Technische Universität München
Fakultät für Physik

Manual

The GRid Accelerated Direcly SCoring OPTimizing tool kit (GRADSCOPT) to create problem-adopted knowledge-based scoring potentials

Sasse A., de Vries S., Schindler C., de Beauchêne I. and Martin Zacharias

Thursday 29th September, 2016

Contents

Chapter 1

Manual for the Development of Knowledge-Based Scoring Functions

The @-Scoring tool kit offers scripts for the parametrization of knowledge-based scoring functions, the fast re-scoring of decoys and for their assessments. Scoring functions can be generated for any type of decoys or any problem by trying out different parametrization methods, several structural representations and various potential forms.
To generate a scoring function out of the combination of these possibilities, a benchmark has to be set up which contains a sufficient number of complexes for the training and the assessment afterwards. Knowledge-based scoring functions are able to improve the overall performance of docking protocols when sampling and scoring are aligned [4]. Hence, it is recommended to generate the decoy set by using the same sampling algorithm as it will be used in the docking protocol.
To gain an optimal scoring performance, the right structural representation has to be considered. It defines the number of atom or bead types the potential should consist of. To make fast training and a fast re-scoring possible it is necessary to generate binary pre-grids for the desired potential form. First, these grids are used to generate a parameter set by direct Monte-Carlo Annealing or linear Regression. Finally, the obtained parameters serve to re-score the decoy-sets of each complex to evaluate the performance of the generated knowledge-based scoring function.
This manual leads the user step by step through all levels to obtain a sufficient scoring function and explains the necessary commands to run the scripts of the @-Scoring tool kit. The shell-script 'example.sh' can be found attached to illustrate the general procedure. Furthermore, the Attract_benchmark can found attached which was used to create scoring functions in the paper. The benchmark consists out of 166 protein-protein complexes and contains decoy sets from Attract's rigid-body sampling with unbound conformations. In general, it might be useful to consider the use of a cluster for the creation of the pre-grids and certain re-scoring since these steps have to be performed for each decoy of each complex.

1.1 Set up Benchmark

To set up a benchmark for the training of knowledge-based scoring functions, it is necessary to create a folder for each complex containing a set of decoys from the targeted sampling or refinement algorithm. Furthermore, files which determine the quality of the conformations have to be present for an optimization of the parameters. These assessments are generated by the comparison of the native complex to the generated decoys by their fnat, Irmsd, Lrmsd, the Capri stars or other assessment values.

Attract

For the Attract docking engine and iAttract refinement, the unbound and bound 3D structures of the constituents and the output-file from the sampling must be in the folder for each complex. Attract generates an output-file which contains the degrees of freedom of each decoy to align the receptor and ligand at the sampled position (www.attract.ph.tum.de). Since the tool kit was created for scoring functions of Attract's sampling and refinement, ensembles, normal modes and solutions from iAttract can used as in Attract itself.

Rigid-body decoys

Rigid-body solutions from other docking protocols, such as HADDOCK (http://haddock.chem.uu.nl), can be transformed into the Attract output format by the tool `haddock2attract.py` from the Attract program collection.

Flexible docking or refinement

Unfortunately, solutions in the pdf-format from other flexible docking protocols have to be divided into their constituents and separately reduced into the favoured structural representation. This is implemented in `pdb_split.sh` using `splitpdb.py` and is very time demanding. The name of the pdb which contained all decoys can be submitted afterwards to generate the pre-grids for the parametrization.

Defining complexes for the training set

The benchmark must be divided into a training and a test set. For that reason, a simple text file with a list of all complexes which are supposed to be in the training set has to be prepared.

1.1.1 cross-distribution.py

`cross-distribution.py` is a simple program to permute the list of training complexes n times to perform crossvalidation. The last n'th part of the list serves as the validation set in a training procedure.
To use `cross-distribution.py` the list of the complexes has to be named by the argument `--complexes`, the number of permutations by `--numtestset` and the name of the output file by `--output`. If `--complexes` is not given, it takes the names of all the folders in the executive directory by default to prepare the crossvalidation files. As output it generates n files supplemented by '-cv' and numbered from 1 to n.

```
$python cross-distribution.py
     --complexes trainset.txt
     --numtestset n
     --output outname
     --exceptions 2 1ACB 1EWY
```

By the use of `--exceptions` the number and names of complexes which should be excluded from the training sets can be given.

1.1.2 capristars.py

The Capri stars evaluate the quality of the generated decoys after the assessment table in Wodak et al. 2007 [2] and thus they may be used as outcomes for the linear regression method or as quality weights in the Monte Carlo Annealing algorithm. The program `capristars.py` generates a file which contains the assigned stars from the files of the lrmsd, irmsd and the fnat. The output-file possess the name of the file from the ligand rmsd and ends on `.capstars`.

```
$python capristars.py file.lrmsd file.irmsd file.fnat
```

1.2 Generation of feature vectors

1.2.1 redatom.py

To generate a pdb-file containing the atomistic coordinates and assigning atom types from OPLS [1], the tool `aareduce.py` from the ATTRACT directory has to be used. To generate a coarse-grained model in Attract's representation out of it, `reduce.py` is used afterwards.
Additionally, the @-scoring tool kit offers the possibility to assign atomistic representations to the structures from Tobi et al. [3] and the new developed atomistic

grouped all atom (GAA) model.
Therefore, `redatom.py` uses a file for the specific atom types by `--atomtypefile atomtypes_model.dat`. With the argument `--output` one sets the name of the output pdb file and with `--pdb` one assigns the pdb-file which has to be converted to the chosen model. By using the command `--addNH` one can insert the hydrogen atoms of the backbone nitrogens. Furthermore, it is possible to define by `--atomtypesonly` that only the atom types of the representation will be changed but their charges won't be adopted from the file with the atom types.

