Wirt_Hm_Suite_Sage Documentation

Usage Guide and Reference Manual

Nathaniel Morrison

5/12/2020

This work was partially supported by grants from the National Science Foundation DMS–1821254.

Table of Contents

Introduction

Purpose

The Wirt_Hm_Suite_Sage package is a set of programs designed to conduct an exhaustive search for surjective homomorphisms from the fundamental group of a topological knot with Wirtinger number *n* to a finite Coxeter group of Coxeter rank *n*. The existence of such a surjective homomorphism is equivalent to a proof of the Cappell-Shaneson Meridional Rank Conjecture (Problem 1.11 on the Kirby List[1]) for the knot. For a proof that this is true, see Baader et al.[2] For an explanation of the Wirtinger number itself, see R. Blair et al.[3]

This package makes extensive use of the calc_wirt algorithm developed by P. Villanueva, used to compute the Wirtinger number of a knot and parse Gauss code into a Python-readable "knot dictionary". For the original calc_wirt.py program, as well as a detailed description of the algorithm, see P. Villanueva's GitHub page.[4]

Requirements to Run

1. A SageMath installation. SageMath 8.6 or above is recommended. Go to https://www.sagemath.org/ for download and installation guide.
2. A Microsoft Excel installation. Excel 2010 or later is recommended. A compatible program, for example LibreOffice Calc, should also work.
3. The following Python modules should be installed ***in Sage***, if they are not already:
    a. numpy
    b. xlsxwriter
    c. xlrd
4. The following files should be placed in the same folder:
    a. Wirt_Hm_Suite_Master.py
    b. gauss_processor.py
    c. excel_reader.py
    d. calc_wirt.py
    e. hm.py
    f. gen.py
    g. D5_gens_pruned.xlsx
    h. D4_gens_pruned.xlsx
    i. H4_gens_pruned.xlsx
    j. H3_gens_pruned.xlsx

[1] R. Kirby, Problems in low-dimensional topology, Proceedings of Georgia Topology Conference, Part 2 (R. Kirby, ed.), Citeseer, 1995.
[2] S. Baader, R. Blair, and A. Kjuchukova, Coxeter groups and meridional rank of links. arXiv:1907.02982, 2019.
[3] R. Blair, A. Kjuchukova, R. Velazquez, and P. Villanueva, Wirtinger systems of generators of knot groups, Communications in Analysis and Geometry. In press - arXiv:1705.03108, 2017.
[4] https://github.com/pommevilla/calc_wirt

General Usage

Wirt_Hm_Suite_Master.py should be run by navigating to the directory containing all the Wirt_Hm_Suite_Sage files and invoking Sage. Typically, once in the correct directory, this is done via the command:

sage Wirt_Hm_Suite_Master.py

The program will start and present a menu. The following is an explanation of menu choices:

1. **Compute Wirtinger Number:** Computes the Wirtinger number of a knot using the calc_wirt.py algorithm.
2. **Search for a surjective homomorphism to a symmetric group:** Computes the Wirtinger number *n* of a knot and carries out, if possible, an exhaustive search for a surjective homomorphism to the S_{n+1} symmetric group.
3. **Search for a surjective homomorphism to a D group:** Computes the Wirtinger number *n* of a knot and carries out, if possible, an exhaustive search for a surjective homomorphism to the D_n Coxeter group.
4. **Search for a surjective homomorphism to an H group:** Computes the Wirtinger number *n* of a knot and carries out, if possible, an exhaustive search for a surjective homomorphism to the H_n Coxeter group.
5. **Search for a surjective homomorphism to any of the symmetric, D, and/or H groups:** Computes the Wirtinger number *n* of a knot and carries out, if possible, an exhaustive search for a surjective homomorphism to the S_{n+1}, D_n, and H_n groups.
6. **Quit:** Terminates the program. Does not shutdown the kernel.

