ADG - Automated Diagram Generator Documentation

Release 3.1.0

ADG Dev Team

Aug 09, 2023

CONTENTS

CHAPTER

ONE

THE ADG PROJECT

1.1 Description

ADG is a tool generating diagrams and producing their expressions for given many-body formalisms. Diagrammatic rules from the formalism are combined with graph theory objects to produce diagrams and expressions in a fast, simple and error-safe way.

The only input consists in the theory and order of interest, and the N-body character of the operators of interest. The main output is a LaTeX file containing the diagrams, their associated expressions and additional informations that can be compiled into PDF by ADG if needed. Other computer-readable files may be produced as well.

1.2 Status

As for now, the code is capable of handling four different formalisms, i.e. Many-Body Perturbation Theory (MBPT), Bogoliubov Many-Body Perturbation Theory (BMBPT), Projected Bogoliubov Many-Body Perturbation Theory (PBMBPT), and Bogoliubov In-Medium Similarity Renormalization Group (BIMSRG).

- For MBPT, the code generates all Hartree-Fock energy diagrams at any given order along with their expression and additional information (conjugate diagram, excitation level…).
- For (P)BMBPT, the code generates all diagrams for a generic observable commuting with the Hamiltonian, along with their time-dependent and time-integrated expressions.
- For BIMSRG, the code generates all diagrams and expressions at any given truncation order for the two operators as well as the commutator itself. The traditional BIMSRG(n) truncation order corresponds to truncating both operators as well as the commutator at the same rank.

1.3 Future developments

ADG is currently being extended to other many-body methods.

1.4 Deprecation warning

In the upcoming major version of ADG, we will be discontinuing support for Python versions prior to 3.7. Please consider moving to newer versions of the language.

CHAPTER

TWO

INSTALL ADG ON YOUR COMPUTER

2.1 Install

To install ADG, download the source files and run

```
pip install <project_folder>
```

or alternatively

```
python setup.py install
```

If you want to install ADG in `develop` mode, then run

```
pip install -e <project_folder>
```

2.2 Dependencies

In order to run the code, you will need a Python2 install >= 2.7.1 and the following Python libraries:

- networkx >= 2.0
- numpy
- scipy
- future
- more-itertools

If you want ADG to compile the LaTeX output file, you will need a Latex install with the PDFLaTeX compiler and the feynmp and feynmp-auto packages installed, which are standard packages in most recent distributions.

CHAPTER

THREE

GENERATE DIAGRAMS WITH ADG

3.1 Run ADG

To run the program and generate BMBPT diagrams at order 4 for example, use

```
adg -o 4 -t BMBPT -d -c
```

where the `-o` flag is for the order, `-t` for the type of theory, `-d` indicates you want the diagrams to be drawn and `-c` that you want ADG to compile the LaTeX output.

You can alternatively run the program in interactive mode by typing

```
adg -i
```

Finally, to obtain more information on all the available flags, use

```
adg -h
```

3.2 CLI options

3.2.1 Generic options

-o, --order order of the diagrams [1-9] or N_A, N_B, N_C truncation for BIMSRG

-t, --theory theory of interest: MBPT, BMBPT or PBMBPT

-i, --interactive execute ADG in interactive mode

3.2.2 (P)BMBPT options

-can, --canonical consider only canonical diagrams

-nobs, --nbody_observable maximal n-body character of the observable [1-3], default = 2

-3NF, --with_3NF use two and three-body forces for BMBPT diagrams

-dt, --draw_tsds draw Time-Structure Diagrams

3.2.3 MBPT option

-cd, --cd_output produce computer-readable output for automated frameworks

3.2.4 Run management options

-d, --draw_diags draw the diagrams using FeynMF

-c, --compile compile the LaTeX output file with PDFLaTeX

3.3 Output files

The output of the program is stored in a folder named after the theory, and a subfolder named after the order, e.g. `/MBPT/Order-4`. In the case of (P)BMBPT, suffixes are added depending on the n-body forces of the observable, and if three-body forces were used or only canonical diagrams computed, i.e. for our previous example, results would be stored under `BMBPT/Order-4_2body_observable`. For BIMSRG, the folder corresponds e.g. to `BIMSRG/Order_1_2_3` if N_A is 1, N_B is 2 and N_B is 3.

The main output file of the program, called `result.tex`, is a LaTeX file containing the expressions of the diagrams along other basic infos on their structure, and, if flag `-d` has been used, drawing instructions. The file is automatically compiled and produces a PDF file `result.pdf` when using the `-c` file.

A list of the adjacency matrices associated with the diagrams is printed separately in the `adj_matrices.list` file to allow for an easy use with another many-body diagrams code.

In the case of a MBPT calculations, it is possible to produce output specifically tailored for automated calculations framework by using the `-cd` flag. The associated output files use `CD_` as a prefix.

CHAPTER

FOUR

ADG REFERENCE FOR DEVELOPERS

4.1 Main script

Main routine of the Automated Diagram Generator.

`adg.main.main()`

Launch the ADG program.

4.2 Run & CLI management

Routines handling the run of ADG.

`adg.run.attribute_directory(commands)`

Create missing directories and return the working directory.

Parameters
commands (*Namespace*) – Flags for the run management.

Returns
Path to the result folder.

Return type
str

```
>>> com = argparse.Namespace()
>>>
>>> com.theory, com.order = 'BMBPT', 4
>>> com.with_3NF, com.nbody_observable, com.canonical = False, 2, False
>>>
>>> attribute_directory(com)
'BMBPT/Order-4_2body_observable'
>>>
>>> com.theory, com.order = 'BMBPT', 5
>>> com.with_3NF, com.nbody_observable, com.canonical = True, 3, False
>>>
>>> attribute_directory(com)
'BMBPT/Order-5_3body_observable_with3N'
>>>
>>> com.theory, com.order = 'MBPT', 3
>>> com.with_3NF, com.nbody_observable, com.canonical = False, 2, False
>>>
>>> attribute_directory(com)
'MBPT/Order-3'
>>>
>>> com.theory, com.order = 'BIMSRG', (1,2,3)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> attribute_directory(com)
'BIMSRG/Order_1_2_3'
```

adg.run.**clean_folders**(*directory*, *commands*)

Delete temporary files and folders.

Parameters

- **directory** (*str*) – Path to the output folder.
- **commands** (*Namespace*) – Flags to manage the program's run.

adg.run.**compile_manager**(*directory*)

Compile the program's LaTeX ouput file.

Parameters
directory (*str*) – Path to the ouput folder.

adg.run.**create_feynmanmp_files**(*diagrams*, *theory*, *directory*, *diag_type*)

Create and move the appropriate feynmanmp files to the right place.

Parameters

- **diagrams** (*list*) – The studied diagrams.
- **theory** (*str*) – Name of the theory of interest.
- **directory** (*str*) – Path to the result folder.
- **diag_type** (*str*) – Type of studied diagrams used for drawing.

adg.run.**generate_diagrams**(*commands*, *id_generator*)

Return a list with diagrams of the appropriate type.

