

User guide for 3-way DD-SIMCA methods

Sergey Kucheryavskiy

June 21, 2026

1 Introduction

From v. 1.1.0 the `ddsimca` package has been extended with `ddsimca_parafac` and `ddsimca_tucker` functions which let you apply the DD-SIMCA approach to 3-way data. More details about both methods can be found in [10.1021/acs.analchem.3c05096](https://doi.org/10.1021/acs.analchem.3c05096).

If you have not used this package before, it is recommended to learn the conventional `ddsimca()` method by following this tutorial: [demo.ipynb](#), as a lot of functionality of the 3-way methods is similar to the conventional DD-SIMCA implementation. This tutorial assumes that you are already familiar with `ddsimca()` and know the basics of PARAFAC.

2 Preparing the data

Although 3-way datasets are stored as tensors (three-dimensional arrays), in order to simplify the workflow both `ddsimca_parafac()` and `ddsimca_tucker()` methods work with Pandas data frames, similar to the conventional `ddsimca()`. The first column of the data frame should contain the class labels in case of training and test set. The other columns contain the unfolded data values (e.g. excitation-emission profiles or GC-MS values unfolded to a vector of values).

If you have your dataset stored as a tensor, simply unfold it to matrix, so the rows of the matrix correspond to the samples and the columns — to the row-wise unfolded data. You can save such representation into CSV file. The only important thing is that you need to know (and provide) the original dimensions when training the model.

This tutorial is based on simulated 3-way data, the simulation was done similar to what is described in [10.1021/acs.analchem.3c05096](https://doi.org/10.1021/acs.analchem.3c05096). Target samples contain two components in random concentrations. Alternative class samples contain the same two components + a third component in random concentrations. The component instrumental profiles are partially overlapped Gaussian functions, defined in 40 and 50 channels in each data mode (hence the data matrices have 50×40 dimension). Gaussian noise was then added to all signals at a level of 10% with respect to the mean signal. The author is grateful to Alejandro Olivieri for sharing a MATLAB script implementing simulations and helping with the tutorial.

You can download the file with the datasets from [simdata.mat](#). The data was simulated in MATLAB and hence is available as a MAT file (MATLAB data file), which we will manually convert to data frames.

Let's load the data first. We will use package `scipy` for this purpose, you need to install it if you do not have it yet.

```
[2]: import scipy.io
data = scipy.io.loadmat('simdata.mat')

X3w = data['X3w']    # training set
X3wt = data['X3wt']  # test set (target)
X3wa = data['X3wa']  # test set (alternative)

(X3w.shape, X3wt.shape, X3wa.shape)
```

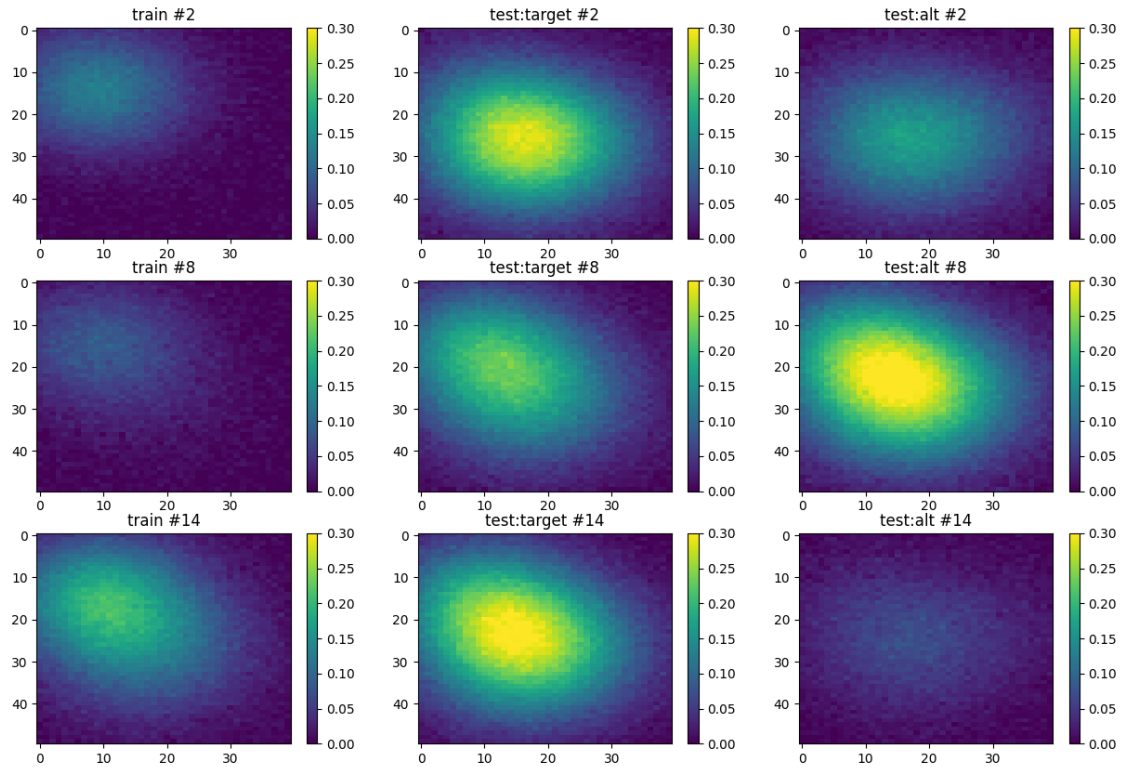
```
[2]: ((80, 50, 40), (20, 50, 40), (60, 50, 40))
```

As we can see, all datasets are represented in the form of 3-way arrays, the training set contains 80 samples, the test target set contains 20 samples and the test alternative set contains 60 samples. The dimension of each sample data is 50 x 40.

Let's visualize some of them in the form of heatmaps.

```
[3]: import matplotlib.pyplot as plt

set = {"train": X3w, "test:target": X3wt, "test:alt": X3wa}
plt.figure(figsize = (15, 10))
n = 1
for i in [1, 7, 13]:
    for name, x in set.items():
        plt.subplot(3, 3, n)
        plt.imshow(x[i, :, :], aspect = "auto", clim = (0, 0.3))
        plt.title(f"{name} #{i + 1}")
        plt.colorbar()
        n = n + 1
```



Now let's prepare the data frames as follows:

1. Reshape the 3-way arrays into 2-way arrays (matrices).
2. Create a data frame from the matrices.
3. Add a column "classname" with class labels.

Here is the code:

```
[4]: import pandas as pd

# train: target
ns, nr, nc = X3w.shape
data_train = pd.DataFrame(X3w.reshape(ns, nr * nc))
data_train.insert(0, "classname", ["target"] * ns)

# test: target
ns, nr, nc = X3wt.shape
data_test_target = pd.DataFrame(X3wt.reshape(ns, nr * nc))
data_test_target.insert(0, "classname", ["target"] * ns)

# test: alt
ns, nr, nc = X3wa.shape
data_test_alt = pd.DataFrame(X3wa.reshape(ns, nr * nc))
data_test_alt.insert(0, "classname", ["alt"] * ns)
```

```
# combine both target and alt together into a single test set
data_test = pd.concat([data_test_target, data_test_alt])
data_test
```

```
[4]:  classname      0      1  ...    1997    1998    1999
0    target  0.027985  0.031098  ...  0.008460  0.012077  0.013859
1    target  0.005608  0.023140  ...  0.021453  0.015444  0.005071
2    target  0.034567  0.038592  ...  0.002163 -0.004820 -0.001560
3    target  0.038813  0.029240  ...  0.003637  0.000404  0.002375
4    target  0.023743  0.026105  ... -0.006154 -0.003769 -0.007631
..    ...    ...    ...    ...    ...    ...
55   alt -0.001382  0.006688  ...  0.000915  0.008008 -0.000891
56   alt  0.031243  0.034019  ...  0.000632 -0.008506  0.009325
57   alt  0.026348  0.024962  ...  0.004145  0.004989 -0.001512
58   alt -0.001668  0.009869  ...  0.006640  0.001368  0.005848
59   alt  0.039432  0.036929  ... -0.005290  0.007036  0.008432
```

[80 rows x 2001 columns]

As you can see there are 2001 columns in total, one column contains the class labels and the others — unfolded data values. We know that the original dimension of the test set was $80 \times 50 \times 40$, so the unfolded matrix will have a dimension of 80×2000 , which is in line with what we observed above.

3 DD-SIMCA PARAFAC

Training a DD-SIMCA PARAFAC model is similar to training a conventional DD-SIMCA model, with one additional argument — the dimension of the sample matrix (in our case 50×40):

```
[5]: from ddsimca import ddsimca_parafac

m = ddsimca_parafac(data_train, dim = (50, 40), ncomp = 4)
m.summary()
```

DDSIMCA-PARAFAC model:

- target class: target
- number of components (total): 4
- number of components (optimal): 4
- number of training samples: 80
- number of variables: 2000 (50 x 40)

Parameters for classic estimators:

Comp	Nh	Nq
1	2	3
2	4	250

3	3	250
4	4	250

There is an important difference though. In the case of conventional DD-SIMCA training, providing `ncomp = 4` will result in a model based on a single PCA decomposition (a single PCA model) with 4 components. In the case of PARAFAC, it will train four different models: the first PARAFAC model will use 1 component (1 factor), the second model — 2 components (factors), and so on.

In our case, we know that 2 components are required to explain the variation in the simulated data. However, usually this information is not available in advance, so the DD-SIMCA model above is trained with four PARAFAC sub-models with the number of components (factors) ranging from 1 to 4.