The following is an explanation of input options:

1. **Input Gauss code of a single knot:** Accepts as input a list of positive and negative integers giving the Gauss code of the knot. The integers must be separated by non-numeric characters, but beyond this the choice of separation characters and the presence or absence of non-numeric leading or trailing characters is irrelevant. Example valid gauss codes: 1 -2 3 -4 5 -6 2 -1 6 -3 4 -5 ; {1,-2,3,-4,5,-6,2,-1,6,-3,4,-5} ; put1w-2ha3t-4ever5y-6o2u-1w6a-3nt4h-5ere!
2. **Input Excel file:** Accepts the path to an .xlsx file. Ensure that the name and Gauss code of each knot you wish to process are present in the file, that all names are placed in one column and all codes are placed in a different column, and that there are no empty rows between the first row of knot data and the last row of knot data. The program will scan through each row and run the desired algorithm on each knot in succession.
    a. **Enter the input file path:** Enter the path to the Excel file containing the knot data (e.g. C:\Users\Me\InputFiles\knotList.xlsx)
    b. **Number of the column containing knot names:** Enter the name of the column in the input spreadsheet, converted to a number as indicated in the prompt, containing the names of the knots to be processed. The names can be any combination of characters.

c. **Number of the column containing gauss code:** Enter the name of the column in the input spreadsheet, converted to a number as indicated in the prompt, containing the Gauss code of the knots to be processed. The Gauss code should be formatted as described in the above "Input Gauss code of a single knot" explanation.
d. **Number of the row containing the first knot's information:** Enter the number of the row in which the first knot is listed.
e. **Name of excel output file:** Enter the desired location and name of the Excel file which will be generated to contain the results of the computation. (e.g. C:\Users\Me\OutputFiles\homomorphismList.xlsx)
f. **Wirtinger number of the knots you wish to analyze:** The program gives the user the option of searching for homomorphisms to knots of a particular Wirtinger number, while ignoring all other knots. This can result in significant improvements to processing time, if it is known that only knots of a particular Wirtinger number will be entered or are needed. Wirtinger numbers 4 and 5 are selectable for *D* group searches, and 3 and 4 for *H* group searches. Alternatively, enter 0 to search for homomorphisms to all knots whenever possible, without regard for Wirtinger number. Note that regardless of selection, knots with Wirtinger number 6 or greater will be ignored, with no search conducted.

The following is an explanation of outputs:

1. For single-knot entries:
    a. **Wirtinger number:** The Wirtinger number of the knot
    b. **Seed strand set used to color the knot:** The set of strands of the knot used as seeds to generate a complete Wirtinger coloring. The number of strands is equal to the Wirtinger number, and is the minimum number of seeds needed to completely color the knot. The labels given correspond to entries in the knot dictionary.
    c. **Knot dictionary:** A Python dictionary with strand labels as keys and two-element lists as entries. The first element of each list is the Gauss code segment corresponding to the strand. The second element is a list of two-tuples, one for each crossing for which this strand is the overstrand. The elements of the two-tuples are the understrands of the crossing.
    d. **Seed strand to symmetric group generating set mapping:** A Python dictionary in which the keys are seed strands of the knot (see above) and the entries are transpositions of the S_{n+1} symmetric group, where *n* is the Wirtinger number. Together, the *n* transpositions form a generating set of the group and satisfy the Wirtinger relations of the knot. This mapping thus provides a surjective homomorphism from the fundamental group of the knot to the S_{n+1} symmetric group.
    e. **Seed strand to D or H group generating set mapping:** A Python dictionary in which the keys are seed strands of the knot (see above) and the entries are matrix elements of the D_n or H_n group, where *n* is the Wirtinger number. Each matrix is represented as a list of its row vectors separated by a "|" symbol. Together, the *n* elements form a generating

set of the group and satisfy the Wirtinger relations of the knot. This mapping thus provides a surjective homomorphism from the fundamental group of the knot to the D_n or H_n group, respectively.