Parameters

- **commands** (*Namespace*) – Flags for the run management.
- **id_generator** (UniqueID) – A unique ID number generator.

Returns
All the diagrams of the appropriate Class and order.

Return type
list

adg.run.**get_bimsrg_truncation_order**(*operator*)

Return the truncation order of a given operator from the user input.

Parameters
operator (*str*) – The letter corresponding to the operator name.py

Returns
The truncation rank of the operator.

Return type
int

adg.run.**interactive_interface**(*commands*)

Run the interactive interface mode, return the appropriate commands.

Parameters
commands (*Namespace*) – Flags for the run management.

Returns
Flags initialized through keyboard input.

Return type
Namespace

`adg.run.order_diagrams`(*diagrams*, *commands*)

Return the ordered unique diagrams with a dict of numbers per type.

Parameters

- **diagrams** (*list*) – The diagrams of the appropriate Class.
- **commands** (*Namespace*) – Flags for the run management.

Returns
First element is the list of ordered and unique diagrams. Second element is a dict with the number of diagrams per type. Third element is a dict with the identifiers of diagrams starting each output file section.

Return type
tuple

`adg.run.parse_command_line`(*cli_args*)

Return run commands from the Command Line Interface.

Parameters
cli_args – Command-line arguments submitted with the program.

Returns
Appropriate commands to manage the program's run.

Return type
Namespace

`adg.run.prepare_drawing_instructions`(*directory*, *commands*, *diagrams*, *diagrams_time*)

Write FeynMP files for the different diagrams.

Parameters

- **directory** (*str*) – Path to the output folder.
- **commands** (*Namespace*) – Flags for the run management.
- **diagrams** (*list*) – All the diagrams of interest.
- **diagrams_time** (*list*) – All the associated TSDs if appropriate.

`adg.run.print_diags_numbers`(*commands*, *diags_nbs*)

Print the number of diagrams for each major type.

Parameters

- **commands** (*Namespace*) – Flags for the run management.
- **diags_nbs** (*dict*) – The number of diagrams for each major type.

`adg.run.write_file_header`(*latex_file*, *commands*, *diags_nbs*)

Write the header of the result tex file.

Parameters

- **latex_file** (*file*) – LaTeX output file of the program.
- **commands** (*Namespace*) – Flags to manage the program's run.
- **diags_nbs** (*dict*) – Number of diagrams per major type.

4.3 Generic Diagram

Routines and class for all types of diagrams, inherited by others.

`class adg.diag.Diagram(nx_graph)`

Bases: `object`

Describes a diagram with its related properties.

Parameters
nx_graph (*NetworkX MultiDiGraph*) – The graph of interest.

degrees

The ascendingly sorted degrees of the graph vertices.

Type
tuple

graph

The actual graph.

Type
NetworkX MultiDiGraph

io_degrees

The sorted version of unsort_io_degrees.

Type
tuple

max_degree

The maximal degree of a vertex in the graph.

Type
int

tags

The tag numbers associated to a diagram.

Type
list

unsort_degrees

The degrees of the graph vertices

Type
tuple

unsort_io_degrees

The list of in- and out-degrees for each vertex of the graph, stored in a (in, out) tuple.

Type
tuple

write_graph(*latex_file*, *directory*, *write_time*)

Write the graph of the diagram to the LaTeX file.

Parameters

- **latex_file** (*file*) – The LaTeX ouput file of the program.
- **directory** (*str*) – Path to the result folder.
- **write_time** (*bool*) – (Here to emulate polymorphism).

`adg.diag.`**`bimsrg_diagram_internals`**(*graph*, *fmf_file*, *prop_type*)

Write to file the vertices and propagators of BIMSRG diagrams.

Parameters

- **graph** (*NetworkX MultiDiGraph*) – The graph to be drawn.
- **fmf_file** (*file*) – The FeymanMF file to be written to.
- **prop_type** (*str*) – The FeymanMF type for drawing the propagators.

`adg.diag.`**`check_vertex_degree`**(*matrices*, *three_body_use*, *nbody_max_observable*, *canonical_only*, *vertex_id*)

Check the degree of a specific vertex in a set of matrices.

Parameters

- **matrices** (*list*) – Adjacency matrices.
- **three_body_use** (*bool*) – True if one uses three-body forces.
- **nbody_max_observable** (*int*) – Maximum body number for the observable.
- **canonical_only** (*bool*) – True if one draws only canonical diagrams.
- **vertex_id** (*int*) – The position of the studied vertex.

```
>>> test_matrices = [numpy.array([[0, 1, 2], [1, 0, 1], [0, 2, 0]]),        ␣
↪numpy.array([[2, 0, 2], [1, 2, 3], [1, 0, 0]]),         numpy.array([[0, 1, 3],
↪ [2, 0, 8], [2, 1, 0]])]
>>> check_vertex_degree(test_matrices, True, 3, False, 0)
>>> test_matrices
[array([[0, 1, 2], [1, 0, 1], [0, 2, 0]]),
 array([[2, 0, 2], [1, 2, 3], [1, 0, 0]])]
>>> check_vertex_degree(test_matrices, False, 2, False, 0)
>>> test_matrices
[array([[0, 1, 2], [1, 0, 1], [0, 2, 0]])]
```

`adg.diag.`**`create_checkable_diagram`**(*pbmbpt_graph*)

Return a graph with anomalous props going both ways for topo check.

Parameters
pbmbpt_graph (*NetworkX MultiDiGraph*) – The graph to be copied.

Returns
Graph with double the anomalous props.

Return type
NetworkX MultiDiGraph

`adg.diag.`**`draw_diagram`**(*directory*, *result_file*, *diagram_index*, *diag_type*)

Copy the diagram feynmanmp instructions in the result file.

Parameters

- **directory** (*str*) – The path to the output folder.
- **result_file** (*file*) – The LaTeX ouput file of the program.
- **diagram_index** (*str*) – The number associated to the diagram.
- **diag_type** (*str*) – The type of diagram used here.

`adg.diag.`**`extract_denom`**(*start_graph*, *subgraph*)

Extract the appropriate denominator using the subgraph rule.

Parameters

- **start_graph** (*NetworkX MultiDiGraph*) – The studied graph.

- **subgraph** (*NetworkX MultiDiGraph*) – The subgraph used for this particular denominator factor.

Returns
The denominator factor for this subgraph.

Return type
str

adg.diag.**feynmf_generator**(*graph*, *theory_type*, *diagram_name*)

Generate the feynmanmp instructions corresponding to the diagram.

Parameters

- **graph** (*NetworkX MultiDiGraph*) – The graph of interest.
- **theory_type** (*str*) – The name of the theory of interest.
- **diagram_name** (*str*) – The name of the studied diagram.

adg.diag.**label_vertices**(*graphs_list*, *theory_type*, *switch_flag*)

Account for different status of vertices in operator diagrams.

Parameters

- **graphs_list** (*list*) – The Diagrams of interest.
- **theory_type** (*str*) – The name of the theory of interest.
- **switch_flag** (*int*) – When to switch A and B operators for BIMSRG.

adg.diag.**no_trace**(*matrices*)

Select matrices with full 0 diagonal.

