix[n_pixels, degree:]` contains the templates convolved by the LOSVD, and multiplied by the multiplicative polynomials if `mdegree > 0`.

.mpoly: Best fitting multiplicative polynomial (or reddening curve when `reddening` is set).

.mpolyweights: This is largely superseded by the `.mpoly` attribute above.

When `mdegree > 0` this contains in output the coefficients of the multiplicative Legendre polynomials of order `1, 2,... mdegree`. The polynomial can be explicitly evaluated as:

```
from numpy.polynomial import legendre
x = np.linspace(-1, 1, len(galaxy))
mpoly = legendre.legval(x, np.append(1, pp.mpolyweights))
```

When `trig = True` the polynomial is evaluated as:

```
mpoly = pp.trigval(x, np.append(1, pp.mpolyweights))
```

.phot_bestfit: array_like with shape (n_phot) When `phot` is given, then this attribute contains the best fitting fluxes in the photometric bands given as input in `phot_galaxy`.

.plot: function Call the method function `pp.plot()` after the call to `pp = ppxf(...)` to produce a plot of the best fit. This is an alternative to calling `pp = ppxf(..., plot=True)`.

Use the command `pp.plot(gas_clip=True)` to scale the plot based on the stellar continuum alone, while allowing for the gas emission lines to go outside the plotting region. This is useful to inspect the fit to the stellar continuum, in the presence of strong gas emission lines. This has effect only if `gas_component is not None`.

.polyweights: This is largely superseded by the `.apoly` attribute above.

When `degree >= 0` contains the weights of the additive Legendre polynomials of order `0, 1,... degree`. The best fitting additive polynomial can be explicitly evaluated as:

```
from numpy.polynomial import legendre
```

```
x = np.linspace(-1, 1, len(galaxy))
apoly = legendre.legval(x, pp.polyweights)
```

When `trig=True` the polynomial is evaluated as:

```
apoly = pp.trigval(x, pp.polyweights)
```

When doing a two-sided fitting (see help for `galaxy` parameter), the additive polynomials are allowed to be different for the left and right spectrum. In that case, the output weights of the additive polynomials alternate between the first (left) spectrum and the second (right) spectrum.

.reddening: Best fitting `E(B-V)` value if the `reddening` keyword is set.

.sol: Vector containing in output the parameters of the kinematics.

- If `moments=2` this contains `[Vel, Sigma]`
- If `moments=4` this contains `[Vel, Sigma, h3, h4]`
- If `moments=N` this contains `[Vel, Sigma, h3,... hN]`

When fitting multiple kinematic `component`, `pp.sol` contains a list with the solution for all different components, one after the other, sorted by `component`: `pp.sol = [sol1, sol2,...]`.

`Vel` is the velocity, `Sigma` is the velocity dispersion, `h3 - h6` are the Gauss-Hermite coefficients. The model parameters are fitted simultaneously.

IMPORTANT: The precise relation between the output `pPXF` velocity and redshift is `Vel = c*np.log(1 + z)`. See Section 2.3 of Cappellari (2017) for a detailed explanation.

These are the default safety limits on the fitting parameters (they can be changed using the `bounds` keyword):

- `Vel` is constrained to be `+/-2000` km/s from the input guess
- `velscale/100 < Sigma < 1000` km/s
- `-0.3 < [h3, h4, ...] < 0.3` (extreme value for real galaxies)

In the case of two-sided LOSVD fitting the output values refer to the first input galaxy spectrum, while the second spectrum will have by construction kinematics parameters `[-Vel, Sigma, -h3, h4, -h5, h6]`. If `vsyst` is nonzero (as required for two-sided fitting), then the output velocity is measured with respect to `vsyst`.

.status: Contains the output status of the optimization. Positive values generally represent success (the meaning of `status` is defined as in scipy.optimize.least_squares).

.weights: Receives the value of the weights by which each template was multiplied to best fit the galaxy spectrum. The optimal template can be computed with an array-vector multiplication:

```
bestemp = templates @ weights
```

These `.weights` do not include the weights of the additive polynomials which are separately stored in `pp.polyweights`.

When the `sky` keyword is used `weights[:n_templates]` contains the weights for the templates, while `weights[n_templates:]` gives the ones for the sky. In that case the best fitting galaxy template and sky are given by:

```
bestemp = templates @ weights[:n_templates]
bestsky = sky @ weights[n_templates:]
```

When doing a two-sided fitting (see help for `galaxy` parameter) *together* with the `sky` keyword, the sky weights are allowed to be different for the left and right spectrum. In that case the output sky weights alternate between the first (left) spectrum and the second (right) spectrum.

How to Set the Kinematic Penalty Keyword

The `bias` keyword is only used if `moments > 2`, otherwise it is ignored.

The `pPXF` routine can give sensible quick results with the default `bias` parameter, however, like in any penalized/filtered/regularized method, the optimal amount of penalization generally depends on the problem under study.

The general rule here is that the penalty should leave the line-of-sight velocity-distribution (LOSVD) virtually unaffected, when it is well sampled and the signal-to-noise ratio (`S/N`) is sufficiently high.

EXAMPLE: If you expect a LOSVD with up to a high `h4 ~ 0.2` and your adopted penalty (`bias`) biases the solution towards a much lower `h4 ~ 0.1`, even when the measured `sigma > 3*velscale` and the S/N is high, then you are *misusing* the `pPXF` method!

THE RECIPE: The following is a simple practical recipe for a sensible determination of the penalty in `pPXF`:

1. Choose a minimum `(S/N)_min` level for your kinematics extraction and spatially bin your data so that there are no spectra below `(S/N)_min`;
2. Perform a fit of your kinematics *without* penalty (keyword `bias=0`). The solution will be noisy and may be affected by spurious solutions, however, this step will allow you to check the expected mean ranges in the Gauss-Hermite parameters `[h3, h4]` for the galaxy under study;
3. Perform a Monte Carlo simulation of your spectra, following e.g. the included `ppxf_example_montecarlo_simulation.py` routine. Adopt as `S/N` in the simulation the chosen value `(S/N)_min` and as input `[h3, h4]` the maximum representative values measured in the non-penalized `pPXF` fit of the previous step;
4. Choose as the penalty (`bias`) the *largest* value such that, for `sigma > 3*velscale`, the mean difference delta between the output `[h3, h4]` and the input `[h3, h4]` is well within (e.g. `delta ~ rms/3`) the rms scatter of the simulated values (see an example in Fig. 2 of Emsellem et al. 2004).

Problems with Your First Fit?

Common problems with your first `pPXF` fit are caused by incorrect wavelength ranges or different velocity scales between galaxy and templates. To quickly detect these problems try to overplot the (log rebinned) galaxy and the template just before calling the `pPXF` procedure.

You can use something like the following Python lines while adjusting the smoothing window and the pixels shift. If you cannot get a rough match by eye it means something is wrong and it is unlikely that `pPXF` (or any other program) will find a good match:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage

sigma = 2       # Velocity dispersion in pixels
shift = -20     # Velocity shift in pixels
template = np.roll(ndimage.gaussian_filter1d(template, sigma), shift)
plt.plot(galaxy, 'k')
plt.plot(template*np.median(galaxy)/np.median(template), 'r')
```