2. For Excel entries: Output is an Excel file with the following format:
    a. **First column:** Knot names, same as those in the input file
    b. **Second column:** Gauss code of the knot
    c. **Third column:** Seed strands used to color the knot as described above. This is a Python dictionary, where the keys are strand labels and the entries are the Gauss code segments corresponding to the strands.
    d. **Fourth column:** Wirtinger number of the knot
    e. **The format for the following entries depends upon initial menu selection:**
        i. **If option 1 was selected from the first menu:** All further columns are empty
        ii. **If option 2, 3, or 4 was selected from the first menu:**
            1. **Fifth column:** Boolean value. True if a surjective homomorphism to the selected group was found, False if a search was conducted but no such homomorphism was found, N/A if no search was conducted.
            2. **Sixth column:** Mapping of seed strands to symmetric, *D,* or *H* group generators which gives a surjective homomorphism from the fundamental group of the knot to the group generated by the generators.
        iii. **If option 5 was selected from the first menu**:
            1. **Fifth column**: Boolean value. True if a surjective homomorphism to a symmetric group was found, False if a search was conducted but no such homomorphism was found, N/A if no search was conducted.
            2. **Sixth column:** Mapping of seed strands to symmetric group generators which gives a surjective homomorphism from the fundamental group of the knot to the group generated by the generators, or N/A if no homomorphism was found.
            3. **Seventh column:** Boolean value. True if a surjective homomorphism to a *D* group was found, False if a search was conducted but no such homomorphism was found, N/A if no search was conducted.
            4. **Eighth column:** Mapping of seed strands to *D* group generators which gives a surjective homomorphism from the fundamental group of the knot to the group generated by the generators, or N/A if no homomorphism was found.
            5. **Ninth column:** Boolean value. True if a surjective homomorphism to an *H* group was found, False if a search was conducted but no such homomorphism was found, N/A if no search was conducted.
            6. **Tenth column:** Mapping of seed strands to *H* group generators which gives a surjective homomorphism from the fundamental group of the knot to the group generated by the generators, or N/A if no homomorphism was found.

Module Documentation

Module: Wirt_Hm_Suite_Master

Functions:

1. menu1(): Displays primary menu, offering choice between computing Wirtinger number only, searching for symmetric group homomorphisms only, searching for *D* group homomorphisms only, searching for *H* group homomorphisms only, or searching for homomorphisms to all three families of groups at once.
    a. Input: None
    b. Output:
        i. Integer from 1 to 6 corresponding to menu choice.
2. menu2(): Gives user the choice to input a single knot's Gauss code or an Excel file, and collects the Gauss code or Excel directory.
    a. Input: None
    b. Output:
        i. Integer, either 1 or 2, corresponding to menu choice; and data, a string consisting either of the Gauss code of the knot or the file name and directory of the input Excel file.
3. main(): Manages Wirt_Hm_Suite operations and displays results of other modules.
    a. Input: None
    b. Output: None

Module: gauss_processor

Functions:

1. process_gauss_code(raw_gauss_code): Converts a string containing the Gauss code of a knot into a list of integers.
    a. Input:
        i. raw_gauss_code: a string containing the Gauss code of a knot. Must be a series of positive and negative numbers each separated by one or more nonnumeric characters; nature of the separation characters and presence or absence of leading and trailing characters is irrelevant.
    b. Output:
        i. code: a list of positive and negative integers.

Module: excel_reader

Functions:

1. knot_processor(cur_row, knot_workbook, knot_table, name_col, gauss_col): Retrieves knot name and Gauss code from an Excel file.
    a. Input:
        i. cur_row: integer, the number of the row containing the name and Gauss code. Row 1 in Excel is row 0 here.
        ii. knot_workbook: xlrd workbook object corresponding to the Excel file
        iii. knot_table: xlrd worksheet object corresponding to the worksheet in the Excel file containing the knot name and Gauss code.
        iv. name_col: integer, the number of the column containing the knot name. Column A in Excel is column 0 here.
        v. gauss_col: integer, the number of the column containing the Gauss code. Column A in Excel is column 0 here.
    b. Output:
        i. name: string consisting of the name of the knot.
        ii. raw_gauss_code: string containing the Gauss code of the knot and any number of separation, leading, and trailing characters.
2. excel_creator(excel_name, choice): Creates an Excel file to hold program output and conducts initial formatting operations.
    a. Input:
        i. excel_name: string consisting of the desired name and directory of the output Excel file, including .xlsx extension.
        ii. choice: integer from 1 to 5 corresponding to the user's menu selection from menu1() in Wirt_Hm_Suite_Master. 1 = only return the Wirtinger number information of the knot. 2 = carry out an *S* group homomorphism search. 3 = carry out a *D* group homomorphism search. 4 = carry out an *H* group homomorphism search. 5 = carry out an *S* group search, a *D* group search, and an *H* group search.
    b. Output
        i. worksheet: xlsxwriter worksheet object consisting of the worksheet on which output data will be written. The first row will contain column headers. The remainder of the worksheet will be empty at this point.
        ii. workbook: xlsxwriter workbook object consisting of the output Excel file.
3. excel_writer(name, seed_strand_with_gauss, knot_dict, raw_gauss_code, wirt_num, gen_set, Knot_Number, worksheet, workbook, choice, hmorph): Writes data for a processed knot to the output Excel file.
    a. Input:
        i. name: string consisting of the knot's name, as provided in the input Excel file.
        ii. seed_strand_with_gauss: dictionary with seed strand labels (strings) as keys and the Gauss code segments (tuples of integers) of the strands as entries.
        iii. knot_dict: dictionary with strand labels (strings) as keys and two-element lists as entries. The first element of each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is a list of two-tuples, one for each crossing for which this strand is the overstrand. The elements of the two-tuples (strings) are the understrands of the crossing.
        iv. raw_gauss_code: string consisting of Gauss code of the knot.

v. wirt_num: integer equal to the Wirtinger number of the knot.
vi. gen_set: Either: an empty list if only the Wirtinger information of the knots is being computed; a dictionary with seed strand labels (strings) as keys and *S*, *D*, or *H* group generators as entries if a search for homomorphisms to a single group is being performed; or a list of three such dictionaries, one for each group, if a search for homomorphisms to all three groups is being performed.
vii. Knot_Number: integer corresponding to the row of the output worksheet in which the knot information will be written. Row 1 in Excel is row 0 here.
viii. worksheet: xlsxwriter worksheet object consisting of the worksheet on which output data will be written.
ix. workbook: xlsxwriter workbook object consisting of the output Excel file.
x. choice: integer from 1 to 5 corresponding to the user's menu selection from menu1() in Wirt_Hm_Suite_Master. 1 = only return the Wirtinger number information of the knot. 2 = carry out an *S* group homomorphism search. 3 = carry out a *D* group homomorphism search. 4 = carry out an *H* group homomorphism search. 5 = carry out an *S* group search, a *D* group search, and an *H* group search.
xi. hmorph: Either: a boolean value if a search for homomorphisms to a single group is being computed, True if a surjective homomorphism was found and False otherwise; or a list of three boolean values, the first indicating whether a surjective homomorphism to an *S* group was found, the second indicating whether a surjective homomorphism to a *D* group was found, and the third indicating whether a surjective homomorphism to an *H* group was found.

b. Output: None

4. excel_main(input_name, choice): Gathers parameters for input Excel file, scans through input file, and calls other modules to gather data to insert into the output Excel file.
    a. Input:
        i. input_name: string, the name and directory of the input Excel file.
        ii. choice: integer from 1 to 5 corresponding to the user's menu selection from menu1() in Wirt_Hm_Suite_Master. 1 = only return the Wirtinger number information of the knot. 2 = carry out an *S* group homomorphism search. 3 = carry out a *D* group homomorphism search. 4 = carry out an *H* group homomorphism search. 5 = carry out an *S* group search, a *D* group search, and an *H* group search.
    b. Output: None