Parameters
matrices (*list*) – A list of adjacency matrices.

Returns
The adjacency matrices without non-zero diagonal elements.

Return type
list

```
>>> test_matrices = [[[0, 1, 2], [2, 0, 1], [5, 2, 0]],      [[2, 2, 2], [1, 2,␣
↪3], [0, 0, 0]],      [[0, 1, 3], [2, 0, 8], [2, 1, 0]]]
>>> no_trace(test_matrices)
[[[0, 1, 2], [2, 0, 1], [5, 2, 0]], [[0, 1, 3], [2, 0, 8], [2, 1, 0]]]
```

adg.diag.**print_adj_matrices**(*directory*, *diagrams*)

Print a computer-readable file with the diagrams' adjacency matrices.

Parameters

- **directory** (*str*) – The path to the output directory.
- **diagrams** (*list*) – All the diagrams.

adg.diag.**prop_directions**(*vert_distance*, *nb_props*)

Return a list of possible propagators directions.

Parameters

- **vert_distance** (*int*) – Distance between the two connected vertices.
- **nb_props** (*int*) – Number of propagators to be drawn.

Returns
Propagators directions stored as strings.

Return type
list

`adg.diag.`**`propagator_style`**(*prop_type*)

Return the FeynMF definition for the appropriate propagator type.

Parameters
prop_type (*str*) – The type of propagators used in the diagram.

Returns
The FeynMF definition for the propagator style used.

Return type
str

`adg.diag.`**`self_contractions`**(*graph*)

Return the instructions for drawing the graph's self-contractions.

Parameters
graph (*NetworkX MultiDiGraph*) – The graph being drawn.

Returns
FeynMF instructions for drawing the self-contractions.

Return type
str

`adg.diag.`**`to_skeleton`**(*graph*)

Return the bare skeleton of a graph, i.e. only non-redundant links.

Parameters
graph (*NetworkX MultiDiGraph*) – The graph to be turned into a skeleton.

Returns
The skeleton of the initial graph.

Return type
NetworkX MultiDiGraph

`adg.diag.`**`topologically_distinct_diagrams`**(*diagrams*)

Return a list of diagrams all topologically distinct.

Parameters
diagrams (*list*) – The Diagrams of interest.

Returns
Topologically unique diagrams.

Return type
list

`adg.diag.`**`update_permutations`**(*comp_graph_perms*, *comp_graph_tag*, *mapping*)

Update permutations associated to the BMBPT diags for a shared TSD.

Parameters

- **comp_graph_perms** (*dict*) – Permutations to be updated.
- **comp_graph_tag** (*int*) – The tag associated to the TSD configuration.
- **mapping** (*dict*) – permutations to go from previous ref TSD to new one.

`adg.diag.`**`vertex_positions`**(*graph*, *order*)

Return the positions of the graph's vertices.

Parameters

- **graph** (*NetworkX MultiDiGraph*) – The graph of interest.

- **order** (*int*) – The perturbative order of the graph.

Returns
The FeynMP instructions for positioning the vertices.

Return type
str

4.4 MBPT diagram

Routines and class for Many-Body Perturbation Theory diagrams.

`class adg.mbpt.MbptDiagram(mbpt_graph, tag_num)`

Bases: *Diagram*

Describes a MBPT diagram with its related properties.

Parameters

- **mbpt_graph** (*NetworkX MultiDiGraph*) – The actual diagram.
- **tag_num** (*int*) – The tag number associated to the graph.

`adjacency_mat`

The adjacency matrix of the graph.

Type
NumPy array

`attribute_expression()`

Initialize the expression associated to the diagram.

`attribute_ph_labels()`

Attribute the appropriate qp labels to the graph's propagators.

`calc_excitation()`

Return an integer coding for the excitation level of the diag.

Returns
The singles / doubles / etc. character of the graph.

Return type
int

`cd_denominator()`

Return the computer-readable denominator of the graph.

Returns
The graph denominator tailored for automated frameworks.

Return type
str

`cd_expr`

The expression associated to the diagram in a computer-readable format.

Type
str

`cd_numerator()`

Return the computer-readable numerator.

Returns
The graph numerator tailored for automated frameworks.

Return type
str

`complex_conjugate`

The tag number of the diagram's complex conjugate. -1 if the graph has none.

Type
int

`count_hole_lines()`

Return an integer for the number of hole lines in the graph.

Returns
The number of holes in the diagram.

Return type
int

`degrees`

The ascendingly sorted degrees of the graph vertices.

Type
tuple

`excitation_level`

The single, double, etc., excitation character.

Type
int

`expr`

The MBPT expression associated to the diagram.

Type
str

`extract_denominator()`

Return the denominator for a MBPT graph.

Returns
The denominator of the diagram.

Return type
str

`extract_numerator()`

Return the numerator associated to a MBPT graph.

Returns
The numerator of the diagram.

Return type
str

`graph`

The actual graph.

Type
NetworkX MultiDiGraph

`incidence`

The incidence matrix of the graph.

Type
NumPy array

io_degrees

The sorted version of unsort_io_degrees.

Type
tuple

is_complex_conjug_of(*test_diagram*)

Return True if self and test_diagram are complex conjugate.

Parameters
test_diagram (MbptDiagram) – A diagram to compare with.

Returns
The complex conjugate status of the pair of diagrams.

Return type
bool

loops_number()

Return the number of loops in the diagram as an integer.

Returns
The number of loops in the graph.

Return type
int

max_degree

The maximal degree of a vertex in the graph.

Type
int

tags

The tag numbers associated to a diagram.

Type
list

unsort_degrees

The degrees of the graph vertices

Type
tuple

unsort_io_degrees

The list of in- and out-degrees for each vertex of the graph, stored in a (in, out) tuple.

Type
tuple

write_graph(*latex_file*, *directory*, *write_time*)

Write the graph of the diagram to the LaTeX file.

Parameters

- **latex_file** (*file*) – The LaTeX ouput file of the program.
- **directory** (*str*) – Path to the result folder.
- **write_time** (*bool*) – (Here to emulate polymorphism).

write_section(*result*, *commands*, *flags*)

Write sections for MBPT result file.

Parameters

- **result** (*file*) – The LaTeX output file to be written in.

- **commands** (*dict*) – The flags associated with run management.
- **flags** (*dict*) – The identifier of each section-starting graph.

adg.mbpt.**attribute_conjugate**(*diagrams*)

Attribute to each diagram its complex conjugate.

The diagrams involved in conjugate pairs receive the tag associated to their partner in the `complex_conjugate` attribute.

Parameters
diagrams (*list*) – The topologically unique MbptDiagrams.

adg.mbpt.**diagrams_generation**(*order*)

Generate the diagrams for the MBPT case.

Parameters
order (*int*) – The perturbative order of interest.

Returns
A list of NumPy arrays with the diagrams adjacency matrices.