1.2.2 collect-function.py

To train the parameters of the scoring functions and to rescore all the decoy sets quickly, it is necessary to precalculate feature vectors based on the form of the scoring function. The program `collect-function.py` is a python program using a Fortran written library which is implemented into python with **f2py**. Furthermore, it uses `collectlibpy.py` in combination with `collectlib.so` from the Attract program to generate the coordinates of the decoy poses from the Attract output-file on the run. To use collect-function.py the library `collectgridlib.f` has to be compiled with **f2py** and deposited in the same directory as the program collect-function.py. The library is compiled with f2py by:

```
$f2py -c -m collectgridlib collectgridlib.f
```

In addition, a soft link to `collectlibpy.py` and `collectlib.so` from the `/bin` folder in Attract has to be established.
The program itself can be executed on the terminal by providing it with the `output.dat` file from ATTRACT with the degrees of freedom for each decoy pose or with the name of the pdb-file containing all decoys. Furthermore, with the argument `--proteinmodel` structures of the ligand and the receptor are inserted as pdb's. By default the program generates binary grids for the feature vectors in single precision. It is executed by:

```
$python collect-funtion.py output.dat/decoys.pdb
```

`--proteinmodel`

1. `representation.reduce`
2. `receptor.pdb`
3. `ligand.pdb`

`--gridtype [spline, none, distances]`

The entry behind `--proteinmodel` has to be the position of a file that contains the number of atom-types as its header (# natom) and any replacement of atom-type

numbers in the following, for instance to replace atom-type 88 with 16 (88 16)(See also example files opls.reduce etc.). The reduction is necessary in order to use only uprising numbers as atom types without interruptions to save memory space and computational time.
The output grid contains feature vectors for each decoy in the coordinates file and will be in shape `(bins, numstructures, numparameter)`. The number of parameters `numparameter` is derived for contact grids by `numparameter=atomtypes*(atomtypes-1)/2+atomtypes`. Grids with double precision can be generated by the use of `--double`.
By inserting `--maxstruc n_maxstruc(default 100.000)` the number of maximum decoy poses for a grid can be defined. This might be relevant if grids become too large for the internal memory of the machine. The `--gridtype` can be `spline` for grids which are used for the generation of any potential form by interpolation, `none` for grids which count the number of contacts in given ranges or `distances` for grids which sums up all distances to a given power for all atoms of each contact type.
The argument `--polationparams` defines more precisely the content of the grid for each `gridtype`.

`--polationparams`

`if spline:`

1. `k_degreee(default=4)`
2. `stepsize(default=0.35)`
3. `start(default=1.4)`
4. `end(default=7.)`

`if none or distances:`

1. `stepsize(default=1.)`
2. `start(default=2.)`
3. `end(default=10.)`

For the step potentials (`none`), `--polationparams` defines grids which contain a number of equal steps which can be summed up in the training script afterwards to generate various ranges of the step potential. The `stepsize`, the `start` and the `end` characterize the ranges of the generated regular steps in the vectors.
However, for the step potentials it might be more convenient to define the steps directly and not to precalculate a grid which can serve for the generation of a final step grid. For this purpose, the argument `--bins` can be used to define the step lengths of the grids by giving first the number of steps and then their distances from the center.

`--bins n_bins rcut0 ··· rcutn`

For random potential forms like the saddle point potentials (`spline`) the parameters `stepsize`, `start` and `end` define the placement of the nodes between which will be interpolated in the training. The argument `k_degree` determines the number of nodes considered for an interpolation and thus the degree of the interpolation. Hence, a degree of 2 stands for a linear interpolation, 3 for a quadratic and so on.
To find an appropriate degree for ones interpolation, the parameter of Attract can be inserted by `--attractpar` to print out the interpolated values for each structure and to compare them to the original values.
For the `distances` grids, which can be used for the training of normal LJ-like potentials, one can divide the feature vectors into different steps of size `stepsize`, starting at `start` and ending at `end` in order to sum them up later in the training program for a training at different cutoffs. Furthermore, the distances grids safe the sum of the distances between two atom types for each power separately to use different combinations of repulsive and attractive powers afterwards. Hence, the number of powers `n_power` and the powers themselves to which the sums are stored has to be given by the argument `--powers`. The final distances grid possess a shape of the form `(bins, n_power, numstructures, numparameter)`:

`--powers n_powers p0 ··· pn`

1.2.3 asa.py

The program `asa.py` computes solvent accessible surface areas, the total buried surface area and buried surface areas for defined atom types by the rolling probe algorithm. The total buried surface area can also be seen as a score, whereas the bsA for the atom types can further be used as features for a scoring function. As well as collect-function.py the script is dependent on two libraries which have to be present or soft-linked into the same directory as the main program. Softlinks for the files `colleclibpy.py` and `colleclib.so` have to be set from the Attract directory. The Fortran library `asalib.f` has to be compiled with **f2py** for its use in the python program. The program can be executed easily in the terminal by giving it 4 to 5 arguments:

`$python asa.py`

1. `receptor.pdb`
2. `ligand.pdb`
3. `output.dat/decoys.pdb`
4. `[atomistic, buriedsa, asa]`

if `atomistic:`

5. `[chemical, file.reduce]`

The fourth parameter defines the data which will be generated. The `buriedsa` and `asa` print out the buried surface area and the solvent accessible area of the complex respectively. The parameter `atomistic` generates feature vectors of the bsA for the chosen atom types. For the `chemical` atom types it only respects their chemical symbol H, O, C, N, S on the interface. Any other representation creates grids of atomistic bsA's with a grid shape in the form `(N_structures, n_atomtypes)`. To reduce the vector size, unused atom type numbers should be filled by downgrading high atom types with the file.reduce. This file contains the total number of atom types (# natom) in the first line and all atom types which have to be renumbered below (f.e. 99 32, to substitute atom type 99 with 32, see also opls.reduce, attract.reduce etc.). For the usage of pseudo atoms in the `attract` description, the size for the van der Waals radii of the pseudo atoms can be adjusted with the argument `--coarse_grained`. The coarse grained van der Waals radii were defined by Zacharias 2003 [5].
Furthermore, it is possible to change the size of the rolling probe in the algorithm with the argument `--watersize r_probe(default=1.4)`.

1.2.4 decoysetmixer.py

It might be necessary for a good result to enrich the training set with near-native structures which were scored badly by the Attract-score or to supplement structures which were generated by another sampling method. This can whether be done by resorting and supplementing the ATTRACT output file containing the degrees of freedom or by using `decoysetmixer.py` to resort and supplement grids of the same form.

1.3 Training Parameters

1.3.1 training-MC.py

The program `training-MC.py` uses Monte Carlo Annealing to predict a parameter set for scoring functions for various potential forms. It uses precalculated grids which contain features for each parameter and structure of each complex. To optimize parameters, it is possible to define a parameter set as starting position or to start the algorithm from a random distribution. The program uses the subroutine `fgenlib.py` for the computation of function values at the nodes which are defined for the interpolation algorithm. Furthermore, it uses `duplicatelib.py` to duplicate structures in the decoy sets after a given probability (f.e. to be a *-structure after refinement), so that 10 % of the training structures for each complex have a probability except

from 0. Both libraries have to be in the same directory as the main program for its execution by:

```
$python training-MC.py
    --complexes [ ]
    --grid [ ]
    --mcparams [ ]
    --evaluate [ ]
    --preparameter [ ]
    --output [ ]
```

Due to the variety of the potential types which can be handled by `training-MC.py`, the number and type of parameters which have to be given for each argument can vary. Below it is described which parameter have to be given for the arguments above.