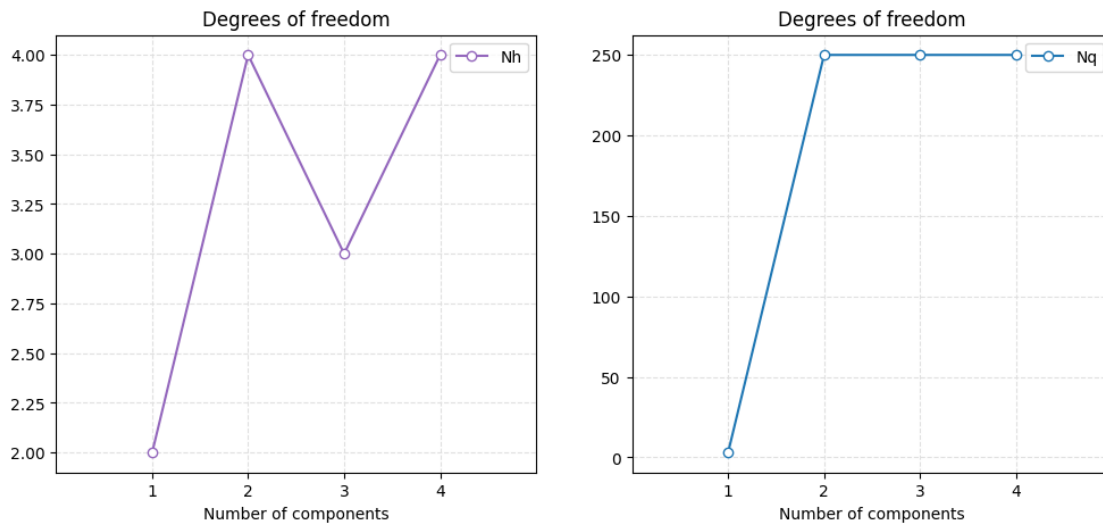
By default, the last model (with 4 components in our case) is considered optimal. You can change this in several ways, which we will discuss below.

Let's look at the number of degrees of freedom first.

```
[6]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
m.plotDoF(ax1, dof = "Nh")

ax2 = plt.subplot(1, 2, 2)
m.plotDoF(ax2, dof = "Nq")
```



Again it is important to underline that both N_h and N_q values are related to different PARAFAC models. Thus one can see that for the PARAFAC model with 2 components/factors, N_q is 250. It is the largest possible value because 2 components explain all systematic variation in the training data, so there is nothing left for the residual distance. Any model with number of factors > 2 will demonstrate similar behaviour.

Like in conventional DD-SIMCA, the DD-SIMCA PARAFAC model object does not have any results, it only contains values and statistics needed for applying this model to any dataset (e.g. PARAFAC factors B and C, parameters of distance distribution, etc.). If you want to visualize the PARAFAC factors you can use the following:

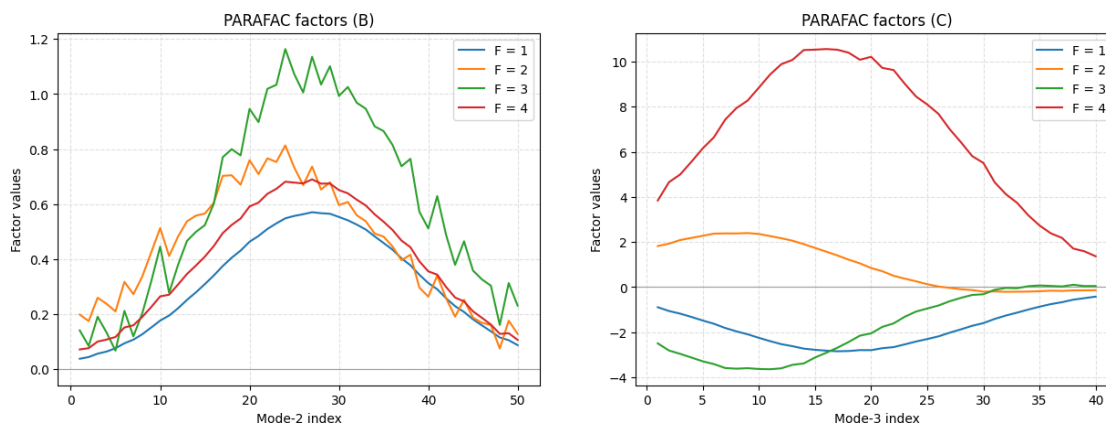
```
[7]: import matplotlib.colors as mcolors

# get a list with four distinct colors from the TABLEAU colormap
cols = list(mcolors.TABLEAU_COLORS.values())[0:4]

plt.figure(figsize = (15, 5))

# factors B
ax1 = plt.subplot(1, 2, 1)
for f in range(1, 5):
    m.plotFactors(ax1, f, mode = "B", color = cols[f - 1], label = f"F = {f}")
ax1.legend()

# factors C
ax2 = plt.subplot(1, 2, 2)
for f in range(1, 5):
    m.plotFactors(ax2, f, mode = "C", color = cols[f - 1], label = f"F = {f}")
ax2.legend();
```



Apparently the plots do not look convincing. Let's now tell the model that the optimal number of components is 2 and show factors for the PARAFAC model with 2 components/factors:

```
[8]: # change number of optimal components to 2
m.select_ncomp(2)

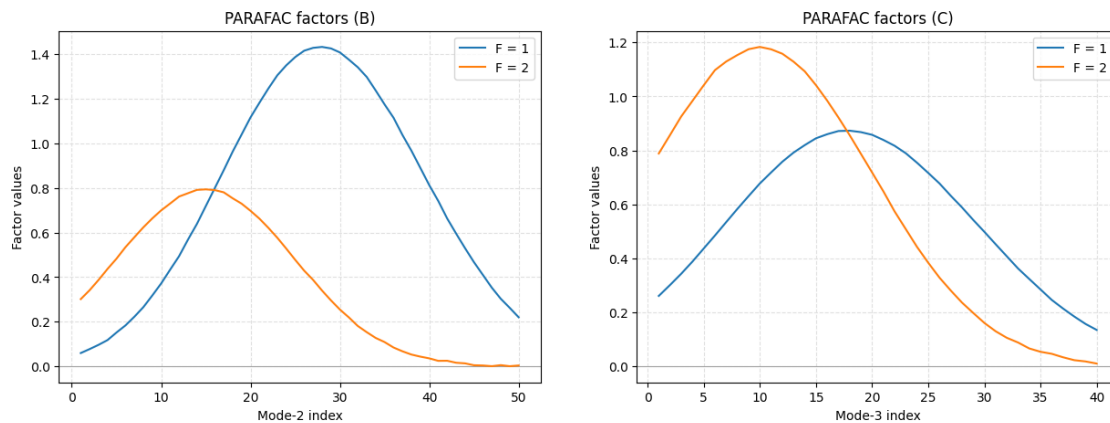
plt.figure(figsize = (15, 5))
```

```

# factors B
ax1 = plt.subplot(1, 2, 1)
for f in range(1, 3):
    m.plotFactors(ax1, f, mode = "B", color = cols[f - 1], label = f"F = {f}")
ax1.legend()

# factors C
ax2 = plt.subplot(1, 2, 2)
for f in range(1, 3):
    m.plotFactors(ax2, f, mode = "C", color = cols[f - 1], label = f"F = {f}")
ax2.legend();

```



Now we indeed see two partially overlapped Gaussian curves used for simulation of the data.

Like in conventional DD-SIMCA, to get the results, you need to apply this model to a dataset. Here is how to do it for the training set:

```

[9]: r_train = m.predict(data_train)
     r_train.summary()

```

DDSIMCA-PARAFAC results:

- number of components (total): 4
- number of components (selected): 2
- limit type: classic
- alpha: 0.050
- gamma: 0.010

- class labels: provided
- number of objects: 80
- number of members: 80
- number of strangers: 0