Return type
list

```
>>> diagrams_generation(2)
[array([[0, 2], [2, 0]])]
>>> diagrams_generation(3)
[array([[0, 2, 0], [0, 0, 2], [2, 0, 0]]),
 array([[0, 1, 1], [1, 0, 1], [1, 1, 0]]),
 array([[0, 0, 2], [2, 0, 0], [0, 2, 0]])]
>>> diagrams_generation(1)
[]
```

adg.mbpt.**extract_cd_denom**(*start_graph*, *subgraph*)

Extract the computer-readable denominator using the subgraph rule.

Parameters

- **start_graph** (*NetworkX MultiDiGraph*) – The studied graph.
- **subgraph** (*NetworkX MultiDiGraph*) – The subgaph for this particular factor.

Returns
The denominator factor associated to this subgraph.

Return type
str

adg.mbpt.**order_diagrams**(*diagrams*)

Order the MBPT diagrams and return the number of diags for each type.

Parameters
diagrams (*list*) – The unordered MbptDiagrams.

Returns
First element are the ordered MbptDiagrams. Second element is the number of diagrams for each excitation level type.

Return type
tuple

adg.mbpt.**print_cd_output**(*directory*, *diagrams*)

Print a computer-readable file for automated frameworks.

Parameters

- **directory** (*str*) – The path to the output directory.
- **diagrams** (*list*) – All the MbptDiagrams.

adg.mbpt.**write_diag_exp**(*latex_file*, *mbpt_diag*)

Write the expression associated to a diagram in the LaTeX file.

Parameters

- **latex_file** (*file*) – The LaTeX output file to be written in.
- **mbpt_diag** (MbptDiagram) – The diagram which expression is being written.

adg.mbpt.**write_header**(*tex_file*, *diags_nbs*)

Write tha appropriate header for the LaTeX file for MBPT diagrams.

Parameters

- **tex_file** (*file*) – The LaTeX ouput file to be written in.
- **diags_nbs** (*dict*) – A dict with the number of diagrams per excitation level type.

4.5 BMBPT Diagram

Routines and class for Bogoliubov MBPT diagrams.

class adg.bmbpt.**BmbptFeynmanDiagram**(*nx_graph*, *tag_num*)

Bases: *Diagram*

Describes a BMBPT Feynman diagram with its related properties.

Parameters

- **nx_graph** (*NetworkX MultiDiGraph*) – The graph of interest.
- **tag_num** (*int*) – The tag number associated to the diagram.

attribute_expressions(*time_diag*)

Attribute the correct Feynman and Goldstone expressions.

Parameters
time_diag (TimeStructureDiagram) – The associated TSD.

attribute_qp_labels()

Attribute the appropriate qp labels to the graph's propagators.

degrees

The ascendingly sorted degrees of the graph vertices.

Type
tuple

diag_exp

The Goldstone expression associated to the diagram.

Type
str

equivalent_permutations()

Return the permutations generating equivalent diagrams.

Returns
Vertices permutations as dictionnaries.

Return type
list

`extract_integral()`

Return the integral part of the Feynman expression of the diag.

Returns
The integral part of its Feynman expression.

Return type
str

`extract_numerator()`

Return the numerator associated to a BMBPT graph.

Returns
The numerator of the graph.

Return type
str

`feynman_exp`

The Feynman expression associated to the diagram.

Type
str

`graph`

The actual graph.

Type
NetworkX MultiDiGraph

`has_crossing_sign()`

Return True for a minus sign associated with crossing propagators.

Use the fact that all lines propagate upwards and the canonical representation of the diagrams and vertices.

Returns

Encode for the sign factor associated with crossing
propagators.

Return type
bool

`has_sign_factor()`

Return True if a sign factor is associated to the diagram.

Wrapper allowing for easy refactoring of expression code.

Returns
The presence of a sign factor.

Return type
bool

`hf_type`

The Hartree-Fock, non-Hartree-Fock or Hartree-Fock for the energy operator only character of the graph.

Type
str

`io_degrees`

The sorted version of unsort_io_degrees.

Type
tuple

`max_degree`
The maximal degree of a vertex in the graph.

Type
int

`multiplicity_symmetry_factor()`
Return the symmetry factor associated with propagators multiplicity.

Returns
The symmetry factor associated with equivalent lines.

Return type
str

`symmetry_factor()`
Return the overall symmetry factor of the diagram.

Returns
The combination of all symmetry factors.

Return type
str

`tags`
The tag numbers associated to a diagram.

Type
list

`time_tag`
The tag number associated to the diagram's associated TSD.

Type
int

`time_tree_denominator(`*time_graph*`)`
Return the denominator for a time-tree graph.

Parameters
`time_graph` (*NetworkX MultiDiGraph*) – Its associated time-structure graph.

Returns
The denominator of the graph.

Return type
str

`tsd_is_tree`
The tree or non-tree character of the associated TSD.

Type
bool

`two_or_three_body`
The 2 or 3-body characted of the vertices.

Type
int

`unique_id`
A unique number associated to the diagram.

Type
int

`unsort_degrees`
The degrees of the graph vertices

Type
tuple

`unsort_io_degrees`
The list of in- and out-degrees for each vertex of the graph, stored in a (in, out) tuple.

Type
tuple

`vert_exp`
The expression associated to the vertices.

Type
list

`property vertex_exchange_sym_factor`
Return the symmetry factor associated with vertex exchange.

Returns
The symmetry factor for vertex exchange.

Return type
int

`vertex_expression`(*vertex*)
Return the expression associated to a given vertex.

Parameters
vertex (*int*) – The vertex of interest in the graph.

Returns
The LaTeX expression associated to the vertex.

Return type
str

`write_diag_exps`(*latex_file*, *norder*)
Write the expressions associated to a diagram in the LaTeX file.

Parameters

- **latex_file** (*file*) – The LaTeX outputfile of the program.
- **norder** (*int*) – The order in BMBPT formalism.

`write_graph`(*latex_file*, *directory*, *write_time*)
Write the BMBPT graph and its associated TSD to the LaTeX file.

Parameters

- **latex_file** (*file*) – The LaTeX output file of the program.
- **directory** (*str*) – The path to the result folder.
- **write_time** (*bool*) – `True` if we want informations on the associated TSDs.

`write_section`(*result*, *commands*, *section_flags*)
Write section and subsections for BMBPT result file.

Parameters

- **result** (*file*) – The LaTeX output file of the program.
- **commands** (*dict*) – The flags associated with run management.
- **section_flags** (*dict*) – UniqueIDs of diags starting each section.

write_tsd_info(*diagrams_time*, *latex_file*)

Write info related to the BMBPT associated TSD to the LaTeX file.

Parameters

- **diagrams_time** (*list*) – The associated TSDs.
- **latex_file** (*file*) – The LaTeX output file of the program.

write_vertices_values(*latex_file*, *mapping*)

Write the qp energies associated to each vertex of the diag.

Parameters

- **latex_file** (*file*) – The LaTeX output file of the program.
- **mapping** (*dict*) – A mapping between the vertices in the diagram and the vertices in its euivalent TSD, since permutations between vertices are possible.

adg.bmbpt.**check_topologically_equivalent**(*matrices*, *max_vertex*)

Exclude matrices that would spawn topologically equivalent graphs.