`--complexes`

- `list_of_trainset.txt`
- `n_crossvalidation`

By `--complexes` a data-file containing the list of all the protein complexes for training purposes is provided. The folders in this list must contain the precalculated grids and the qualitative evaluations (irmsd, fnat, ...) which are used by `training-MC.py`. Secondly, the number `n_crossvalidation` for leave one out crossvalidation has to be defined. The program trains the parameter on the fraction of the first $(n-1)/n$ complexes of the given training benchmark, and validates the results on the last $1/n$ complexes in the list.

`--grid`

- `precalc-grid.npy`
- `n_structures`
- `['interpolate', 'distances', 'step']`

if `'interpolate'`

- `start_nodes`
- `end_nodes`
- `stepsize_nodes`

if `'distances'`

    - r_cut_off
    - number_bins_power
    - list_of_bins[...]

By the argument --grid, first the name of the precalculated training grids is given and secondly the number of structures which will be taken from it. Thirdly, the grid type has to be defined, which can be 'interpolate', 'distances' or 'step'. For 'step' potentials no further parameters have to be defined. The grid type 'interpolate' is for grids which contain the sum of Lagrange functions on the nodes for an interpolation of saddle point functions. For this type, one has to define the position of the first node, the position of the last and the stepsize between them. The parameter 'distances' is used for grids which contain the sum over $1/r^z$ for different powers of z. Due to the fact that these sums can be calculated in different ranges, r_cut_off defines the number of range-bins which will be summed up for a total distance cutoff. Thereby, any integer cut off can be generated by summing up the contents for example in the ranges between 0 to 7 Å. Secondly, the number of bins with sums to different powers is given. That is usually 2 for a normal LJ-potential but also potentials which use further terms can be created by MC annealing. For a LJ-potential form, a grid which contains the sum over $1/r^{12}$ in the fist bin and over $1/r^6$ in the fourth, the argument looks as follows.

--grid ··· r_cut 2 1 4

--mcparams

- Δp_type('normal', 'adaptive')
- Δp_size
- MC_steps
- targetfunction('rank', 'refine', 'positiontop', 'simple', 'positionlinear', 'positionquadratic', 'refine-positionlinear')

if 'rank', 'refine', 'positiontop', 'refine-positionlinear'

    - number_subset

- Annealing_function('exponential', 'log', 'linear', 'ziczac')
- start_temperature
- mc_type('interpolate', 'normal', 'keepsign')

if 'interpolate'

    - functiontype
    - power1

- power2

if 'normal', 'keepsign'

- parameter_change('normal', 'saddle')

if 'saddle'

- power1
- power2

The parameters provided with `--mcparams` determine the way the direct Monte Carlo search is executed. The type of parameter change Δ`p_type` can be `'normal'` or `'adaptive'`. `'normal'` changes a scoring parameter at each step by a given size Δ`p_size` whereas the `'adaptive'` scheme adapts the stepsize Δ`p_size` to the temperature by the factor t/T_0. `MC_steps` determines the number of Monte Carlo steps, that is equivalent to the number of parameter changes and rescorings.
The `targetfunction` determines the criteria for the optimization of the scoring functions parameters. The `'rank'` and the `'refine'` function builds the sum over `number_subset` structures on top for each complex but the `'rank'` function gives these an extra linear weight which decreases from rank 1 to `number_subset`.
The `'simple'` function takes the rank of the the best scoring near-native structure as its value for the targetfunction.
For `'positoinlinear'` and `'positionquadratic'` each position receives a weight ω_{pos} which decreases linearly and quadratic respectively. The `'positiontop'` function assigns also increasing linear weights to each position but just until the structure reaches the `number_subset` position. From that rank to the top, the same weight is assigned. The target function `refine-positionlinear` multiplies the value from the target function `refine` with the value from `'positoinlinear'`.
In these cases, the near-native structures possess a weight $\omega_{quality}$ dependent on their quality whereas this weight is 0 for the incorrect structures. These weights are given by the values of the quality assessment. Furthermore, these values are normalized by the sum over them to avoid overfitting on complexes with a lot of near-native structures. The average over all complexes for the sum over the product between these weights is taken as target function which has to be maximized.

$$\tau = \sum_{m}^{M_{complexes}} \sum_{n}^{N_{structures}} \omega_{pos}(rk(E_n)) \cdot \omega_{quality}^{(n)}$$

The `Annealing_function` defines the way the temperature is cooled down from the starting temperature `start_temperature`. The annealing functions can be whether `'exponential'`, `'linear'` or `'log'` which defines a logarithmic decrease. The `'ziczac'` annealing uses a sinus^2 between two linearly decreasing functions to

heat up the temperature several times to enable the search algorithm to overcome barriers more easily.
The `mc_type` defines how the scoring parameters will be changed. This is dependent on the given grid or potential form. For `'interpolate'` grids only the `'interpolate'` method works. In the library `functionlib.py`, three different `functiontype`s are predefined. Functiontype `'0'` is the normal ATTRACT potential with attractive and repulsive terms depending on the given third parameter i_{vor}. Funtiontypes `'1'` and `'2'` are variations of that ATTRACT potential, increasing the range of the attractive potentials at the minimum by a constant value and putting a repulsive term at the end of the potential respectively. For all three potential shapes, it is important to define `power1` for the first term of the LJ-potential and `power2` for the second. For `mc_type = 'keepsign'`, the scoring parameters are constrained to be larger than zero. This is important to use for the generation of a LJ-potential by the distance grids while using the parameters α and β directly. For `'keepsign'` and `'normal'`, it has to be defined in which parameter space the changes are performed. The `parameter_change = 'normal'` changes the parameter α_{AB} and β_{AB} but by choosing `'saddle'`, the parameter ϵ_{AB} and σ_{AB} got changed. If `'saddle'` is chosen as parameter space, `power1` and `power2` need to be defined for the computation of α_{AB} and β_{AB} out of ϵ_{AB} and σ_{AB}.