Module: calc_wirt

Functions:

1. create_knot_dictionary(gauss_code): Calls functions which build the knot dictionary for a knot.
    a. Input:
        i. gauss_code: list of integers consisting of the Gauss code of the knot.
    b. Output:
        i. knot_dict: dictionary with strand labels (strings) as keys and two-element lists as entries. The first element of each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is a list of two-tuples, one for each crossing for which this strand is the overstrand. The elements of the two-tuples (strings) are the understrands of the crossing.
2. find_strands(gauss_code): Scans through Gauss code and extracts all segments corresponding to strands of the knot, giving each a label.
    a. Input:
        i. gauss_code: list of integers consisting of the Gauss code of the knot.
    b. Output:
        i. strands_dict: dictionary with strand labels (strings) as keys and two-element lists as entries. The first element in each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is an empty list.
3. find_crossings(knot_dict, gauss_code): Determines which strand serves as the overstrand and which serve as the understrands at each crossing of the knot, and generates the knot_dict.
    a. Input:
        i. knot_dict: Actually the strands_dict returned as output from find_strands(); dictionary with strand labels (strings) as keys and two-element lists as entries. The first element in each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is an empty list.
    b. Output:
        i. knot_dict: Complete version, populated with crossing information; dictionary with strand labels (strings) as keys and two-element lists as entries. The first element of each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is a list of two-tuples, one for each crossing for which this strand is the overstrand. The elements of the two-tuples (strings) are the understrands of the crossing.
4. is_valid_coloring(seed_strands, knot_dict): Checks whether a given set of seed strands provide a complete coloring of the knot as per the Wirtinger coloring algorithm.
    a. Input:
        i. seed_strands: itertools combination object (also accepts lists, sets, and tuples) consisting of strand labels from the knot dictionary. The number of elements is greater than or equal to the Wirtinger number of the knot **if** the is_valid_coloring function returns True.
        ii. knot_dict: dictionary with strand labels (strings) as keys and two-element lists as entries. The first element of each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is a list of two-tuples, one for each crossing for which this strand is the overstrand. The elements of the two-tuples (strings) are the understrands of the crossing.
    b. Output:

        i. Returns True if the seed_strands set results in a complete coloring of the knot given by knot_dict as per the Wirtinger coloring algorithm. Otherwise, returns False.
5. calc_wirt_info(knot_dict): Finds a minimal seed strand set to completely color a knot. Generates all possible seed strand sets of length two, then of length three, and so on until is_valid_coloring returns True or all sets of length less than the number of strands in the knot have been tried. The length of the seed strand set that causes is_valid_coloring to return True is equal to the Wirtinger number of the knot.
    a. Input:
        i. knot_dict: dictionary with strand labels (strings) as keys and two-element lists as entries. The first element of each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is a list of two-tuples, one for each crossing for which this strand is the overstrand. The elements of the two-tuples (strings) are the understrands of the crossing.
    b. Output:
        i. Two-tuple consisting of the first seed strand set which completely colored the knot and the length of the set, which is equal to the Wirtinger number. If is_valid_coloring never returned true for any seed strand set of length less than the number of strands in the knot diagram, then the function will return all strands as a seed strand set and a Wirtinger number equal to the number of strands in the diagram.
6. wirt_main(gauss_code): Creates knot dictionary and calculates the Wirtinger number of a knot with the entered Gauss code.
    a. Input:
        i. gauss_code: a list of positive and negative integers consisting of the Gauss code of a knot.
    b. Output:
        i. knot_dict: dictionary with strand labels (strings) as keys and two-element lists as entries. The first element of each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is a list of two-tuples, one for each crossing for which this strand is the overstrand. The elements of the two-tuples (strings) are the understrands of the crossing.
        ii. seed_strand_set: set of strand labels (strings) which completely color the knot.
        iii. wirt_num: positive integer, the Wirtinger number of the knot.