Parameters

- **matrices** (*list*) – Adjacency matrices to be checked.
- **max_vertex** (*int*) – The maximum vertex which have been filled.

Returns

The topologically unique matrices.

Return type

list

```
>>> import numpy
>>> mats = [numpy.array([[0, 2, 0, 0], [2, 0, 0, 1], [0, 0, 0, 0], [0, 0, 0,␣
↪0]]),                  numpy.array([[0, 0, 2, 0], [0, 0, 0, 0], [2, 0, 0, 1],␣
↪[0, 0, 0, 0]])]
>>>
>>> mats = check_topologically_equivalent(mats, 2)
>>> mats
[array([[0, 2, 0, 0], [2, 0, 0, 1], [0, 0, 0, 0], [0, 0, 0, 0]])]
>>>
>>> mats = check_topologically_equivalent([], 2)
>>> mats
[]
```

adg.bmbpt.**check_unconnected_spawn**(*matrices*, *max_filled_vertex*)

Exclude some matrices that would spawn unconnected diagrams.

Do several permutations among the rows and columns corresponding to already filled vertices, and check if one obtains a block-diagonal organisation, where the off-diagonals blocks connecting the already-filled and yet-unfilled parts of the matrix would be empty. In that case, remove the matrix.

Parameters

- **matrices** (*list*) – The adjacency matrices to be checked.
- **max_filled_vertex** (*int*) – The furthest vertex until which the matrices have been filled.

```
>>> import numpy
>>> mats = [numpy.array([[0, 2, 0], [2, 0, 0], [0, 0, 0]]),                    ␣
↪numpy.array([[0, 2, 1], [2, 0, 1], [0, 0, 0]])]
>>>
```

(continues on next page)

(continued from previous page)

```
>>> check_unconnected_spawn(mats, 1)
>>> mats
[array([[0, 2, 1], [2, 0, 1], [0, 0, 0]])]
```

adg.bmbpt.**diagrams_generation**(*p_order*, *three_body_use*, *nbody_obs*, *canonical*)

Generate diagrams for BMBPT from bottom up.

Parameters

- **p_order** (*int*) – The BMBPT perturbative order of the studied diagrams.
- **three_body_use** (*bool*) – Flag for the use of three-body forces.
- **nbody_obs** (*int*) – N-body character of the obervable of interest.
- **canonical** (*bool*) – True if one draws only canonical diagrams.

Returns
NumPy arrays encoding the adjacency matrices of the graphs.

Return type
list

```
>>> diags = diagrams_generation(1, False, 2, False)
>>> len(diags)
2
>>> any(np.array_equal([[0, 4], [0, 0]], diag) for diag in diags)
True
>>> any(np.array_equal([[0, 2], [0, 0]], diag) for diag in diags)
True
>>> diags = diagrams_generation(1, True, 3, False)
>>> len(diags)
3
>>> any(np.array_equal([[0, 6], [0, 0]], diag) for diag in diags)
True
>>> any(np.array_equal([[0, 4], [0, 0]], diag) for diag in diags)
True
>>> any(np.array_equal([[0, 2], [0, 0]], diag) for diag in diags)
True
>>> diags = diagrams_generation(2, False, 2, True)
>>> len(diags)
2
>>> any(np.array_equal([[0, 2, 2], [0, 0, 2], [0, 0, 0]], d) for d in diags)
True
>>> any(np.array_equal([[0, 1, 1], [0, 0, 3], [0, 0, 0]], d) for d in diags)
True
```

adg.bmbpt.**order_and_remove_topologically_equiv**(*matrices*, *max_vertex*)

Order the matrices in sub-list and remove topologically equivalent ones.

Parameters

- **matrices** (*list*) – The adjacency matrices to be checked.
- **max_vertex** (*int*) – The maximum vertex which has been filled.

Returns
The ordered topologically unique matrices.

Return type
list

`adg.bmbpt.order_diagrams(diagrams)`

Order the BMBPT diagrams and return number of diags for each type.

Parameters
diagrams (*list*) – Possibly redundant BmbptFeynmanDiagrams.

Returns

First element is the list of topologically unique, ordered
diagrams. Second element is a dict with the number of diagrams for each major type. Third element is a dict with the identifiers of diagrams starting each output file section.

Return type
tuple

`adg.bmbpt.produce_expressions(diagrams, diagrams_time)`

Produce and store the expressions associated to the BMBPT diagrams.

Parameters

- **diagrams** (*list*) – The list of all BmbptFeynmanDiagrams.
- **diagrams_time** (*list*) – Their associates TSDs.

`adg.bmbpt.remove_disconnected_matrices(matrices)`

Remove matrices corresponding to disconnected diagrams.

Parameters
matrices (*list*) – List of adjacency matrices.

`adg.bmbpt.write_header(tex_file, commands, diags_nbs)`

Write overall header for BMBPT result file.

Parameters

- **tex_file** (*file*) – The ouput LaTeX file of the program.
- **commands** (*Namespace*) – Flags for the program run.
- **diags_nbs** (*dict*) – The number of diagrams per type.

4.6 PBMBPT Diagram

Routines and class for Projected Bogoliubov MBPT diagrams.

`class adg.pbmbpt.ProjectedBmbptDiagram(graph, unique_id, tag, child_tag)`

Bases: *BmbptFeynmanDiagram*

Describes a PBMBPT diagram with its related properties.

Parameters

- **graph** (*NetworkX MultiDiGraph*) – The graph of interest.
- **unique_id** (*int*) – The unique number associated to the diagram.
- **tag** (*int*) – Tag of the parent BMBPT diagram.
- **child_tag** (*int*) – Identifier of the PBMBPT diagram within the children of the BMBPT diagram.

`anomalous_contractions_factor()`

Return the factor associated with anomalous self-contractions.

Returns
The anomalous self-contractions factor.

Return type
int

`attribute_expressions`(*time_diag*)

Attribute the correct Feynman and Goldstone expressions.

Parameters
time_diag (`TimeStructureDiagram`) – The associated TSD.

`attribute_qp_labels()`

Attribute the appropriate qp labels to the graph's propagators.

`property check_graph`

Return a graph that can be used for topological equivalence checks.

Lazy-initialized to reduce memory and CPU costs as this operation requires a deep copy.

Returns
The graph with doubled anomalous props.

Return type
NetworkX MultiDiGraph

`degrees`

The ascendingly sorted degrees of the graph vertices.

Type
tuple

`diag_exp`

The Goldstone expression associated to the diagram.

Type
str

`equivalent_permutations()`

Return the permutations generating equivalent diagrams.

Returns
Vertices permutations as dictionnaries.

Return type
list

`extract_integral()`

Return the integral part of the Feynman expression of the diag.

Returns
The integral part of its Feynman expression.

Return type
str

`extract_numerator()`

Return the numerator associated to a PBMBPT graph.

Returns
The numerator of the graph.

Return type
str

`feynman_exp`

The Feynman expression associated to the diagram.

Type
str

`graph`

The actual graph.