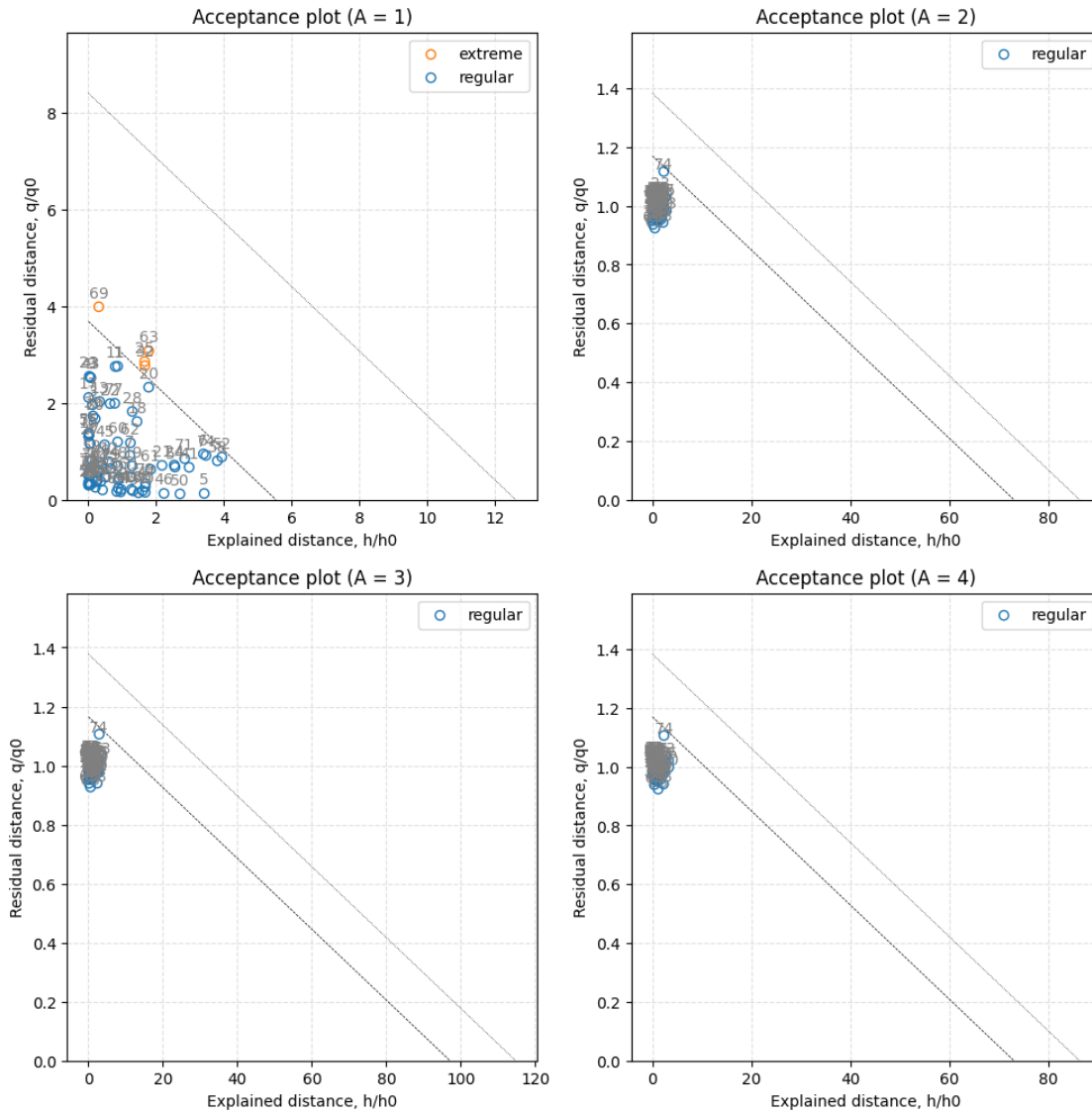
PCs	eCrit	oCrit	in	out	TP	FN	sens
1	11.070	25.233	76	4	76	4	0.95
2	292.488	345.679	80	0	80	0	1.00
3	291.336	344.319	80	0	80	0	1.00
4	292.488	345.679	80	0	80	0	1.00

As you can see, most of the parameters — like the significance levels for extremes (**alpha**) and outliers (**gamma**), and the type of distance-limit estimator (**lim_type**) — are set to default values (0.05, 0.01, and "classic" respectively). If you want to change any of them, simply provide the proper values as arguments of the method **predict()**. These are identical to how you do it for conventional DD-SIMCA, so we skip the details here.

Now let's check the acceptance plot for the training set using different models. Although the model now knows that the decomposition with 2 components is optimal and should be used as default, the distance values and the acceptance limits are still computed for all models, so we can show the Acceptance plot for models with different number of components/factors.

```
[10]: plt.figure(figsize = (12, 12))

for A in [1, 2, 3, 4]:
    ax = plt.subplot(2, 2, A)
    r_train.plotAcceptance(ax, ncomp = A, show_labels = True)
```

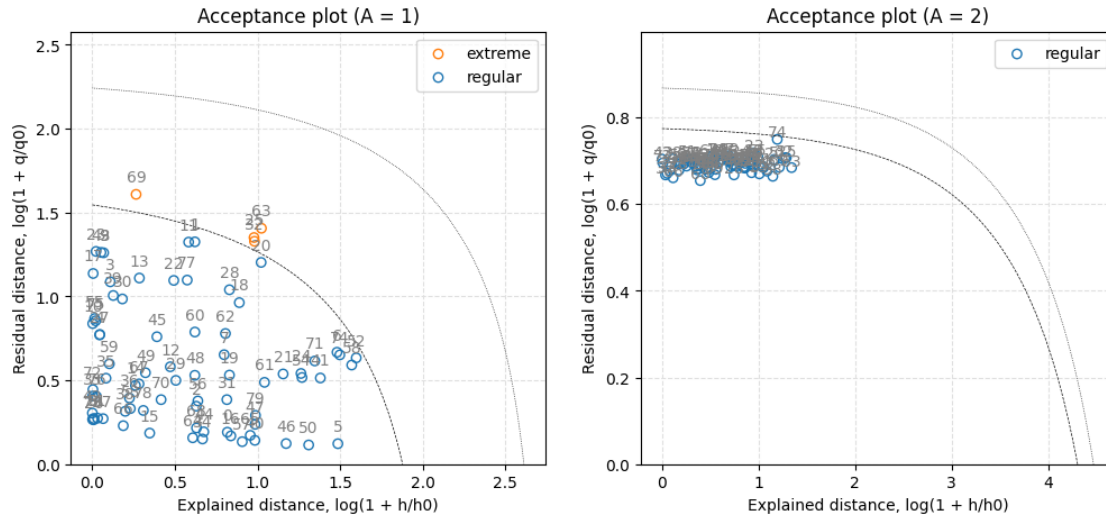
As one can see, starting from $A = 2$ only the score (explained) distance varies, as expected.

We can also show these two plots using log-transformed coordinates:

```
[11]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
r_train.plotAcceptance(ax1, ncomp = 1, do_log = True, show_labels = True)

ax2 = plt.subplot(1, 2, 2)
r_train.plotAcceptance(ax2, ncomp = 2, do_log = True, show_labels = True)
```

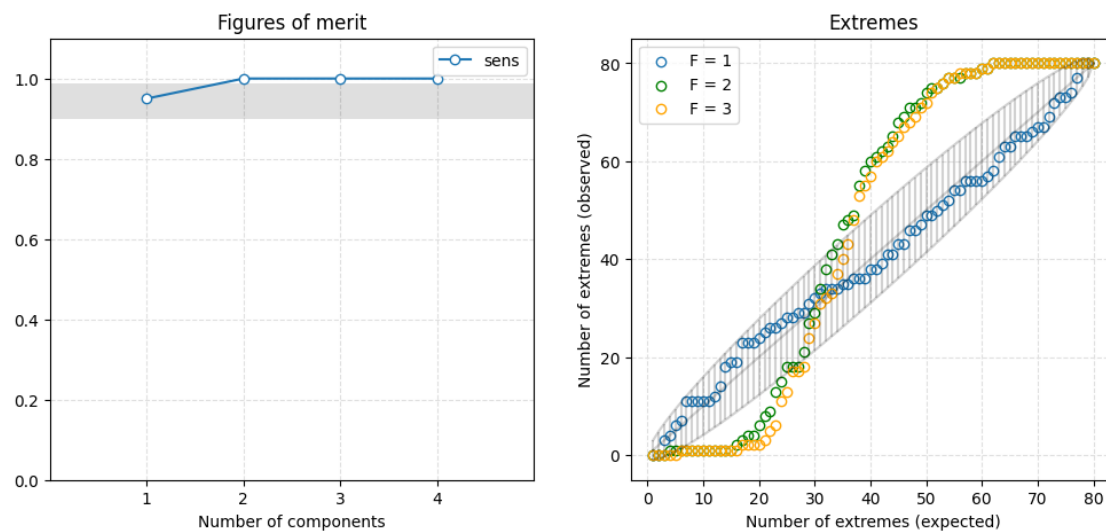


Finally, here are sensitivity and extremes plots made for the training set.

```
[12]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
r_train.plotFoM(ax1, fom = "sens", show_ci = True)

ax2 = plt.subplot(1, 2, 2)
r_train.plotExtremes(ax2, ncomp = 1, label = "F = 1")
r_train.plotExtremes(ax2, ncomp = 2, edgecolors = "green", label = "F = 2")
r_train.plotExtremes(ax2, ncomp = 3, edgecolors = "orange", label = "F = 3")
ax2.set_title("Extremes")
ax2.legend();
```



Predictions can be made using data frames with or without reference class labels. In the first case, the result object will contain all necessary figures of merit (sensitivity for members, specificity and selectivity for non-members, and accuracy and efficiency if the dataset contains both members and strangers). If reference classes are not provided, the model will simply make predictions (accepted / rejected).

We already have all the datasets prepared; let's create one more — the same as the alternative class samples but without the column of reference class labels, so it can be treated as new samples with unknown classes.

```
[13]: data_new_alt = data_test_alt.iloc[:, 1:]
```

Let's check what is inside the datasets:

```
[14]: data_test_alt.iloc[:5, :5]
```

```
[14]:
```

	classname	0	1	2	3
0	alt	0.024904	0.015852	0.030074	0.028308
1	alt	0.014783	0.008056	0.008097	0.014809
2	alt	0.030832	0.027685	0.037621	0.033441
3	alt	0.010113	0.010998	0.021467	0.023453
4	alt	0.006956	0.006930	0.004747	0.007169

```
[15]: data_new_alt.iloc[:5, :5]
```

```
[15]:
```

	0	1	2	3	4
0	0.024904	0.015852	0.030074	0.028308	0.019261
1	0.014783	0.008056	0.008097	0.014809	0.017050
2	0.030832	0.027685	0.037621	0.033441	0.038789
3	0.010113	0.010998	0.021467	0.023453	0.020079
4	0.006956	0.006930	0.004747	0.007169	-0.002168