`--evaluate`

- `evaluation_values.type`
- `type('fnat', 'irmsd', 'lrmsd', 'capstars' 'probabilities', 'duplication')`

if `'duplication', 'probabilities'`

  - `list_of_probabilities.txt`
  - `n_column`

With the argument `--evaluate` the list of values for the qualitative weighting of the structures $\omega_{quality}$ is given. The second argument defines whether these values are `fnat`'s, `irmsd`'s, `lrmsd`'s, `capstars`'s or `probabilities` based on the quality file. If `'probabilities'` or `'duplication'` is chosen, the name of the `list_of_probabilities.txt` must be given. In that file, the first column consists of evaluation values for which a probability is assigned from the `n_column` column. The parameter `'duplication'` uses the probabilities as weights for the structures, too. Furthermore, it duplicates the structures on their probability to generate at least 10 % of structures for each complex with a probability except from 0.

`--preparameter`

- `startparameter('random', 'saddlepoint', 'ABC')`
- `bins`
- `number_of_parameter`

if `'saddlepoint', 'ABC'`

  - `array_of_parameter.par`

By the argument `--preparameter` the format of the starting parameter `startparameter` is defined. Parameter `'random'` generates a random set of scoring parameters as a starting point for the optimization in the form `(bins, number_of_parameter)`. `bins` determines the number of different parameters, that means for example 3 for ϵ, σ and i_{vor}. The `number_of_parameter` defines the number of scoring parameters for the interaction types between λ atoms types A and B: $n_{AB} = \frac{\lambda(\lambda-1)}{2} + \lambda$. Whereas for a grid with the atomistic buried surface areas, the number parameters would be $n_{AB} = \lambda$.
By the parameters `'saddlepoint'` and `'ABC'`, it is possible to read in a set of scoring parameters from a file. `'saddlepoint'` defines the parameter file to be form of ϵ_{AB}, σ_{AB} and i_vor_{AB} and `'ABC'` is used for parameters in the form α and β.

`--output`

- `output_name`
- `form_of_outputfile('matrix', 'linear')`

For the output file it is necessary to define the arrangement of the parameter. `'matrix'` leads to a matrix of the form (`bins` $\times \lambda \times \lambda$) and `'linear'` to a form (`bins` $\times$ `number_of_parameter`).

In addition to these main arguments, other arguments can be used to improve the results of the training process. By `--maxweight` and `--cutoffweight` the weights $\omega_{quality}$ can be cut off at low values to train only on 'very' good structures and for high weights respectively if overtraining might be coming from them.
Furthermore, `--insertnatives` gives the possibility to insert a grid of one structure for each complex which is supposed to be the native one.
By `--eraseatomtype` it is possible to erase all the entries of the grids for the given atom type due to the possibility of dummy atoms which should not receive parameters.
By the argument `--converge c_steps` it is possible exit the Monte Carlo algorithm before the total number of steps is executed. Therefore, `c_steps` defines after how many steps without a change of the target function, the search is stopped.
For the `spline` and the `distances` grids it might be necessary to constrain the range of the scoring parameters to avoid overfitting. For that reason, the argument

`--searchrange` can be used. As parameters, the lower and the upper boundary for each bin has to be given in their order.
Finally, the program generates files containing the parameters in the linear order with the name `MC-Parameter-'outname'.par`, for the parameters in `matrix` form, the output file is named `MC-Paramatrix-'outname'.par` and for the output in form of the parameter ϵ, σ and i_{vor}, the file is called `'MC-Paramatrix-'outname'_parm.par`. Furthermore, a file containing the development of the targetfunction is created with the name `Annealing-'outname'.txt`.

1.3.2 training-glm.py

The program `training-glm.py` offers the possibility to use various generalized linear models to estimate parameters for a scoring function from linear regression or linear classifications. Therefore, the feature vectors are fitted to a given set of output values. The outcomes have to be characteristics of the quality of the structures. Thus, fnats, irmsds, lrmsds, capstars of fnon-nats can be used as outcome values for the regression or to classify near-natives for a classification.
To enrich the training set with near-native decoys on a given probability (f.e. to become 2-star from 1-star) `duplicatelib.py` is implemented in the program. Thus, duplicatelib.py has to be present in the same folder as the main program. `training-glm.py` can be executed on the terminal by:

```
$python training-glm.py
    --complexes [ ]
    --grid [ ]
    --regressiontype [ ]
    --evaluate [ ]
    --functionshape [ ]
    --output [ ]
```

The parameters which have to be defined for the arguments `--complexes` and `--evaluate` are the same as already described for `training-MC.py`.
Due to the fact that linear regression can only deal with functions which are linear dependent on its parameters, the import of the grids varies slightly from training-MC.py.

`--grid`

- `precalc-grid.npy`
- `n_structures`

- `gridtype('distances', 'step')`

if `'distances'`

  - `sign(keepsign, normal)`
  - `r_cut_off`
  - `number_bins_power`
  - `list_of_bins[...]`

The first parameter for the argument `--grid`, represents the name of the precalculated grids for each complex, followed by the number of decoys which are considered for the training. Thirdly, the type of the grid is given, which can be whether `distances` or `step`. Grids of the form `distances` contain the sum over the distances for each atom type to a defined power and `step` grids contain the number of contacts in each defined step.
For the grids `distances`, it has to be defined which sign is assigned for each bin. The parameter `keepsign` assigns a negative value to the second sum in the potential as in a typical Lennard-Jones potential. Additionally, to generate a Lennard-Jones potential, the `--regressiontype nonneglsq` must be used to generate exclusively positive values for α_{AB} and β_{AB}.
The parameter `r_cut_off` defines the number of bins which are sumed up to generate different cutoffs. `number_bins_power` represents the number of bins which contain sums over the distances to different powers. After defining the number of terms which will be used in the potential (for LJ typically 2), the location of precalculated terms in the grid has to be determined.

The argument `--regressiontype` defines the linear regression or classification algorithm which is used to generate the parameter. The various regression methods perform their optimization on different cost functions based on different underlying models for the data. The characteristics of the listed models will roughly be explained in the following.