Type
NetworkX MultiDiGraph

`has_anom_non_selfcontracted_props()`

Return True if the diagram has anomalous propagators.

Returns
The presence of anomalous propagators.

Return type
bool

`has_anom_props_linked_sign()`

Return True if there is a minus sign associated to anom props.

Anomalous propagators departing to higher vertices introduce a sign factor if a normal propagator is going to an even higher vertex, as it departs from the canonical representation used for numerator extraction.

Returns
The presence of the sign factor.

Return type
bool

`has_crossing_sign()`

Return True for a minus sign associated with crossing propagators.

Use the fact that all lines propagate upwards and the canonical representation of the diagrams and vertices.

Returns

Encode for the sign factor associated with crossing
propagators.

Return type
bool

`has_sign_factor()`

Return True if a sign factor is associated to the diagram.

Wrapper allowing for easy refactoring of expression code.

Returns
The presence of a sign factor.

Return type
bool

`hf_type`

The Hartree-Fock, non-Hartree-Fock or Hartree-Fock for the energy operator only character of the graph.

Type
str

`io_degrees`

The sorted version of unsort_io_degrees.

Type
tuple

`max_degree`

The maximal degree of a vertex in the graph.

Type
int

`multiplicity_symmetry_factor()`

Return the symmetry factor associated with propagators multiplicity.

Returns
The symmetry factor associated with equivalent lines.

Return type
str

`set_io_degrees()`

Attribute the correct in- and out-degrees to a PBMBPT diagram.

`symmetry_factor()`

Return the overall symmetry factor of the diagram.

Returns
The combination of all symmetry factors.

Return type
str

`tags`

The tag numbers associated to a diagram.

Type
list

`time_tag`

The tag number associated to the diagram's associated TSD.

Type
int

`time_tree_denominator`(*time_graph*)

Return the denominator for a time-tree graph.

Parameters
time_graph (*NetworkX MultiDiGraph*) – Its associated time-structure graph.

Returns
The denominator of the graph.

Return type
str

`tsd_is_tree`

The tree or non-tree character of the associated TSD.

Type
bool

`two_or_three_body`

The 2 or 3-body characted of the vertices.

Type
int

`unique_id`

A unique number associated to the diagram.

Type
int

`unsort_degrees`
The degrees of the graph vertices

Type
tuple

`unsort_io_degrees`
The list of in- and out-degrees for each vertex of the graph, stored in a (in, out) tuple.

Type
tuple

`vert_exp`
The expression associated to the vertices.

Type
list

`property vertex_exchange_sym_factor`
Return the symmetry factor associated with vertex exchange.

Returns
The symmetry factor for vertex exchange.

Return type
int

`vertex_expression`(*vertex*)
Return the expression associated to a given vertex.

Parameters
vertex (*int*) – The vertex of interest in the graph.

Returns
The LaTeX expression associated to the vertex.

Return type
str

`write_diag_exps`(*latex_file*, *norder*)
Write the expressions associated to a diagram in the LaTeX file.

Parameters

- **latex_file** (*file*) – The LaTeX outputfile of the program.
- **norder** (*int*) – The order in BMBPT formalism.

`write_graph`(*latex_file*, *directory*, *write_time*)
Write the PBMBPT graph and its associated TSD to the LaTeX file.

Parameters

- **latex_file** (*file*) – The LaTeX output file of the program.
- **directory** (*str*) – The path to the result folder.
- **write_time** (*bool*) – `True` if we want informations on the associated TSDs.

`write_section`(*result*, *commands*, *section_flags*)
Write section and subsections for BMBPT result file.

Parameters

- **result** (*file*) – The LaTeX output file of the program.

- **commands** (*dict*) – The flags associated with run management.
- **section_flags** (*dict*) – UniqueIDs of diags starting each section.

write_tsd_info(*diagrams_time*, *latex_file*)

Write info related to the BMBPT associated TSD to the LaTeX file.

Parameters

- **diagrams_time** (*list*) – The associated TSDs.
- **latex_file** (*file*) – The LaTeX output file of the program.

write_vertices_values(*latex_file*, *mapping*)

Write the qp energies associated to each vertex of the diag.

Parameters

- **latex_file** (*file*) – The LaTeX output file of the program.
- **mapping** (*dict*) – A mapping between the vertices in the diagram and the vertices in its euivalent TSD, since permutations between vertices are possible.

adg.pbmbpt.**equiv_generating_permutations**(*graph*)

Return the list of permutations generating equivalent PBMBPT diags.

Parameters
graph (*Networkx MultiDiGraph*) – The graph to be checked.

Returns
The mappings giving equivalent graphs, inc. identity.

Return type
list

adg.pbmbpt.**filter_new_diagrams**(*new_diags*, *old_diags*)

Eliminate diagrams having a topologically equivalent diag.

Parameters

- **new_diags** (*list*) – The list of newly created PBMBPT diagrams.
- **old_diags** (*list*) – The list of already checked PBMBPT diagrams.

adg.pbmbpt.**generate_anomalous_diags**(*diag*, *nbody_max*)

Generate PBMBPT graphs with anomalous lines, with some redundancy.

Parameters

- **diag** (BmbptFeynmanDiagram) – The diagram to generate children from.
- **nbody_max** (*int*) – The maximal n-body character of a graph vertex.

Returns
The anomalous graphs generated.

Return type
list

adg.pbmbpt.**generate_combinations**(*iter_list*)

Generate all possible combinations of length 1 to total.

Parameters
iter_list (*list*) – A list of iterable objects.

Returns
A list with all the possible combinations of all lengths.

Return type
list

```
>>> print(generate_combinations([1, 2, 3]))
[(1,), (1, 2), (1, 2, 3), (1, 3), (2,), (2, 3), (3,)]
```

adg.pbmbpt.**unique_edge_combinations**(*edges*, *permutations*)

Return all edge combinations not producing equivalent anomalous graphs.

Parameters

- **edges** (*list*) – The edges that can be modified.
- **permutations** (*list*) – The permutation generating equivalent diagrams.

Returns
The list of edges producing unique anomalous diagrams.

Return type
list

```
>>> edges = [(1, 3), (2, 3)]
>>> permutations = [{1: 1, 2: 2}, {1: 2, 2: 1}]
>>> print(unique_edge_combinations(edges, permutations))
[((1, 3), (2, 3)), ((2, 3),)]
```

adg.pbmbpt.**unique_vertex_combinations**(*vertices*, *permutations*)

Return vertex combinations generating unique anomalous diagrams.

Return combinations of vertices on which self-contractions can be added without producing topologically equivalent PBMBPT diagrams.

Parameters

- **vertices** (*list*) – Vertices that can be self-contracted.
- **permutations** (*list*) – The permutations that generate equivalent diags.

Returns
Vertex combinations that do not produce equivalent diags.

Return type
list

```
>>> vertices = [1, 3]
>>> permutations = [{1: 1, 3: 3}, {1: 3, 3: 1}]
>>> print(unique_vertex_combinations(vertices, permutations))
[(1, 3), (3,)]
```

4.7 Time-Structure Diagram

Module with functions relative to time-stucture diagrams, called by ADG.

class adg.tsd.**TimeStructureDiagram**(*bmbpt_diag*)

Bases: *Diagram*

Describes a time-structure diagram with its related properties.