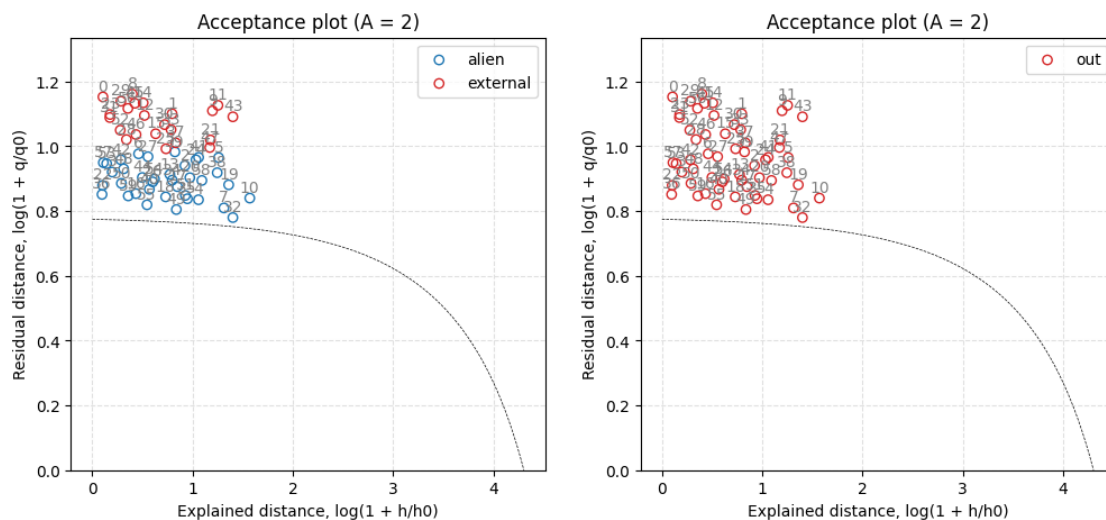
Now let's apply the model to both sets (remember that they contain the same objects, but one has a column with reference class labels and the second one does not). Then check the acceptance plot:

```
[16]: r_test_alt = m.predict(data_test_alt)
      r_new_alt = m.predict(data_new_alt)

      plt.figure(figsize = (12, 5))

      ax1 = plt.subplot(1, 2, 1)
      r_test_alt.plotAcceptance(ax1, do_log = True, show_labels = True)

      ax2 = plt.subplot(1, 2, 2)
      r_new_alt.plotAcceptance(ax2, do_log = True, show_labels = True)
```



As you can see, in the first case the model indeed treated the objects as belonging to non-target classes and shows the corresponding roles (alien and external in this case) on the plot. In the second case, it simply splits the samples into accepted (in) and rejected (out).

Let's see how different the summary information is:

```
[17]: r_test_alt.summary()
```

DDSIMCA-PARAFAC results:

- number of components (total): 4
- number of components (selected): 2
- limit type: classic
- alpha: 0.050
- gamma: 0.010

- class labels: provided
- number of objects: 60
- number of members: 0
- number of strangers: 60

PCs	eCrit	oCrit	in	out	TN	FP	spec	sel
1	11.070	25.233	51	9	9	51	0.15	0.024
2	292.488	345.679	0	60	60	0	1.00	0.993
3	291.336	344.319	0	60	60	0	1.00	0.998
4	292.488	345.679	0	60	60	0	1.00	0.996

```
[18]: r_new_alt.summary()
```

DDSIMCA-PARAFAC results:

- number of components (total): 4
- number of components (selected): 2
- limit type: classic
- alpha: 0.050
- gamma: 0.010

- class labels: not provided
- number of objects: 60

PCs	eCrit	oCrit	in	out
1	11.070	25.233	51	9
2	292.488	345.679	0	60
3	291.336	344.319	0	60
4	292.488	345.679	0	60

As you can see, again, most of the plots and outcomes are identical to those obtained for the conventional DD-SIMCA method. The main difference is that for the second dataset there are no columns with figures of merit, true negatives and false positives.

Now let's use data which has reference classes and objects of both target and alternative classes:

```
[19]: data_test
```

```
[19]:
```

	classname	0	1	...	1997	1998	1999
0	target	0.027985	0.031098	...	0.008460	0.012077	0.013859
1	target	0.005608	0.023140	...	0.021453	0.015444	0.005071
2	target	0.034567	0.038592	...	0.002163	-0.004820	-0.001560
3	target	0.038813	0.029240	...	0.003637	0.000404	0.002375
4	target	0.023743	0.026105	...	-0.006154	-0.003769	-0.007631
..
55	alt	-0.001382	0.006688	...	0.000915	0.008008	-0.000891
56	alt	0.031243	0.034019	...	0.000632	-0.008506	0.009325
57	alt	0.026348	0.024962	...	0.004145	0.004989	-0.001512
58	alt	-0.001668	0.009869	...	0.006640	0.001368	0.005848
59	alt	0.039432	0.036929	...	-0.005290	0.007036	0.008432

[80 rows x 2001 columns]

As in conventional DD-SIMCA, the acceptance plot can be shown only for members, only for strangers, or for all samples (the default option). In the latter case, the acceptance plot colors the points by class instead of by role until you change this by providing an explicit value for the parameter `show_set`:

```
[20]: r_test_all = m.predict(data_test)

plt.figure(figsize = (13, 4))

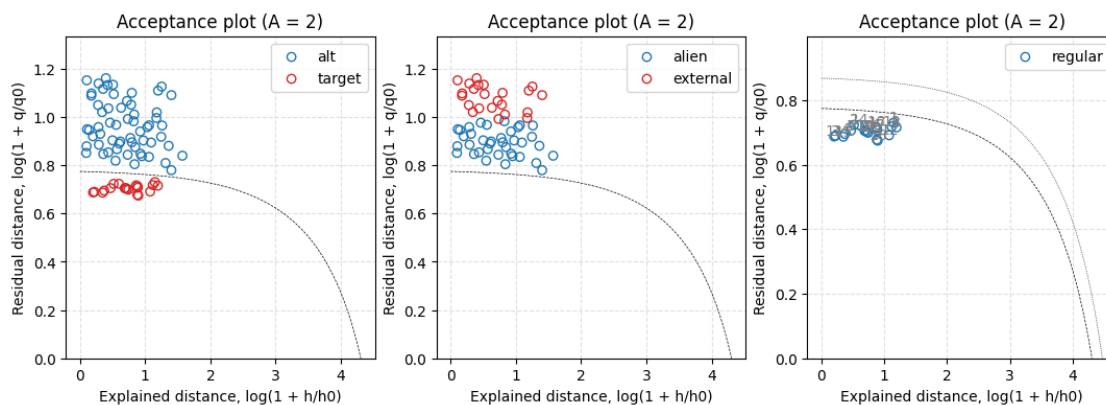
ax1 = plt.subplot(1, 3, 1)
r_test_all.plotAcceptance(ax1, do_log = True)
```

```

ax2 = plt.subplot(1, 3, 2)
r_test_all.plotAcceptance(ax2, do_log = True, show_set = "strangers")

ax3 = plt.subplot(1, 3, 3)
r_test_all.plotAcceptance(ax3, do_log = True, show_set = "members",
↪ show_labels=True)

```



You can provide your own colors, both for classification roles and for classes. Here are examples:

```

[21]: colors_classes = {"target": "gray", "alt": "magenta"}
      markers_classes = {"target": "s", "alt": "o"}

      colors_roles1 = {"external": "magenta", "alien": "green"}
      markers_roles1 = {"external": "s", "alien": "o"}

      colors_roles2 = {"regular": "green", "extreme": "magenta", "outlier": "red"}
      markers_roles2 = {"regular": "s", "extreme": "o", "outlier": "d"}

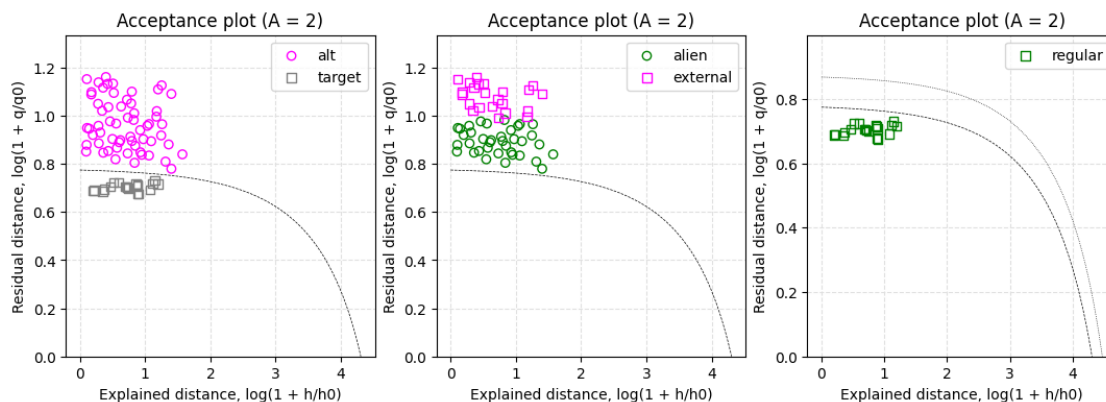
      plt.figure(figsize = (13, 4))

      ax1 = plt.subplot(1, 3, 1)
      r_test_all.plotAcceptance(ax1, do_log = True, colors = colors_classes, markers_
↪ markers_classes)

      ax2 = plt.subplot(1, 3, 2)
      r_test_all.plotAcceptance(ax2, do_log = True, show_set = "strangers",
                                colors = colors_roles1, markers = markers_roles1)

      ax3 = plt.subplot(1, 3, 3)
      r_test_all.plotAcceptance(ax3, do_log = True, show_set = "members",
                                colors = colors_roles2, markers = markers_roles2)

```



Also, in this case all figures of merit, including accuracy and efficiency, are available:

```
[22]: r_test_all.summary()
```

DDSIMCA-PARAFAC results:

- number of components (total): 4
- number of components (selected): 2
- limit type: classic
- alpha: 0.050
- gamma: 0.010