`--regressiontype`

(a) `Robustlinearmodels`

(b) `svr-robust`

(c) `ols`

(d) `nonneglsq`

(e) `Ridge`

(f) `Lasso`

(g) `elasticNet`

(h) `RANSAC`

(i) `Bayesianridge`

(j) `logistic`

(k) `SGDClass`

The `Robustlinearmodel` represents a robust regression based on the statsmodel library which can be found under http://statsmodels.sourceforge.net. Instead of the usual assumption of Gaussian noise, the statsmodel library offers the possibility to use different probability models which can be chosen by:

```
--regressionparams regmodel(Huber, Andrew, Hampel, Ramsay, TrimmedMean,
    Tukey)
```

The regression with `svr-robust` represents a support vector regression. Based on the parameters given by `--regressionparams`, its robustness can be adjusted.
In general, the results of some linear regression algorithms may be affected towards unfavoured solutions if outliers exist in the training set. Robust regression methods intend to be less prone towards outliers due to their adapted cost-functions.
The `ols` regression represents the ordinary least squares regression which assumes a Gaussian noise distribution in the data.
Also `nonneglsq` uses an ordinary least squares fit to generate only positive parameters. Both methods are based on the SciPy library for python which can be found under http://www.scipy.org.
The regression and classifier algorithms `Ridge`, `Lasso`, `elasticNet`, `RANSAC`, `Bayesianridge`, `logistic`, `SGDClass` are based on the scikit-learn module which can be found on http://scikit-learn.org.
`Ridge` and `Bayesianridge` regression penalize the size of the scoring parameters to avoid overfitting by including their average value into its cost function.
`Lasso` regression prefers solutions with fewer parameter values and thus may be able to recover the exact set of non-zero weights.
Elastic Networks combine the properties of Lasso and Ridge regression in their cost function and can be used for regression by choosing `elastic`.
`RANSAC` is a robust parameter estimator which uses random subsets of inliers for its predictions. Its result is highly dependent on the number of iterations which can be defined explicitly with `--regressionsparams`.
The regression types `logistic` and `SGDClass` represent a logistic classifier and stochastic gradient descent classifier respectively. The input for the y vectors have to consist of 0's and 1's for each class respectively. For the separation into classes the maximum or minimum value for each class must be given on the second position behind the `--regressiontype`.

In addition, with `--regressionparams` further parameters can be inserted for each regression method to optimize their results. An explanation for the argument `--regressionparams` must be taken from the source code and the manuals for the libraries which were mentioned above.

With the argument `--functionshape`, the `bins` and the number of atom types `n_atomtypes` for the scoring parameters have to be defined. For grids which use the atomistic buried surface area the number of bins must be defined as 0.

`--functionshape`

- `bins`
- `n_atomtypes`

To define the name and the form of the `--output` first the name and then the chosen form has to be given.

`--output outname outform(normal, saddle)`

The form of the output can whether be `normal` or `saddle`. `saddle` means that the fitted parameters α and β will be transformed into ϵ and σ. For the the `saddle` form, the power of the first and the second part of the LJ-potential have to be given for the transformation by `--powers p1 p2`.
Due to the different size and thus diverse average numbers of contacts or buried surface areas for each complex, it might be useful to normalize the feature vectors before using a regression method. With `--preprocessing` the method to normalize the input data can be defined. `Standard` creates a Gaussian distribution of contacts for each complex separately. `MinMax` distributes the contacts for each structure between 0 and 1. Finally, `meancomplex` divides the contacts through the mean total number of contacts for each complex.

In addition, further arguments can improve or influence the results of the regressions. For instance `--insertnatives` provides the possibility to insert a grid of one structure for each complex which is supposed to be the native structure or by `--eraseatomtype` it is possible to erase all the entries of the grid for the given atom type due to the possibility of dummy atoms.
The output of the program is a file named `LinReg-Parameter-'outname'.par` which contains the determined scoring parameters. Furthermore, it is possible to check the regression results by `--prediction`. This creates two files which contain the real and the predicted values for the training and the test naming them `LinReg-Predictions_'set'_'outname'.txt`.

1.3.3 training-nonlinear-classfier.py

Classifiers also make their prediction of classes on an underlying function and a threshold for each class. Using classifiers possesses the advantage that they are able to separate classes of structures non-linearly on its features. To use classifiers as a scoring function, the structures for training must be divided into a class of near-native and incorrect decoys based on a qualitative assessment by fnat, irmsd, lrmsd or Capri-stars. The program `training-nonlinear-classfier.py` provides this possibility. It uses the same arguments as `training-glm.py` to read in the grids and to prepare the classification. The program can be executed by:

```
$python training-nonlinear-classfier.py
    --complexes [ ]
    --grid [ ]
    --classifier [logistic, SVM, SGDclass, gaussianNB]
    --evaluate quality.file qualitycut qualitytype
    --functionshape [ ]
    --output [ ]
```

Just for the argument `--evaluate` an extra parameter has to be defined which divides the structures into a near-native and an incorrect class on the values in the `quality.file`.
The nonlinear classifiers which can be chosen are `logistic`, `SVM`, `SGDclass` and `gaussianNB`. For all four of them, different `--fitparams` can be defined to obtain an optimal result. The exact adjustments for the `--fitparams` must be taken from the source code. The meaning of the adjustments can be looked up on http://scikit-learn.org which contains a description of the classifiers.
The output of these methods is a classifier which is stored in a binary file by the **cPickle** library named `Classpredictor_'classifier-outname.pkl`. This file can be reloaded and used in `grid-rescore.py` for the rescoring of decoy sets.

1.3.4 average-params.py

After creating n different parameter sets for crossvalidation, a sufficient approach to prevent overfitting on any set of complexes might be to average the parameter values. Therefore, `average-params.py` uses the argument `--scores` followed by the number of parameter sets and their names.

```
$python average-params.py --scores n_scores score1 ··· scoren
```

By using the argument `--scaleparams`, the parameters are normalized by the division through their standard deviation. Thereby, the parameter sets with larger scales do not dominate the total average. The standard deviation is calculated for every type of parameter separately and thus the number of bins has to be provided with `--bins`. If bins is set to 0 the parameters are not given in a matrix form but as a linear vector, for example for the buried surface area potentials or the weights of scoring combinations. To declare the name of the output the argument `--output` can be used.

1.3.5 combine_score.py

The program `combine_score.py` can combine scores from different scoring functions linearly and nonlinearly. To determine the weights for the linear combination, it is possible to use Monte Carlo Annealing but also linear regression or linear support vector machines. Furthermore, the program includes the possibility to combine scores nonlinearly by using support vector machines with nonlinear kernels and naive bayesian estimators.