Parameters
bmbpt_diag (BmbptFeynmanDiagram) – The BMBPT graph to be turned into a TSD.

degrees

The ascendingly sorted degrees of the graph vertices.

Type
tuple

`draw_equivalent_tree_tsds`(*latex_file*)

Draw the equivalent tree TSDs for a given non-tree TSD.

Parameters
`latex_file` (*file*) – The output LaTeX file of the program.

`equivalent_trees`

The tag numbers of the equivalent tree TSDs associated to a non-tree TSD.

Type
list

`expr`

The Goldstone denominator associated to the TSD.

Type
str

`get_feynman_diags`(*feyn_diagrams*)

Return the list of Feynman diagrams associated to the TSD.

Parameters
`feyn_diagrams` (*list*) – All produced BmbptFeymmanDiagrams.

Returns
All the identifiers of associated BmbptFeymmanDiagrams.

Return type
str

`graph`

The actual graph.

Type
NetworkX MultiDiGraph

`io_degrees`

The sorted version of unsort_io_degrees.

Type
tuple

`is_tree`

The tree or non-tree character of a TSD.

Type
bool

`max_degree`

The maximal degree of a vertex in the graph.

Type
int

`perms`

The permutations on the vertices for all the BMBPT diagrams associated to this TSD.

Type
dict

`resum`

The resummation power of a tree TSD.

Type
int

`resummation_power()`

Calculate the resummation power of the tree TSD.

Returns
The resummation power associated to the TSD.abs

Return type
int

`tags`

The tag numbers associated to a diagram.

Type
list

`treat_cycles()`

Find and treat cycles in a TSD diagram.

Returns
The unique tree TSDs associated to a non-tree TSD.

Return type
list

`unsort_degrees`

The degrees of the graph vertices

Type
tuple

`unsort_io_degrees`

The list of in- and out-degrees for each vertex of the graph, stored in a (in, out) tuple.

Type
tuple

`write_graph`(*latex_file*, *directory*, *write_time*)

Write the graph of the diagram to the LaTeX file.

Parameters

- **latex_file** (*file*) – The LaTeX ouput file of the program.
- **directory** (*str*) – Path to the result folder.
- **write_time** (*bool*) – (Here to emulate polymorphism).

`adg.tsd.disentangle_cycle`(*time_graph*, *cycle_nodes*)

Separate a cycle in a sum of tree diagrams.

Parameters

- **time_graph** (*NetworkXn MultiDiGraph*) – A time-structure diagram.
- **cycle_nodes** (*tuple*) – Integers encoding the positions of the end nodes of the cycle.

Returns
New graphs produced from treating the cycles in the TSD.

Return type
list

`adg.tsd.equivalent_labelled_tsds`(*equivalent_trees*, *labelled_tsds*)

Return the list of labelled TSDs corresponding to equivalent TSDs.

Parameters

- **equivalent_trees** (*list*) – The equivalent tree TSDs of a non-tree TSD.

- **labelled_tsds** (*list*) – The labelled TSDs obtained from BMBPT diagrams.

Returns
The list of tag numbers of the equivalent TSDs.

Return type
str

adg.tsd.**find_cycle**(*graph*)

Return start and end nodes for an elementary cycle.

Parameters
graph (*NetworkX MultiDiGraph*) – A TSD with cycle(s) to be treated.

Returns
Positions of the two end nodes of a cycle in the graph.

Return type
tuple

adg.tsd.**time_structure_graph**(*diag*)

Return the time-structure graph associated to the graph.

Parameters
diag (*BmbptFeymmanDiagram*) – The BMBPT graph of interest.

Returns
The time-structure diagram.

Return type
NetworkX MultiDiGraph

adg.tsd.**treat_tsds**(*diagrams_time*)

Order TSDs, produce their expressions, return also number of trees.

Parameters
diagrams_time (*list*) – All the associated TSDs.

Returns
List of TSDs, number of tree TSDs

Return type
tuple

adg.tsd.**tree_time_structure_den**(*time_graph*)

Return the denominator associated to a tree time-structure graph.

Parameters
time_graph (*NetworkX MultiDiGraph*) – The TSD of interest.

Returns
The denominator associated to the TSD.

Return type
str

adg.tsd.**write_section**(*latex_file*, *directory*, *pdiag*, *time_diagrams*, *nb_tree_tsds*, *diagrams*)

Write the appropriate section for tsd diagrams in the LaTeX file.

Parameters

- **latex_file** (*file*) – The LaTeX output file of the program.
- **directory** (*str*) – Path to the output folder.
- **pdiag** (*bool*) – True if diagrams are to be drawn.
- **time_diagrams** (*list*) – The ensemble of TSDs.

- **nb_tree_tsds** (*int*) – Number of tree TSDs.
- **diagrams** (*list*) – All produced BmbptFeymmanDiagrams.

4.8 BIMSRG Diagram

Routines and class for Bogoliubov IMSRG diagrams.

class adg.bimsrg.**BimsrgDiagram**(*nx_graph*, *tag_num*)

Bases: *Diagram*

Describes a B-IMSRG Feynman diagram with its related properties.

Parameters

- **nx_graph** (*NetworkX MultiDiGraph*) – The graph of interest.
- **tag_num** (*int*) – The tag number associated to the diagram.

adjacency_mat

The adjacency matrix of the diagram.

Type

Numpy array

attribute_expression()

Returns the LaTeX expression of the diagram.

The expression is extracted in a way that assumes the canonical representation of the vertices as well as a labeling of the external lines that would correspond to a canonical representation of the C operator vertex. As such, there is no sign factor associated to departing from the canonical representation of the diagram. This additionally prevents any line crossing from appearing.

Returns

The LaTeX expression for the diagram.

Return type

str

degrees

The ascendingly sorted degrees of the graph vertices.

Type

tuple

expr

The B-IMSRG expression associated to the diagram.

Type

str

ext_io_degree

The degree of the operator component the diagram corresponds to.

Type

tuple

graph

The actual graph.

Type

NetworkX MultiDiGraph

`io_degrees`
The sorted version of unsort_io_degrees.

Type
tuple

`is_AB`
True if the diagram contributes to +AB, false if to -BA.

Type
bool

`max_degree`
Maximal degree over vertices or external lines (A, B, C).

Type
int

`permutator()`
Return the permutator associated to the diagram.

The labelling of the external lines correspond to the canonical representation of the C operator vertex.

Returns
The permutator associated to the diagram in LaTeX format.

Return type
str

`sign()`
Return the sign of the diagram.

As the diagrams are made with vertices in canonical representation and avoiding crossings, the only sign left is associated with the commutator.

Returns
The sign of the diagram.

Return type
str

`symmetry_factor()`
Returns the symmetry factor of the diagram in LaTeX format.

Returns
The LaTeX-formatted symmetry factor.

Return type
str

`tags`
The tag numbers associated to a diagram.