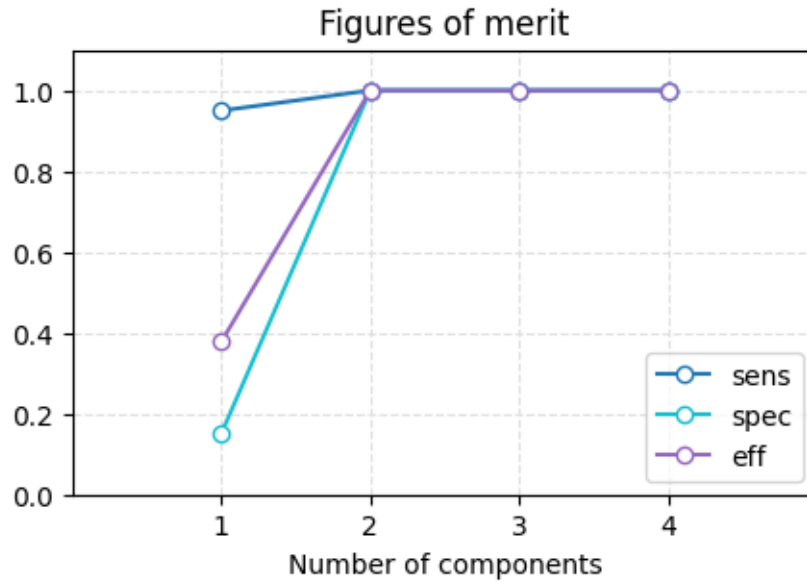
- class labels: provided
- number of objects: 80
- number of members: 20
- number of strangers: 60

PCs	eCrit	oCrit	in	out	TP	FN	sens	TN	FP	spec	sel	acc	eff
1	11.070	25.233	70	10	19	1	0.95	9	51	0.15	0.024	0.35	0.377
2	292.488	345.679	20	60	20	0	1.00	60	0	1.00	0.993	1.00	1.000
3	291.336	344.319	20	60	20	0	1.00	60	0	1.00	0.998	1.00	1.000
4	292.488	345.679	20	60	20	0	1.00	60	0	1.00	0.996	1.00	1.000

And they can be plotted together:

```
[23]: plt.figure(figsize = (5, 3))

ax = plt.subplot(1, 1, 1)
r_test_all.plotFoM(ax, fom = "sens")
r_test_all.plotFoM(ax, fom = "spec")
r_test_all.plotFoM(ax, fom = "eff")
```



3.0.1 Extra plots, features, and details

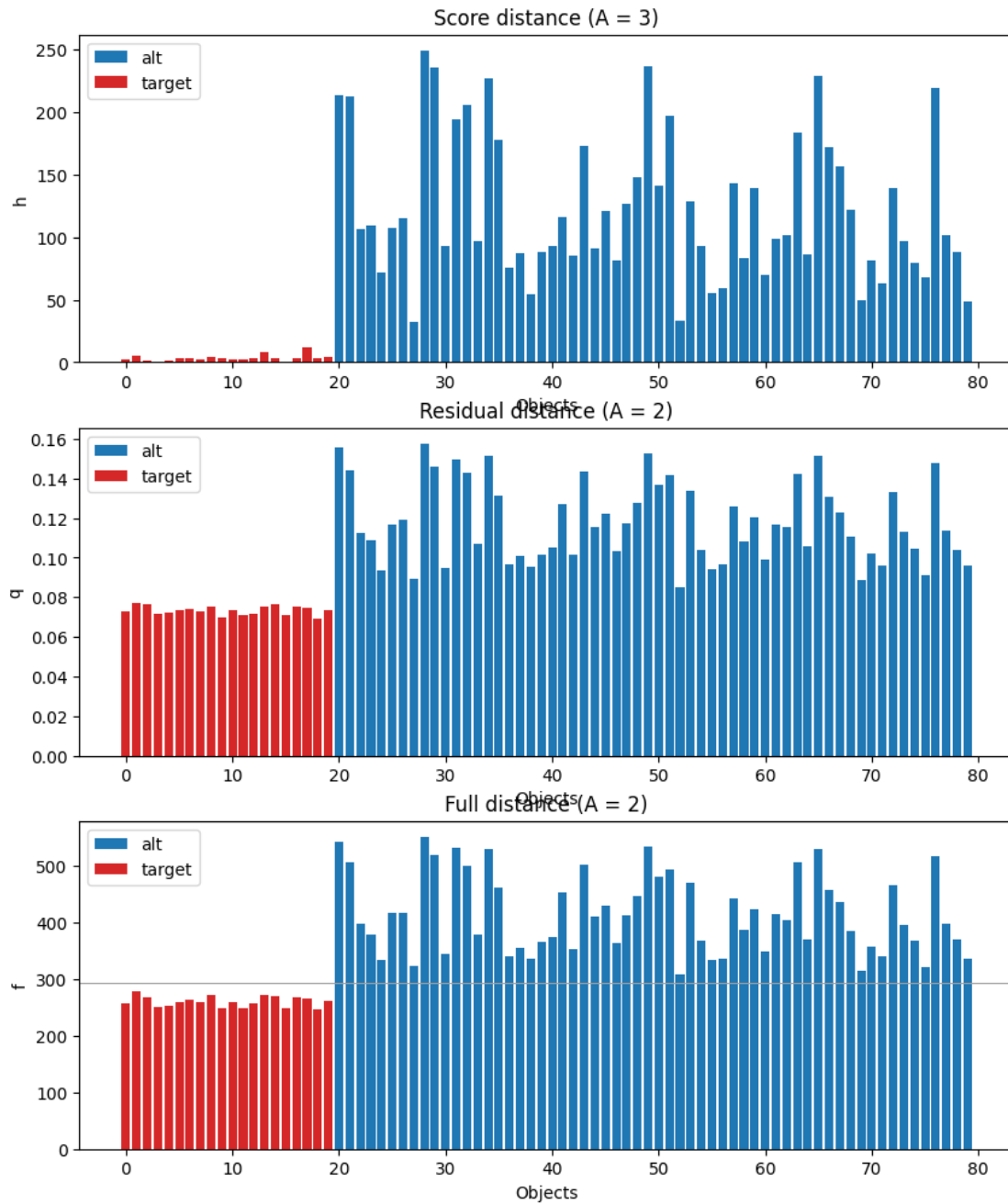
It is also possible to show every distance separately for a given number of components (this works for any result object):

```
[24]: plt.figure(figsize = (10, 12))

ax1 = plt.subplot(3, 1, 1)
r_test_all.plotDistance(ax1, ncomp = 3, distance = "h")

ax2 = plt.subplot(3, 1, 2)
r_test_all.plotDistance(ax2, ncomp = 2, distance = "q")

ax3 = plt.subplot(3, 1, 3)
r_test_all.plotDistance(ax3, ncomp = 2, distance = "f", show_crit = True)
```

You can also show the PARAFAC scores plot:

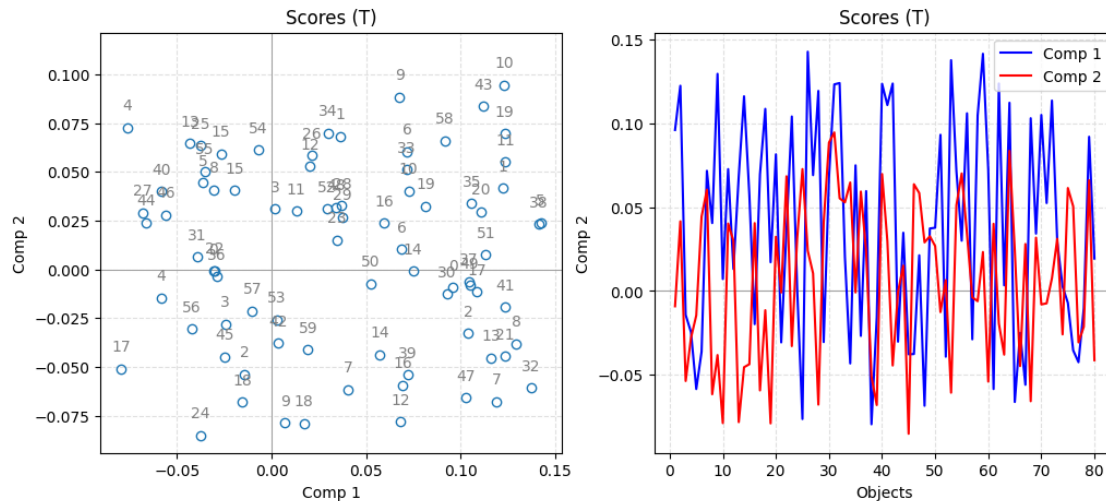
```
[25]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
r_test_all.plotScores(ax1, comp = (1, 2), type = "p", show_labels = True)
```

```

ax2 = plt.subplot(1, 2, 2)
r_test_all.plotScores(ax2, comp = (1,), type = "l", color = "blue")
r_test_all.plotScores(ax2, comp = (2,), type = "l", color = "red")
ax2.legend();

```



By default, T scores are shown. If you want to show the standardized scores (also known as left singular vectors), U, just provide an extra argument:

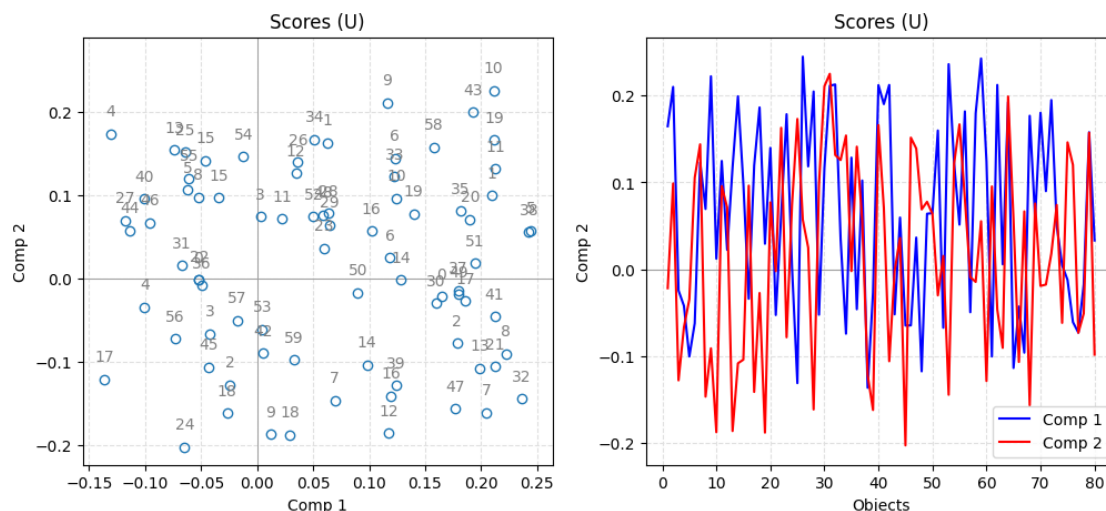
```

[26]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
r_test_all.plotScores(ax1, comp = (1, 2), type = "p", scores = "U", show_labels_
↳ True)

ax2 = plt.subplot(1, 2, 2)
r_test_all.plotScores(ax2, comp = (1,), type = "l", scores = "U", color = 
↳ "blue")
r_test_all.plotScores(ax2, comp = (2,), type = "l", scores = "U", color = "red")
ax2.legend();

```



There is an important difference here from conventional DD-SIMCA. Scores are always shown for the fixed optimal PARAFAC model — in our case, the model with 2 components. If you want to show scores for another sub-model you need to create a new set of results.

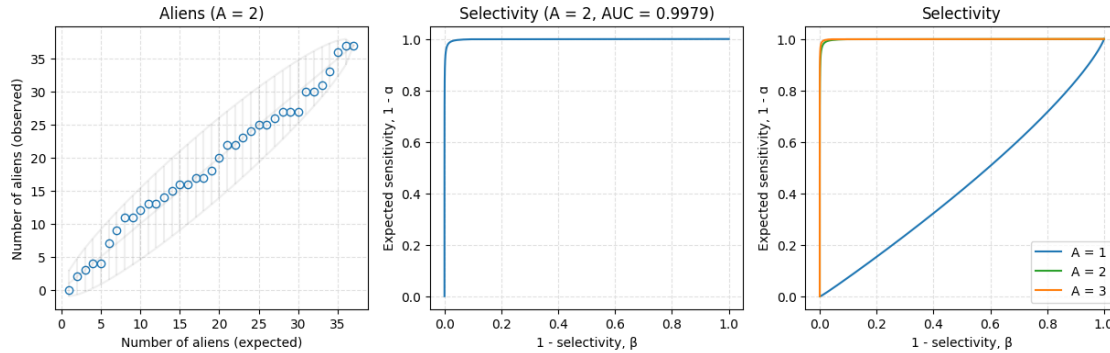
And of course you can also make a plot with expected vs. observed alien objects and a selectivity vs. sensitivity plot (if there are non-target class objects in the dataset):

```
[27]: plt.figure(figsize = (15, 4))

ax1 = plt.subplot(1, 3, 1)
r_test_all.plotAliens(ax1)

ax2 = plt.subplot(1, 3, 2)
r_test_all.plotSelectivity(ax2)

ax3 = plt.subplot(1, 3, 3)
r_test_all.plotSelectivity(ax3, 1, color = "tab:blue", label = "A = 1")
r_test_all.plotSelectivity(ax3, 2, color = "tab:green", label = "A = 2")
r_test_all.plotSelectivity(ax3, 3, color = "tab:orange", label = "A = 3")
ax3.set_title("Selectivity")
ax3.legend();
```



Any result object can also be converted to a data frame, where every object is represented by several columns: reference class (if provided), decision (in / out), role, and distance values. By default, these values are computed for the optimal number of components, but you can specify which number of components to use for creating the data frame:

```
[28]: rdf = r_test_all.as_df(ncomp = 2)
rdf.head()
```

```
[28]:
```

	class	decision	role	h	q	f
0	target	in	regular	2.178241	0.072702	257.382809
1	target	in	regular	4.250242	0.077303	277.586898
2	target	in	regular	1.337474	0.076223	267.929626
3	target	in	regular	0.490682	0.071412	249.473893
4	target	in	regular	0.889685	0.072154	252.865854

You can also access all computed values directly. For a model, this includes factors B and C, the centering vector for scores A, classic and robust parameters of the distance distributions (for the h-, q-, and f-distances), and more. Here are some examples:

```
[29]: # dictionary with the second PARAFAC model (for 2 components)
m.models[1].keys()
```

```
[29]: dict_keys(['weights', 'factors', 'B', 'C', 'A_train', 'A_mean', 'A_V',
'A_sigma', 'N_train', 'S_pinv', 'T_full', 'U_full'])
```

```
[30]: # Matrix with factors B for the model
m.models[1]["B"]
```

```
[30]: array([[5.98369743e-02, 3.01594550e-01],
[7.77551690e-02, 3.41869006e-01],
[9.67856558e-02, 3.87862551e-01],
[1.18060929e-01, 4.37049491e-01],
[1.51931600e-01, 4.8288292e-01],
[1.83607209e-01, 5.34346213e-01],
[2.22659144e-01, 5.79006272e-01],
```

```
[2.65320781e-01, 6.23183585e-01],
[3.17518165e-01, 6.63179038e-01],
[3.71878436e-01, 6.99397255e-01],
[4.33454078e-01, 7.30008923e-01],
[4.93908078e-01, 7.61318453e-01],
[5.68362294e-01, 7.76280109e-01],
[6.38748869e-01, 7.91227084e-01],
[7.19124467e-01, 7.93513473e-01],
[7.99344858e-01, 7.90258207e-01],
[8.79501453e-01, 7.80216404e-01],
[9.62693726e-01, 7.53536574e-01],
[1.03950221e+00, 7.29397850e-01],
[1.11639122e+00, 6.97207944e-01],
[1.18217445e+00, 6.61949190e-01],
[1.24592909e+00, 6.20735432e-01],
[1.30503489e+00, 5.76355823e-01],
[1.34960994e+00, 5.27729249e-01],
[1.38659378e+00, 4.77493494e-01],
[1.41501610e+00, 4.29142935e-01],
[1.42783805e+00, 3.87576163e-01],
[1.43199428e+00, 3.39890801e-01],
[1.42523485e+00, 2.95754643e-01],
[1.40728544e+00, 2.54759657e-01],
[1.37505435e+00, 2.19612061e-01],
[1.34083246e+00, 1.81125266e-01],
[1.29736168e+00, 1.53434160e-01],
[1.23687554e+00, 1.27605181e-01],
[1.17405182e+00, 1.08959848e-01],
[1.11446486e+00, 8.46458368e-02],
[1.03712626e+00, 6.75925479e-02],
[9.66759326e-01, 5.30799129e-02],
[8.89683402e-01, 4.38121060e-02],
[8.12137738e-01, 3.60050906e-02],
[7.41914572e-01, 2.49088032e-02],
[6.63172075e-01, 2.52592931e-02],
[5.95437237e-01, 1.63737090e-02],
[5.29961245e-01, 1.36178588e-02],
[4.67090836e-01, 5.17121479e-03],
[4.09849573e-01, 4.09375792e-03],
[3.52616173e-01, 9.79859576e-04],
[3.03005155e-01, 5.66678434e-03],
[2.62445879e-01, 8.64576380e-04],
[2.20190406e-01, 4.12231583e-03]])
```

```
[31]: # matrix with mean values of training scores A
m.models[1]["A_mean"]
```

```
[31]: array([0.09238168, 0.08676261])
```

Like in conventional DD-SIMCA, the distance parameters are saved as dictionaries `hParams`, `qParams`, and `fParams`. Each dictionary has two fields, `classic` and `robust`. Each field contains a tuple with scaling factors (e.g. `h0`) and the number of degrees of freedom (e.g. `Nh`) computed for each component in the model. Here is an example for score distance parameters computed using classic estimates:

```
[32]: m.hParams["classic"]
```

```
[32]: (array([0.98749993, 1.97500105, 2.96166349, 3.95035332]),
      array([2., 4., 3., 4.]))
```

Here is another example showing how to get `q0` and `Nq` values for the robust estimator and 2 components:

```
[33]: q0, Nq = m.qParams["robust"]
      A = 2
      print(f"A = {A}: q0 = {q0[A - 1]:.3f}, Nq = {Nq[A - 1]}")
```

A = 2: q0 = 0.044, Nq = 100.0

For a result object, you can access all outcomes — including critical values, distances, calculations related to the Type II error (if non-target class objects are provided), and many other things — via the `outcomes` data frame:

```
[34]: r_test_all.outcomes.head()
```

```
[34]:
```

	PCs	eCrit	oCrit	...	sel	acc	eff
0	1	11.070498	25.232900	...	0.023679	0.35	0.377492
1	2	292.487605	345.679460	...	0.992673	1.00	1.000000
2	3	291.336079	344.318517	...	0.997793	1.00	1.000000
3	4	292.487605	345.679460	...	0.996176	1.00	1.000000

[4 rows x 22 columns]