The generated .rescore files are usually generated in the original order. Therefore, the program `structure_resort.py` can be used to sort the rescores of near-native structures in front of the rescore-file to enrich the training set in the parametrization process for the combination parameters. The program can be executed on the terminal by:

```
$python combine_score.py
    --complexes [ ]
    --numstruc [ ]
    --scores [ ]
    --method [bayes, svm, mc regression]
    --yvalue [ ]
    --output [ ]
```

For the prediction of weights, a training set of complexes and the number of sets for crossvalidation has to be defined with as in the single training scripts:

```
--complexes trainingset.txt n_crossval
```

Furthermore, the number and the names of the different scores is given by:

```
--scores n_scores score_0 ··· score_n
```

The files containing the scores must be named `.rescore` if they just contain a list of the pure score or `.dat` when they are in the Attract output format, containing also the degrees of freedom etc. By the argument `--numstruc` the number of decoys for each complex is defined.
The `--method` for the estimation of the weights can be whether `bayes`, `svm`, `mc` or `regression`.
The method `bayes` uses the given `--yvalues` from a file to divide the given structures into two classes 'near-native' and 'incorrect' on the given `cutoff`.

```
--yvalues quality.file cutoff
```

As output the program generates a file named `Combine_parameter_'outname'.pkl` from which the classifier can be reloaded with the library `cPickle` in the program `combine_rescore.py`. Furthermore, it stores the deviation σ and the mean μ of the Gaussian probability distribution for each class in a file named `Combine_parameter_'outname'.par`. From these parameters for each class and feature the probability to be in a certain class can be computed.

For the method `svm`, the `--yvalues` and the `cutoff` has to be provided also to divide the structures into classes. With the `--fitparams` the support vector classifier can further be defined:

`--fitparams`

- `kernel[linear, rbf, polynomial](default=linear)`
- `cachesize[in MB](default=4000)`
- `Ci(default=1.)`
- `calc_prob[bool](default=False)`
- `tolerance(default=0.001)`

The `cachesize` defines how much cache space will be used for the minimization. The parameter `Ci` represents the weight to which strength classification errors contribute to the cost function of the svm. The parameter `calc_prob` ($\in$ `{True,False}`) defines whether a probability distribution is generated for the classes. The probability is derived from the decision function of the support vector machine and serves just as an additional output form. The `tolerance` adjusts the convergence criteria of the minimization algorithm. The `linear` kernel produces a list of parameters for the decision function in the file `Combine_parameter_'outname'.par`. The nonlinear kernels `rbf` and `polynomial` generate a file named `Combine_parameter_'outname'.pkl` which can be reloaded in `combine_rescore.py` to compute the values of their nonlinear decision function.

For the method `mc` the `--yvalues` have to be given and the type of these values must be defined. This type can be `fnat`, `lrmsd`, `irmsd` or `capstars`. The parameters for the annealing algorithm have to be defined behind the argument `--fitparams`:

`--fitparams`

- `targetfunction('rank', 'refine', 'positiontop', 'simple', 'positionlinear', 'positionquadratic', 'refine-positionlinear')`
- `Annealing_function(linear, exponential, ziczac, logarithmic)`
- `start_temperature`
- Δ`p_type(normal, adaptive)`
- Δ`p_size`
- `MC_steps`

The meaning of the parameter for the Monte Carlo annealing algorithm can be looked up in the manual for `training-MC.py`.
With the argument `--meanscale` the different scores can be divided by their mean to adjust the stepsize of the weights for each type of score. Thus, the change of a weight will have the same influence on the combined score for each score. Hence, the parameter space might be sampled more successfully. This division will be respected in the final output of the weights. As output, a parameter file is generated with the name `Combine_parameter_'outname'.par` which can be used directly as weights for the combination of scores. The development of the target function during the annealing is stored in `Combine_Annealing_'outname'.txt`.

For the method `regression`, only the name of the file containing qualitative evaluations has to be provided behind the argument `--yvalues`. Fnat, irmsd, lrmsd and capstars can be used. The fit is performed of the negative values of the fnat and the capstars or negative reciprocal value of the irmsd and lrmsd.
In addition, the `--regressiontype` has to be defined, which can whether be `BayesianRidge`, `ols`, `Ridge` or `svr-robust`. A short description for these types of regression can be found in the manual of `training-glm.py` or in more detail on http://scikit-learn.org.
As mentioned above, the different sizes and chemical compositions of the proteins in all complexes can make a fitting difficult due to the fact that the mean scores will deviate. Therefore, the scores can be preprocessed by the argument `--preprocessing [MinMax, complexmean, Standard]`. The preprocessing type `Standard` distributes the scores for each complex in a Gaussian with a mean 0 and a standard deviation

of 1. `MinMax` distributes the scores between 0 and 1 and `complexmean` divides the scores for each complex by their mean value.

1.4 Rescoring

1.4.1 grid-rescore.py

The program `grid-rescore.py` can be used for fast re-scoring on the precalculated grids. Due to the fact that only simple multiplications have to executed, the rescoring on grids can be performed much faster than by `@rank.py`. Nevertheless, time delays can occur due to the size of the grids which have to be uploaded into the memory, especially when the program is executed many times on a cluster.
The program is executed by:

```
$python grid-reevalutation.py pregrid.npy parameter.par
```

`--gridtype`

- `gridtype [interpolate, distances, step, nonlinear]`
- `bins`
- `atomtypes`

if `'distances'`

- `r_cut_off`
- `number_bins_power`
- `list_of_bins[...]`

if `'interpolate'`

- `start`
- `end`
- `stepsize`
- `functiontype`
- `power1`
- `power2`

The grid type `nonlinear` refers to the possibility of using a nonlinear classifier like svm or naive bayes on a 1-step grid also referred to as contact grid. The parameter file for these grids is supposed to be binary file which contains the classifier from the program `training-nonlinear-classifier.py`.
If `distances` grids are used, the number of the bins which are summed up for the

cutoff have to be given first. Secondly, the number of the bins which contain the sums over distances to different powers and their location have to be defined as described in the training programs.
For the `interpolate` grids, the `start`, the `end` and the `stepsize` for the placement of the nodes has to be given. Furthermore, the `functiontype` and the power of the first and the second part of the potential have to be defined. The functiontypes were explained in `training-MC.py`, 0 stands for a normal Attract shaped saddle point potential.
For grids which use the atomistic buried surface area, the bins must be defined as 0. The program simply prints out the rescores. For further use of these scores for example for combination or the assessment, the files names must end on **.rescore**.

1.4.2 @rank.py

For the rescoring of decoys without precalculated grids, the program `@rank.py` is able to score structures by a LJ-potential (saddlepoint), a step potential, a potential based on the atomistic surface area, a coulomb interaction, the buried surface area and even on a combination of these scores. The combination can be linear by given weights or nonlinear by the use of a classifier. The program uses the Fortran library `scorelib.f` which has to be compiled with **f2py** in advance. The program is executed by:

```
$python @rank.py
    --input coordinate.dat
    --proteinmodel model receptor.pdb ligand.pdb
    --vdwpotential vdwparameter.par
    --steppotential stepparas.par --bins n_bins range_0 ··· range_n
    --solvation solparameter.par
    --electrostatics
    --buriedsa
    --combination method combinationweights.par
```

The degrees of freedom for each decoy are given by the ATTRACT output file `coordinate.dat` which is inserted by `--input`. As in ATTRACT, normal modes, ensembles and refined structures can be inserted by `--modes`, `--ens` and --name. Each scoring type can be chosen separately or in combination with other functions. If no combination method is given, the scores are just summed up.
The model for the given pdb's of the protein constituents can be `opls`, `attract`, `tobi`, `gaa` or `any`. The models `opls`, `attract`, `tobi` and `gaa` reduce the atom types to the total number of atom types (for opls f.e. 13). The model `any` uses as its

maximum number of atom types the input behind the ligand.pdb. To use `any`, the parameter file must contain as many columns and rows as the maximum atom type.