Module: hm

Functions:

1. homomorphism_finder(seed_strand_set, knot_dict, wirt_num,gen_sets): Computes all permutations of the Wirtinger coloration seed strand set and calls other functions within the module to determine if any admit a surjective homomorphism from the fundamental group of the knot to a finite Coxeter group when mapped to group generators
    a. Input:
        i. seed_strand_set: A list of single-element strings corresponding to strand labels in the knot dictionary. These are the strands which will be initially mapped to symmetric group generators when searching for a homomorphism.
        ii. knot_dict: dictionary with strand labels (strings) as keys and two-element lists as entries. The first element of each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is a list of two-tuples, one for each crossing for which this strand is the overstrand. The elements of the two-tuples (strings) are the understrands of the crossing.
        iii. wirt_num: positive integer, the Wirtinger number of the knot.
        iv. gen_sets: a tuple containing either: lists of SageMath matrix objects if a search using *D* or *H* is being performed, each list being a unique generating set of reflections; or lists of two-element tuples if a search using *S* is being performed, each list being a unique generating set of transpositions.
    b. Output:
        i. hmorph: a Boolean flag, True if a surjective homomorphism from the fundamental group of the knot possessing the entered knot dictionary to the symmetric group S_{wirt_num+1} exists. False otherwise.
        ii. cox_gen_set: dictionary with strand labels (strings) from the seed_strand_set as keys and entries consisting of either: SageMath matrix objects if a search using *D* or *H* is being performed; or two-element tuples if a search using *S* is being performed. If a surjective homomorphism was found, the entries are the generators of the group which, when mapped to the strands in the keys, result in the surjective homomorphism. If a surjective homomorphism was not found, the dictionary gives the last strand-generator assignment the program tried.
2. generator_assign(wirt_num, perm, gen_set): Maps the elements of a generating set of transpositions to a set of seed strands.
    a. Input:
        i. wirt_num: positive integer, the Wirtinger number of the knot.
        ii. perm: A list of single-element strings corresponding to strand labels in the knot dictionary. These are the strands which will be initially mapped to symmetric group generators when searching for a homomorphism. The first element of this list will be mapped to the first element of gen_set, the second element of this list will be mapped to the second element of gen_set, etc.
        iii. gen_set: a tuple of either: SageMath matrix objects if a search using *D* or *H* is being performed, these matrices being group generators; or two-element tuples if a search using *S* is being performed, these tuples being generating transpositions.
    b. Output:

i. cox_gen_set: dictionary with strand labels (strings) from the seed_strand_set as keys and entries consisting of either: SageMath matrix objects if a search using *D* or *H* is being performed; or two-element tuples if a search using *S* is being performed. This is the initial strand mapping the program will attempt to propagate throughout the knot via the Wirtinger relations. If successful, this mapping gives a surjective homomorphism.

3. transposition_assignment(cox_gen_set, knot_dict): Propogates an initial mapping of symmetric group generating transpositions to seed strands through the knot diagram given by knot_dict via the Wirtinger relations.
    a. Input:
        i. cox_gen_set: dictionary with strand labels (strings) from the seed_strand_set as keys and entries consisting of either: SageMath matrix objects if a search using *D* or *H* is being performed; or two-element tuples if a search using *S* is being performed. This is the initial strand mapping the program will attempt to propagate throughout the knot via the Wirtinger relations.
        ii. knot_dict: dictionary with strand labels (strings) as keys and two-element lists as entries. The first element of each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is a list of two-tuples, one for each crossing for which this strand is the overstrand. The elements of the two-tuples (strings) are the understrands of the crossing.
    b. Output:
        i. mapping: dictionary with strand labels (strings) as keys and entries consisting of either: SageMath matrix objects if a search using *D* or *H* is being performed, these matrices being group generators; or two-element tuples if a search using *S* is being performed, these tuples being generating transpositions. Every strand in knot_dict will be present and mapped to a generator. Note that while the Wirtinger relations were used to propagate the transposition mapping to all strands, there will be exactly one crossing per seed strand for which the Wirtinger relation was **not** tested. Thus, the mapping is not necessarily a surjective homomorphism.
4. homomorphism_tester(mapping, knot_dict): Tests if mapping satisfies the Wirtinger relations at every crossing in the knot diagram given by knot_dict, and therefore represents a surjective homomorphism.
    a. Input:
        i. mapping: dictionary with strand labels (strings) as keys and entries consisting of either: SageMath matrix objects if a search using *D* or *H* is being performed, these matrices being group generators; or two-element tuples if a search using *S* is being performed, these tuples being generating transpositions. Every strand in knot_dict will be present and mapped to a generator.
        ii. knot_dict: dictionary with strand labels (strings) as keys and two-element lists as entries. The first element of each list is the Gauss code segment (tuple of integers) corresponding to the strand. The second element is a list of two-tuples, one for each crossing for which this strand is the overstrand. The elements of the two-tuples (strings) are the understrands of the crossing.
    b. Output:
        i. Boolean flag: True if the Wirtinger relations are satisfied by the mapping at every crossing (thus making the mapping a surjective homomorphism). False otherwise.