Type
list

`unique_id`
A unique number associated to the diagram.

Type
int

`unsort_degrees`
The degrees of the graph vertices

Type
tuple

unsort_io_degrees

The list of in- and out-degrees for each vertex of the graph, stored in a (in, out) tuple.

Type
tuple

vertices_expression()

Return the expression associated to the vertices in LaTeX format.

The expressions are extracted in a way that assumes the canonical representation of the vertices as well as a labeling of the external lines that would correspond to a canonical representation of the C operator vertex. This prevents any crossing as well as additional sign tied to departing from the canonical representation.

Returns
The LaTeX-formatted expression for the vertices.

Return type
str

write_graph(*latex_file*, *directory*, *write_time*)

Write the graph of the diagram to the LaTeX file.

Parameters

- **latex_file** (*file*) – The LaTeX ouput file of the program.
- **directory** (*str*) – Path to the result folder.
- **write_time** (*bool*) – (Here to emulate polymorphism).

write_section(*result*, *commands*, *section_flags*)

Write section and subsections for BMBPT result file.

Parameters

- **result** (*file*) – The LaTeX output file of the program.
- **commands** (*dict*) – The flags associated with run management.
- **section_flags** (*dict*) – UniqueIDs of diags starting each section.

adg.bimsrg.**diagrams_generation**(*orders*)

Generate diagrams for B-IMSRG.

Parameters
orders (*tuple*) – The B-IMSRG (N_A, N_B, N_C) order of the diagrams.

Returns

NumPy arrays encoding the adjacency matrices of the graphs,
and number of diagrams with the A vertex on top.

Return type
tuple

adg.bimsrg.**diagrams_subset**(*deg_max_top*, *deg_max_bot*, *deg_max_ext*)

Generate diagrams for B-IMSRG.

Parameters
orders (*tuple*) – The max ranks (2*N_A, 2*N_B, 2*N_C) of the vertices.

Returns
NumPy arrays encoding the adjacency matrices of the graphs.

Return type
list

```
>>> diagrams_subset(2, 2, 0)
[array([[0, 0, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])]
>>> len(diagrams_subset(2, 2, 2))
5
>>> len(diagrams_subset(4, 4, 4))
41
>>> len(diagrams_subset(0, 0, 0))
0
```

adg.bimsrg.**order_diagrams**(*diagrams*, *order*)

Order the BIMSRG diagrams and return the number of diags for each type.

Parameters

- **diagrams** (*list*) – The unordered BimsrgDiagrams.
- **order** (*int*) – The order of the B-IMSRG truncation.

Returns

First element are the ordered BimsrgDiagrams. Second element is the number of diagrams for each type. Third element is flags for the output processing.

Return type

tuple

adg.bimsrg.**permutator**(*set_1*, *set_2*)

Write the definition of the permutation operator given by the two sets.

Parameters

- **set_1** (*list*) – The list of the left-hand-side qp labels.
- **set_2** (*list*) – The list of the right-hand-side qp labels.

Returns

The LaTeX expression for the permutation operator.

Return type

str

```
>>> print(permutator([1, 2], [3]))
P(k_{1}k_{2}/k_{3}) &= 1 - P_{k_{1} k_{3}} - P_{k_{2} k_{3}} \\
```

adg.bimsrg.**two_partitions**(*number*)

Return 2-partitions of the given integer.

Parameters

numbr (*int*) – The integer to partition.

Returns

All the 2-partitions as tuples.

Return type

list

```
>>> two_partitions(3)
[(0, 3), (1, 2), (2, 1), (3, 0)]
```

```
>>> two_partitions(0)
[(0, 0)]
```

```
>>> two_partitions(-1)
[]
```

`adg.bimsrg.write_header`(*tex_file*, *commands*, *diags_nbs*)

Write overall header for BIMSRG result file.

Parameters

- **tex_file** (*file*) – The ouput LaTeX file of the program.
- **commands** (*Namespace*) – Flags for the program run.
- **diags_nbs** (*dict*) – The number of diagrams per type.

`adg.bimsrg.write_permutator_section`(*tex_file*, *commands*)

Write the section defining permutation operators.

Parameters

- **tex_file** (*file*) – The ouput LaTeX file of the program.
- **commands** (*Namespace*) – Flags for the program run.

4.9 Tools and utilities

Miscellaneous diagram-unrelated tools for ADG.

class `adg.tools.UniqueID`

Bases: `object`

Counter making sure of generating a unique ID number for diagrams.

current

The unique identifier to be attributedto a diagram.

Type
int

get()

Iterate on counter value and return current value.

Returns
A unique identifier for the diagram.

Return type
int

`adg.tools.reversed_enumerate`(*data*)

Return the index and item of the data in reversed order.

Parameters
data (*iterable data structure*) – The data to be used..

Returns
Index and item.

Return type
tuple

```
>>> list(reversed_enumerate(['A', 'B', 'C']))
[(2, 'C'), (1, 'B'), (0, 'A')]
```

CHAPTER

FIVE

DEVELOPERS TEAM

They have been involved in the making of ADG over the past years:

- Pierre Arthuis - TU Darmstadt & ExtreMe Matter Institute EMMI, GSI, Darmstadt (previously University of Surrey & Irfu, CEA, Université Paris-Saclay & CEA, DAM, DIF)
- Thomas Duguet - Irfu, CEA, Université Paris-Saclay & KU Leuven, IKS
- Jean-Paul Ebran - CEA, DAM, DIF
- Heiko Hergert - NSCL/FRIB Laboratory & Department of Physics and Astronomy, Michigan State University
- Raphaël-David Lasseri - ESNT, Irfu, CEA, Université Paris-Saclay (previously IPN, CNRS/IN2P3, Université Paris-Sud, Université Paris-Saclay)
- Julien Ripoche - CEA, DAM, DIF
- Alexander Tichai - MPI fuer Kernphysik, Heidelberg & TU Darmstadt & ExtreMe Matter Institute EMMI, GSI, Darmstadt (previsously ESNT, Irfu, CEA, Université Paris-Saclay)

CHAPTER

SIX

CITING

If you use ADG in your research work, we kindly ask you to cite the following papers:

- P. Arthuis, T. Duguet, A. Tichai, R.-D. Lasseri and J.-P. Ebran, Comput. Phys. Commun. **240**, 202-227 (2019). It is available under the following DOI.
- P. Arthuis, A. Tichai, J. Ripoche and T. Duguet, Comput. Phys. Commun. **261**, 107677 (2021). It is available here.
- A. Tichai, P. Arthuis, H. Hergert and T. Duguet, Eur. Phys. J. A **58**, 2 (2022). It is available under this URL.

CHAPTER

SEVEN

LICENSE

ADG is licensed under under GNU General Public License version 3 (see LICENSE.txt for the full GPLv3 License).

```
Copyright (C) 2018-2023 ADG Dev Team
Pierre Arthuis
Thomas Duguet
Jean-Paul Ebran
Heiko Hergert
Raphaël-David Lasseri
Julien Ripoche
Alexander Tichai
```

CHAPTER EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

E

F

G

H

I

U

V

W