For the scoring by models for van der Waals interactions the parameter file has to be given by the argument `--vdwpotential`.
A `--shift` can be defined which sets every distance between two atoms at least to that value. By shifting the distances to a certain value, clashes between atoms will be avoided. Clashes might result from the change of a coarse grained to an atomistic representation after sampling. In addition, a cutoff can be defined with `--vdwcutoff` which is by default set to 100.
For the van der Waals potentials, the `--vdwfunctiontype` must be chosen `attract`, `opls` or `free`. `attract` uses a saddlepoint potential with the powers 8 and 6 whereas `opls` uses 12 and 6. Any other potential with other powers can be given by `free power1 power2`.
The parameter file for the step potentials is provided by the argument `--steppotential`. The bins of the step potential have to be defined by `--bins` with the number of bins and their ranges.
The `--solvation` potential uses its parameter file for the computation of a score by the atomistic bsA's. The radius of the probe can be changed with `--watersize`. For the coarse grained representation of `attract`, coarse grained van der Waals radii may be used with `--coarse_grained`.
The `--electrostatics` term uses the charges in the pdb files to calculate a Coulomb energy between the proteins.
The `--buriedsa` calculates the negative buried surface area. As for the solvation term, the `--coarse_grained` vdw-radii can be used and the radius of the probe can be changed by `--watersize`.

The weights for linear combinations or the classifier to combine the different scores are given by `--combination`. The `method` must be defined first. This can be `linear` to combine the scores linearly by the determined weights, `probability` to estimate the probability for each structure to be in a class, or `decisionfunction` to use the distance from the boundary between classes from svm as a score.
For all nonlinear combinations but also for some linear combinations from linear regression, the normalization of the scores is necessary before combination. Therefore, `--preprocessing` can be used. The `method` can be `Standard` for a Gaussian distribution or `MinMax` for a distribution between 0 and 1 for each score.
The argument `--printout` serves to print out the final combined scores directly. `--rescore` gives out a file ending on `-rescore.dat` which contains each score and the combined score in the Attract file format in the orignal order of the Attract dat-file. `--rerank` creates such a file ending on `-resorted.dat` in which the structures

are resorted after their new score.
A new name for the output can be set by `--output`.

1.4.3 combine_rescore.py

The program `combine_rescore.py` combines the given scores from files with a given set of weights linearly or nonlinearly by trained classifiers. The program is executed on the terminal by:

```
$python combine_rescore.py method weights.par
    --scores n_scores s1 ··· sn
```

To receive the correct result, the scores must be in the same order as they were used when the weights were generated. Usually the order of the scores can be taken from the header of the file which contains the weights. The method to combine the scores is defined as `linear` if weights are used or `nonlinear` if a binary file with a classifier is loaded. For many methods it is necessary to normalize the scores before combination, therefore three methods can be chosen with `--preprocessing`. `Standard` makes a Gaussian distribution, `MinMax` scales the scores between 0 and 1 and `meancomplex` divides them by their mean. The name of the output file can be defined with the argument `--output`.

1.5 Assessment of the Performance and Characteristics of Scoring Functions

1.5.1 scoring_assessment.py

To check out whether a new scoring approach was successful, fast rescoring and a performance assessment is necessary. The program `scoring_assessment.py` takes the scores and a file for the quality assessment of the decoys to evaluate the performance on the whole benchmark or the training set and the test set. The program can be executed on the terminal by:

```
$python scoring_assessment.py
    --scores n_scores s1 ··· sn
    --evaluate quality.file qualitycut
    --benchmark benchmark.file n_cross
    --classification classtype classification.txt
    --atleast
```

`--averages`

`--ROCcurve`

`--plotstructures`

`--ROCstructures`

The files containing the scores can whether be files ending on `.rescore` if they contain only the pure score or files in the Attract output format `.dat`. The names of the scores are given by the argument `--scores` followed by the number of scores `n_scores` and their file names.
The qualitative evaluations of the structures are given by using the argument `--evaluate`. `qualitycut` defines the border to divide the structures into near-native and incorrect solutions based on the values of the `quality.file`.
For the evaluation, the structures of each complex are sorted after each score separately. Finally, a binary file is created for each score which contains a step function showing the number of near-native structures in the set for each rank. By this procedure, the evaluation must only be performed once for each score and the time consuming import each time the score is compared can be avoided.

By providing a `--benchmark`, the output windows will be divided into figures for the training and the test set. The number `n_cross` defines how the benchmark is divided. If `n_cross=1` the whole list of complexes will be taken as the training set and all other folders (complexes) which are not listed are taken as the testset. For any other value, the last $1/n\_cross$ fraction of the complexes in the list and the remaining directories in the executive folder are taken as the test set.
With `--classification` the file `classification.txt` can be used to sort the complexes after their `difficulty` or their `proteintype` which has to be defined as the classtype. If `--benchmark` and `--classification` are chosen, the program sorts the complexes after their sets but divides the bars in the `--barchart` representation into classes.

To visualize the performances, various modi can be chosen. With `--Plotstructures` the step curves for each complex can be regarded. Switching between the complexes can be done by a slider. The modi `--atleast` plots the fraction of complexes for which a near-native complex can be found against its ranks. `--averages` plots the average number of near-natives in the decoy set against their rank. Furthermore, it is possible to use `--ROCcurve` to plot the fraction of near-natives against the fraction of incorrect structures. The same can also be done for each complex by `--ROCstructures`.
The argument `--%` causes that the average fraction of near-natives is plotted instead of the average number for the figure from `--averages`. By using `--double%` the

rank on the abscissa in `--atleast` and `--averages` will be changed into the fraction of decoys in the set.
The argument `--barchart` creates a bar-chart for the figures of `--atleast` and `--averages`. The bar-chart shows the values for four ranks and fractions of decoys respectively. Usually, the names for the scores in the legend are taken from their files. Changing these names can be done by `--names name_0 ··· name_n`.