5. transpose_product(overstrands, understrands): Computes the conjugation of two group elements.
    a. Input:
        i. overstrands: Either: a SageMath matrix object if a search using *D* or *H* is being performed; or a two-tuple of integers if a search using *S* is being performed. This is the generator that will do the conjugating. Must be involutory.
        ii. understrands: Either: a SageMath matrix object if a search using *D* or *H* is being performed; or a two-tuple of integers if a search using *S* is being performed. The conjugate of this generator will be computed via overstrands.
    b. Output:
        i. product: Either: a SageMath matrix object if a search using *D* or *H* is being performed; or a two-tuple of integers if a search using *S* is being performed. Equal to overstrands*understrands*overstrands.
6. cox_writer(cox_gen_set): Formats the seed-strand-to-generator mapping for easier viewing by a human.
    a. Input:
        i. cox_gen_set: dictionary with strand labels (strings) from the seed_strand_set as keys and *D* or *H* group generators (SageMath matrices) as entries. Intended to be the initial strand mapping that the program used to find a homomorphism.
    b. Output:
        i. cox_gen_set_writable: dictionary identical to cox_gen_set, but with entries replaced by strings containing the information that was previously imbedded in the matrices. For example, the matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

would have the string representation:

"(1, 2, 3)|(4, 5, 6)|(7, 8, 9)"

Module: gen

Functions:

1. file_retriever(grp,g_ord): Retrieves the Excel file containing the pruned generating sets of a given group. Note that the "pruned" set of generating sets is not an exhaustive list of all generating sets of a given group. It is instead a subset of such a list which we have proven to be sufficient for an exhaustive search for surjective homomorphisms.[5]
    a. Input:
        i. grp: string; either "D" if *D* group generators are to be retrieved or "H" if *H* group generators are to be retrieved.
        ii. g_ord: integer. Coxeter rank of the group for which generators are to be retrieved.
    b. Output:
        i. wksht: xlrd worksheet object, corresponding to the worksheet in the Excel file on which the generating sets are listed
        ii. myfile: xlrd workbook object, corresponding to the Excel file containing the generating sets
2. entry_processor(gen_str): Converts list-notation of a square matrix, such as appears in the D4_gens_pruned.xlsx, H3_gens_pruned.xlsx, etc. files, into a list of RationalExpression objects.
    a. Input:
        i. gen_str: string consisting of the list-notation of a square matrix, such as appears in the D4_gens_pruned.xlsx, H3_gens_pruned.xlsx, etc. files. May include integers, rationals, and the value "a", which is defined to be sqrt(5).
    b. Output:
        i. gen: List of SageMath RationalExpression objects.
3. reader_main(grp,g_ord): Reads all pruned generating sets of the [grp] algebraic group of rank [g_ord] and converts them into a list of lists of SageMath matrix objects.
    a. Input:
        i. grp: string; either "D" if *D* group generators are to be retrieved or "H" if *H* group generators are to be retrieved.
        ii. g_ord: integer. Coxeter rank of the group for which generators are to be retrieved.
    b. Output:
        i. master: List of lists of SageMath matrix objects. Each sublist has length [g_ord], and is a generating set of the group [grp] with Coxeter rank [g_ord]. The elements of each generating set are square matrices of dimension [g_ord] with RationalExpression entries.
4. sym_gen_crafter(all_perms): Generates the pruned list of generating sets of transpositions for S_3, S_4, S_5, and S_6.
    a. Input:
        i. all_perms: List with at least one element.
    b. Output:
        i. all_perms: List with at least one element. The first element will be a tuple of four sub-tuples. The *i*th sub-tuple is the set of generating sets of the S_i group. Each generating set is itself a tuple of two-tuples, the two-tuples being