You can also access matrices (`nrows x ncomp`) with the h-, q-, and f-distances, and a matrix with decisions:

```
[35]: r_test_all.H[:3, :5]
```

```
[35]: array([[ 2.1085236 ,  2.1782414 ,  2.18420421,  2.30312973],
          [ 2.27534117,  4.25024159,  5.63723947, 16.54555987],
          [ 0.01998362,  1.33747393,  1.34287756,  2.15289113]])
```

```
[36]: r_test_all.Q[:3, :5]
```

```
[36]: array([[0.4589531 , 0.07270233, 0.07271126, 0.07285977],
          [1.28962277, 0.07730283, 0.07713284, 0.07547985],
          [0.85173541, 0.0762228 , 0.07612351, 0.07623553]])
```

```
[37]: r_test_all.F[:3, :5]
```

```
[37]: array([[ 6.5198539 , 257.3828089 , 256.09202702, 257.37720172],
          [ 10.92899974, 277.58689762, 275.02822632, 280.97016677],
          [ 4.21500823, 267.92962642, 267.15408425, 269.041892  ]])
```

```
[38]: r_test_all.D[:3, :5]
```

```
[38]: array([[ True,  True,  True,  True],
          [ True,  True,  True,  True],
          [ True,  True,  True,  True]])
```

Here, for example, are the decisions obtained for each object using $A = 3$ components:

```
[39]: r_test_all.D[:, 2]
```

```
[39]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
          True,  True,  True,  True,  True,  True,  True,  True,  True,
          True,  True, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False, False])
```

And a matrix with roles, which are coded as integer values: 0 for **regular**, 1 for **extreme**, 2 for **outlier**, 3 for **alien**, and 4 for **external**. The first three are assigned to target class members; the last two — to objects from non-target classes and to unknown objects:

```
[40]: r_test_all.R[:, 2]
```

```
[40]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4,
          3, 3, 3, 4, 4, 3, 4, 4, 3, 4, 4, 3, 4, 4, 3, 3, 3, 3, 3, 4, 3, 4,
          3, 4, 3, 4, 4, 4, 4, 4, 3, 4, 3, 3, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4,
          4, 4, 3, 3, 3, 3, 4, 3, 3, 3, 4, 3, 3, 3], dtype=int16)
```

Here is a simple way to get the role names only for members:

```
[41]: import numpy as np
r_names = np.array(["regular", "extreme", "outlier", "alien", "external"])
r_names[r_test_all.R[:, 2]]
```

```
[41]: array(['regular', 'regular', 'regular', 'regular', 'regular', 'regular',
          'regular', 'regular', 'regular', 'regular', 'regular', 'regular',
          'regular', 'regular', 'regular', 'regular', 'regular', 'regular',
          'regular', 'regular', 'external', 'external', 'alien', 'alien',
          'alien', 'external', 'external', 'alien', 'external', 'external',
          'alien', 'external', 'external', 'alien', 'external', 'external',
          'alien', 'external', 'external', 'alien', 'external', 'external'])
```

```
'alien', 'alien', 'alien', 'alien', 'alien', 'external', 'alien',
'external', 'alien', 'external', 'alien', 'external', 'external',
'external', 'external', 'external', 'alien', 'external', 'alien',
'alien', 'alien', 'external', 'alien', 'external', 'alien',
'external', 'alien', 'external', 'alien', 'external', 'external',
'external', 'alien', 'alien', 'alien', 'alien', 'alien', 'external',
'alien', 'alien', 'alien', 'external', 'alien', 'alien', 'alien'],
dtype='<U8')
```

4 DD-SIMCA Tucker

Because the Tucker-based DD-SIMCA is very similar to the conventional and PARAFAC DD-SIMCA, we will show only what is different here and omit the rest of the functionality.

The main difference between PARAFAC and Tucker is that for Tucker, matrices A, B, and C may have different numbers of components/factors, e.g. N_A , N_B , N_C . So when you train the model you need to provide all three numbers, not just one.

The first number, N_A , is treated as the maximum number of components. `ddsimca_tucker()` will create N_A decompositions, starting from $[1, N_B, N_C]$ and ending with $[N_A, N_B, N_C]$. So when you select the optimal number of components, or when you show a plot with figures of merit vs. the number of components, it is always the sample-mode component that varies — the other two are fixed.

Therefore the main difference is that the value of `ncomp` for `ddsimca_tucker` is a tuple of three numbers.

Here is how to train the model:

```
[42]: from ddsimca import ddsimca_tucker

m = ddsimca_tucker(data_train, (50, 40), ncomp = (3, 2, 3))
m.summary()
```

DDSIMCA-TUCKER model:

- target class: target
- number of components (total): 3 (1x2x3-3x2x3)
- number of components (optimal): 3 (3x2x3)
- number of training samples: 80
- number of variables: 2000 (50 x 40)

Parameters for classic estimators:

Comp	Nh	Nq
1	2	2
2	4	250
3	3	250

As you can see, the line with the total number of components shows three items: 3 — the maximum

number of components for the sample mode; 1x2x3 — the smallest number of components in the model; and 3x2x3 — the largest. As mentioned above, the last two are fixed while the first varies from 1 to the maximum number.

Here is what happens if you change the optimal number:

```
[43]: m.select_ncomp(2)
      m.summary()
```

DDSIMCA-TUCKER model:

- target class: target
- number of components (total): 3 (1x2x3-3x2x3)
- number of components (optimal): 2 (2x2x3)
- number of training samples: 80
- number of variables: 2000 (50 x 40)

Parameters for classic estimators:

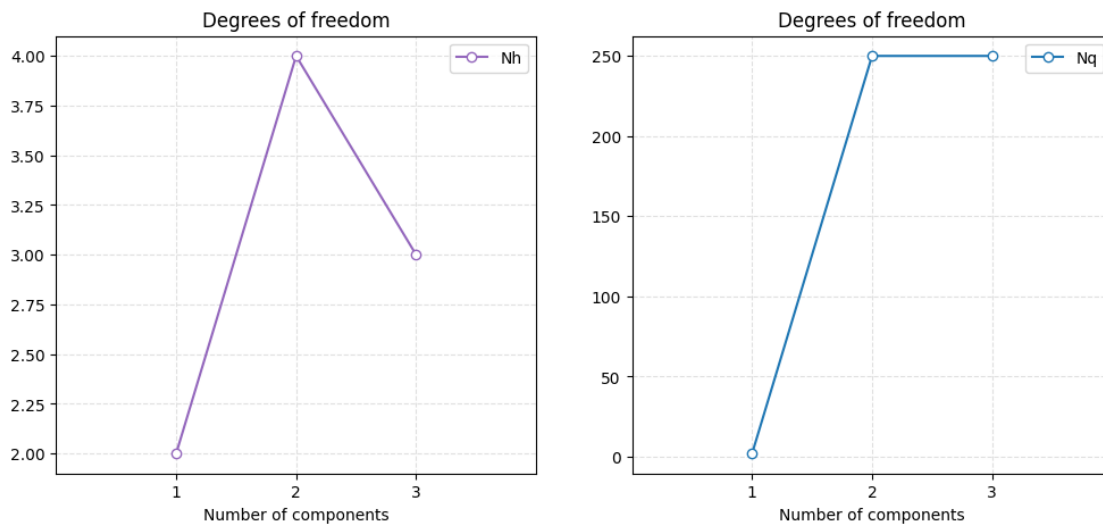
Comp	Nh	Nq
1	2	2
2	4	250
3	3	250

If we make a DoF plot:

```
[44]: plt.figure(figsize = (12, 5))

      ax1 = plt.subplot(1, 2, 1)
      m.plotDoF(ax1, dof = "Nh")

      ax2 = plt.subplot(1, 2, 2)
      m.plotDoF(ax2, dof = "Nq")
```



The number of components is also related to the sample mode, so 1 here means $1 \times 2 \times 3$, and so on.

Like in PARAFAC, you can show the factors as plots — just remember that the number of factors for B and C can be different:

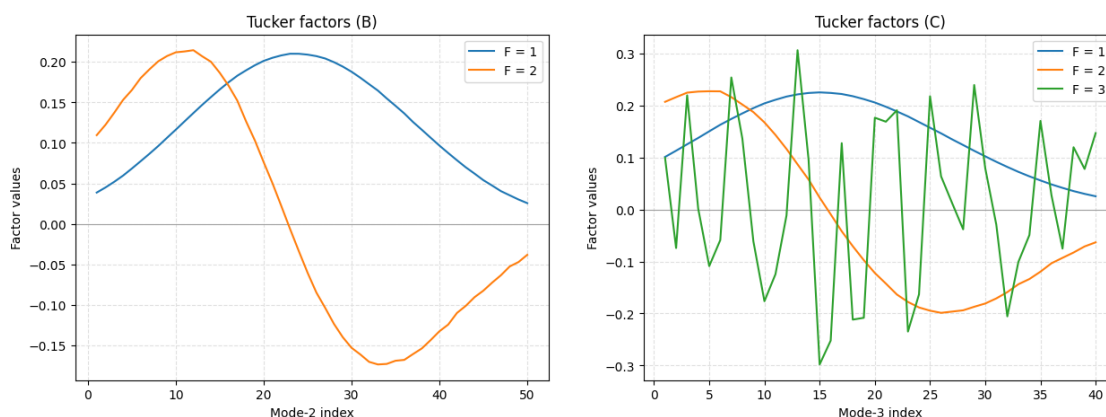
```
[45]: import matplotlib.colors as mcolors

# get a list with four distinct colors from the TABLEAU colormap
cols = list(mcolors.TABLEAU_COLORS.values())[4:]

plt.figure(figsize = (15, 5))

# factors B
ax1 = plt.subplot(1, 2, 1)
for f in range(1, 3):
    m.plotFactors(ax1, f, mode = "B", color = cols[f - 1], label = f"F = {f}")
ax1.legend()

# factors C
ax2 = plt.subplot(1, 2, 2)
for f in range(1, 4):
    m.plotFactors(ax2, f, mode = "C", color = cols[f - 1], label = f"F = {f}")
ax2.legend();
```



The third component/factor in C is clearly noise, as expected.