1.5.2 scoring_native_assessment.py

The program `scoring_native_assessment.py` plots the fraction of complexes for which the native structure was found against its rank in the decoy set. Therefore, the file's names of the scores of the decoys are given by the argument `--scores`. For the evaluation, a file containing only the native score must be created and named after the file which contains the scores of the decoy set, supplemented by the ending `-native.rescore`.

```
$python scoring_native_assessment.py

    --scores n_scores s1 ··· sn

    --benchmark benchmark.file n_cross

    --classification classtype classification.txt
```

As in `scoring_assessment.py` the plots can be divided into a test and a training set by the usage of the argument `--benchmark` or can be divided into classes by the argument `--classification`. Just as in `scoring_assessment.py` the values on the abscissa can be changed from the absolute rank into the fraction of all decoys by `--double%`. As well, it is possible to use `--barchart` to plot the figure as a bar-chart for four positions on the abscissa instead as a step function.

1.5.3 correlation_assessment.py

The program `correlation_assessment.py` evaluates correlations between different scoring function by comparing sets of well scored structures or by comparison of their ranking in the decoy set. Therefore, the names for the files with the scores of the structures have to be provided by `--scores`. Also a file containing a qualitative evaluation to define near-native structures has to be defined with the argument `--evaluate`.

```
$python correlation_assessment.py

    --scores n_scores s1 ··· sn

    --evaluate quality.file qualitycut
```

```
--benchmark benchmark.file n_cross
--classification classtype classification.txt
--Rankerror
--Rankcorr
--topcorr topfraction toptype
--symdifference
--union
--complement
```

The structures are divided by the `quality.file` into near-natives and incorrect solutions and sorted by their scores. The value for `qualitycut` defines the border between near-native and incorrect solutions. Using `--Rankcorr` the correlation of the ranks of the structures between the scoring functions is generated. `--Rankerror` generates a matrix of the mean error between the ranks of the scoring functions. Both matrices show the correlations between scoring functions based on the ranking of the structures.
To analyse the correlation on the well scored structures for each scoring function, subsets of the best scored structures can be compared by `--topcorr`. For that reason, the fraction of decoys which will be regarded has to be defined by `topfraction` ($\in [0., 1]$). Moreover, it has to be determined which structures will be regarded in the subsets, the `good`, the `bad` or `all`. For the chosen set of structures, a matrix of the `--union`, the `--symdifference` and the `--complement` between the sets from all scores is created.

1.5.4 contacts-evaluation.py

The program `contacts-evaluation.py` is able to evaluate the distribution of contacts in for step or contact potentials as well as the distribution of atomistic buried surface areas. In addition the program can use the average number of contacts to evaluate the average contribution of each parameter to the overall scoring performance. The program is executed by:

```
$python contacts-evaluation.py
    --grid [ ]
    --benchmark [ ]
    --evaluate [ ]
    --normcontacts
    --deletezeros
```

```
--nativestructure [ ]
--namelist [ ]
--plotcontact
--plotcontactdist numcontacts
--plotparametercontribution params.par numcontacts
--plotparameter params.par numcontacts
```

The program uses the precalculated grids with the number of contacts or the buried surface areas through the argument `--grid`. A `--benchmark` can be defined, if the evaluation should not be executed for the whole benchmark but for certain structures. The whole benchmark will be taken with the argument `--benchmark benchmark`. Since the program evaluates contacts for false and near-native solutions, `--evaluate` defines the quality assessment which will be used to divide the structures on the cut off given on the second position behind the argument. Because of the various sizes of the complexes and their various average number of contacts, one can normalize the contacts of each complex by their total average number by `--normcontacts`. `--deletezeros` eliminates complexes from the evaluation which do not possess a near-native structure in their decoy set. With the argument `--naitvestructure` the name of the grid for native structures can be given. From these grids a contact analysis will be performed for native structures separately. To label the contacts, a list of names for all the atom or coarse-grained bead types can be passed with the argument `--namelist`.
Using the argument `--norm` the contacts of the near-native and the native structures are normed to the contacts of the false solutions for a better comparison. The argument `--plotcontact` plots the average number of contacts for false, near-native and native structures sorted after the averages of near-native solutions. `--plotcontactdist` only plots the best and worst `numcontacts` contacts sorted after the difference between contacts of near-native and false solutions. Through the argument `--plotparametercontribution` a set of parameters can be inserted to multiply them by their average number of contacts and sort them afterwards by the largest difference between near-native and false solutions to plot the best and the worst `numcontacts` contributions. Through the multiplication of each parameter with its average numbers of contacts, the contribution of each parameter can be better evaluated since the distribution of atoms and contacts varies from their type. By sorting the parameter contributions by the contribution's difference to the near-native structures, a feeling of the most differentiating parameters can be gained.
Furthermore, only the parameter can be plotted without any normalization by `--plotparameter`. Nevertheless, also here `--normparameter` derives a plot of the contributions by multiplying them with the average number of their contact type for

false solutions and `--normparameter2` by multiplying them by the difference between the averages of false and near-native contacts.

Bibliography

[1] Jorgensen, William L. ; Tirado-Rives, Julian.: The OPLS [optimized potentials for liquid simulations] potential functions for proteins, energy minimizations for crystals of cyclic peptides and crambin. In: *Journal of the American Chemical Society* 110 (1988), Nr. 6, 1657-1666. http://dx.doi.org/10.1021/ja00214a001. – DOI 10.1021/ja00214a001

[2] Lensink, MF ; Mendez, R ; Wodak, SJ: Docking and scoring protein complexes: CAPRI 3rd Edition. In: *Proteins* 69 (2007), Nr. 4, S. 704–718. http://dx.doi.org/10.1002/prot.21804. – DOI 10.1002/prot.21804

[3] Tobi, D: Designing coarse grained-and atom based-potentials for protein-protein docking. In: *BMC Struct Biol* 10 (2010), S. 40. http://dx.doi.org/10.1186/1472-6807-10-40. – DOI 10.1186/1472–6807–10–40

[4] Vajda, S ; Hall, DR ; Kozakov, D: Sampling and scoring: A marriage made in heaven. In: *Proteins* (2013). – doi: 10.1002/prot.24343

[5] Zacharias, Martin: Protein–protein docking with a reduced protein model accounting for side-chain flexibility. In: *Protein Science* 12 (2003), Nr. 6, 1271–1282. http://dx.doi.org/10.1110/ps.0239303. – DOI 10.1110/ps.0239303. – ISSN 1469–896X