[5] R. Blair, A. Kjuchukova, and N. Morrison. Coxeter quotients of knot groups through 16 crossings. In preparation, 2020.

transpositions. All elements of all_perms beyond the first element, if they exist, will be identical to the elements in the input list.

5. d_gen_crafter(user_wirt, all_perms): Retrieves the pruned list of generating sets of reflections for D_4 and D_5 from D4_pruned_gens.xlsx and D5_pruned_gens.xlsx, respectively.
    a. Input:
        i. all_perms: List with at least three elements.
        ii. user_wirt: Integer, indicating for which group generators must be retrieved. A value of 0, 4, or 5 will alter the output, as described below. Anything else will result in an output list equal to the input list.
    b. Output:
        i. all_perms: List with at least three elements. The second element will be either: a tuple of four-element lists, each list being a D_4 generating set, if user_wirt=4 or 0; or, the same as in the input list otherwise. The third element will be either: a tuple of five-element lists, each list being a D_5 generating set, if user_wirt=5 or 0; or, the same as in the input list otherwise. The entries in the generating set lists will be SageMath matrix objects populated with RationalExpressions. The first element of all_perms and any elements beyond the third, if they exist, will be identical to the elements in the input list.
6. h_gen_crafter(user_wirt, all_perms): Retrieves the pruned list of generating sets of reflections for H_3 and H_4 from H3_pruned_gens.xlsx and H4_pruned_gens.xlsx, respectively.
    a. Input:
        i. all_perms: List with at least five elements.
        ii. user_wirt: Integer, indicating for which group generators must be retrieved. A value of 0, 3, or 4 will alter the output, as described below. Anything else will result in an output list equal to the input list.
    b. Output:
        i. all_perms: List with at least five elements. The fourth element will be either: a tuple of three-element lists, each list being an H_3 generating set, if user_wirt=3 or 0; or, the same as in the input list otherwise. The fifth element will be either: a tuple of four-element lists, each list being an H_4 generating set, if user_wirt=4 or 0; or, the same as in the input list otherwise. The entries in the generating set lists will be SageMath matrix objects populated with RationalExpressions. The first three elements of all_perms and any elements beyond the fifth, if they exist, will be identical to the elements in the input list.

Known Limitations

1. Attempting to input knots with more than 702 strands (or, equivalently, 702 crossings) will lead to a fatal error and/or nonsense output. The current calc_wirt.py algorithm cannot handle more than 702 strands as it runs out of strand labels. This number may be increased by editing the find_strands() function in calc_wirt.py.
2. Symmetric groups of Coxeter rank greater than 6 and *D* groups of Coxeter rank greater than 5 are not implemented. Thus, no search for surjective homomorphisms can be carried out on knots with Wirtinger number greater than or equal to 6.
3. There is no check for the validity of Gauss code. Invalid Gauss code will lead to a fatal error and/or nonsense output.
4. Attempting to import an Excel file that does not exist, or attempting to write an Excel file to a nonexistent directory, will result in a fatal error.
5. The package is compatible with MacOS, Linux, and Windows environments. Use of other environments may require alteration of directory naming in gen.py.