Predictions work similarly to the conventional and PARAFAC implementations.

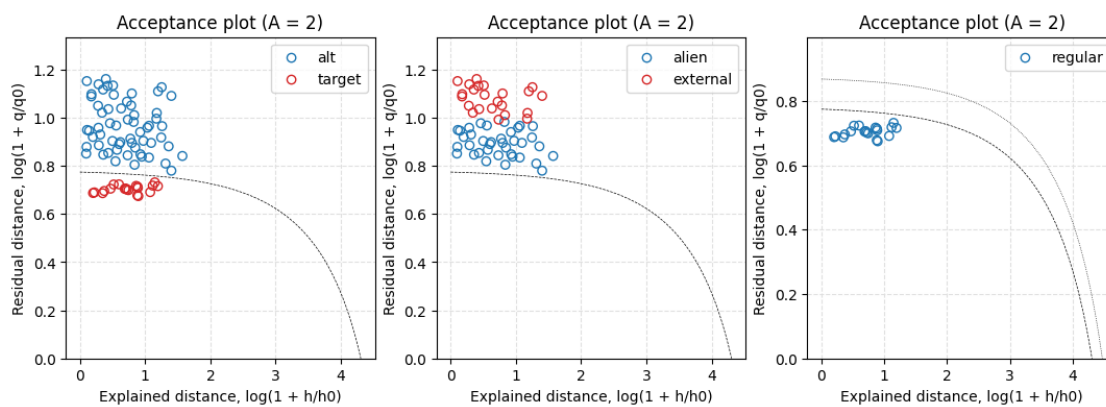
```
[46]: r_test_all = m.predict(data_test)

plt.figure(figsize = (13, 4))

ax1 = plt.subplot(1, 3, 1)
r_test_all.plotAcceptance(ax1, do_log = True)

ax2 = plt.subplot(1, 3, 2)
r_test_all.plotAcceptance(ax2, do_log = True, show_set = "strangers")

ax3 = plt.subplot(1, 3, 3)
r_test_all.plotAcceptance(ax3, do_log = True, show_set = "members")
```

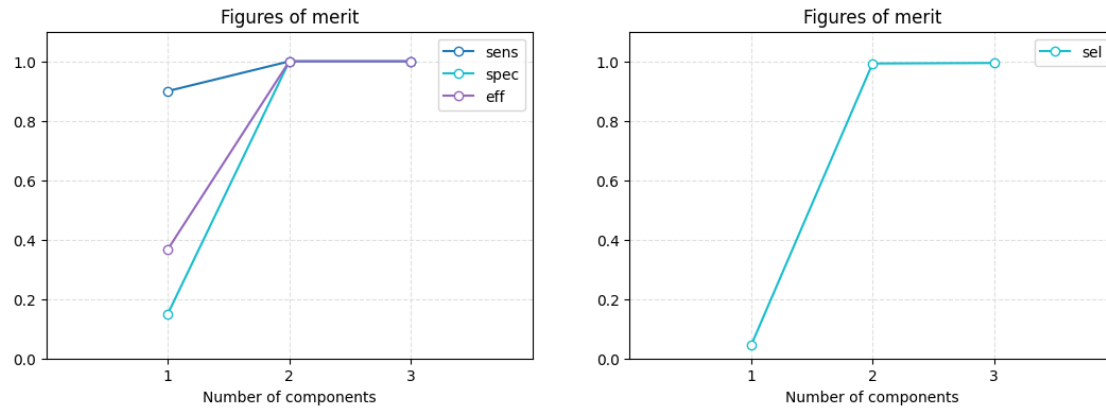


The figures of merit are also similar — just remember that the number of components on this plot means the number of components/factors along the sample mode:

```
[47]: plt.figure(figsize = (13, 4))

ax = plt.subplot(1, 2, 1)
r_test_all.plotFoM(ax, "sens")
r_test_all.plotFoM(ax, "spec")
r_test_all.plotFoM(ax, "eff")

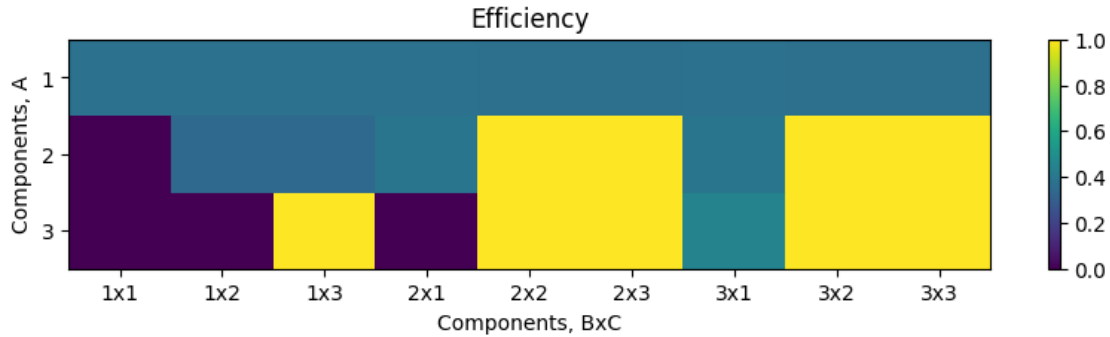
ax = plt.subplot(1, 2, 2)
r_test_all.plotFoM(ax, "sel")
```



If you want to find optimal efficiency across all 3 modes, just use loops. Here is an example where we will search for the best combination of N_A , N_B , N_C in a range $[1, 3]$:

```
[48]: eff = np.zeros((9, 3))
labels = [""] * 9
n = 0
for b in [1, 2, 3]:
    for c in [1, 2, 3]:
        a = np.min([3, b * c])
        m = ddsimca_tucker(data_train, (50, 40), ncomp = (a, b, c))
        r = m.predict(data_test)
        eff[n, :a] = np.array(r.outcomes["eff"])
        labels[n] = f"{b}x{c}"
        n = n + 1

fig = plt.figure(figsize = (10, 2))
ax = plt.subplot(1, 1, 1)
im = ax.imshow(eff.T, clim = (0, 1), aspect="auto")
plt.colorbar(im)
ax.set_xticks(np.arange(0, len(labels)))
ax.set_xticklabels(labels);
ax.set_yticks(np.arange(0, 3))
ax.set_yticklabels(np.arange(1, 4))
ax.set_ylabel("Components, A")
ax.set_xlabel("Components, BxC")
ax.set_title("Efficiency");
```



4.1 Imposing non-negativity

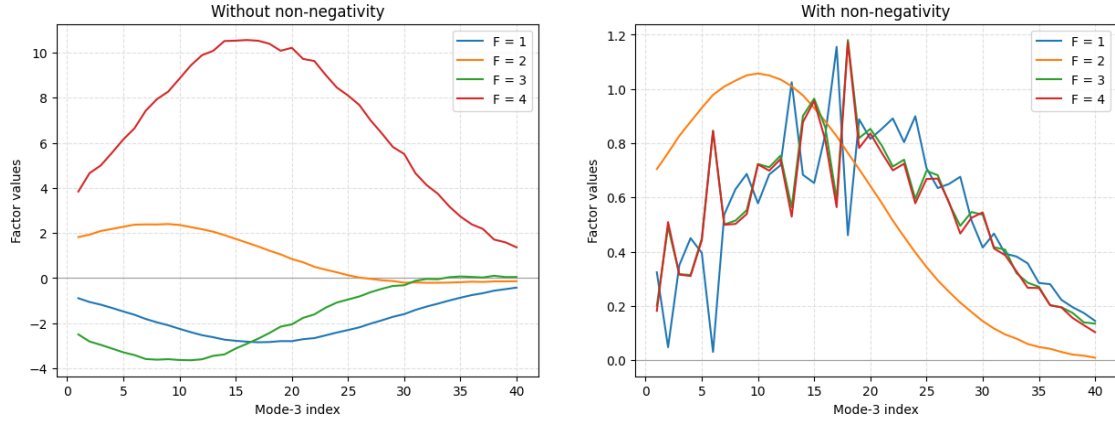
Both `ddsimca_parafac` and `ddsimca_tucker` can use a non-negativity constraint for the corresponding decomposition. Simply specify `non_negative = True` when training a model. Here is an example for PARAFAC:

```
[49]: # train two PARAFAC based models with and without non-negativity
m1 = ddsimca_parafac(data_train, dim = (50, 40), ncomp = 4)
m2 = ddsimca_parafac(data_train, dim = (50, 40), ncomp = 4, non_negative=True)
```

```
[50]: plt.figure(figsize = (15, 5))

# factors C without non-negativity constraint
ax1 = plt.subplot(1, 2, 1)
for f in range(1, 5):
    m1.plotFactors(ax1, f, mode = "C", color = cols[f - 1], label = f"F = {f}")
ax1.set_title("Without non-negativity")
ax1.legend()

# factors C with non-negativity constraint
ax2 = plt.subplot(1, 2, 2)
for f in range(1, 5):
    m2.plotFactors(ax2, f, mode = "C", color = cols[f - 1], label = f"F = {f}")
ax2.set_title("With non-negativity")
ax2.legend();
```



As you can see, without the constraint some of the C-mode factors take negative values, while with `non_negative = True` all factors are non-negative. This is often useful when the underlying signals (e.g. concentration profiles or spectra) are known to be non-negative on physical grounds — the constrained decomposition then becomes easier to interpret. The same option is available for `ddsimca_tucker`.