Introduction to computational group theory

Van Thuan Romoli

Semester 2, 2022/2023

1 Introduction

The aim of this project is to explore a particular way of constructing groups which is fundamental throughout group theory. To get there, we will need to see some ways computers can represent and work with binary operations (Section 2) and equivalence relations (Section 3). In Section 4, we will bring these together to construct groups via coset representatives.

2 Binary operations

We can represent binary operations by their Cayley tables. For example, the Cayley table

$*$	0	1	2
0	0	1	0
1	0	0	0
2	0	1	1

represents a binary operation where, for example, $2 * 1 = 1$ and $1 * 2 = 0$. The entry in row x and column y is the value of $x * y$.

The binary operation given by the table above could be represented in Python by a list of lists like so:

```python
X = [[0, 1, 0], [0, 0, 0], [0, 1, 1]]
```

Note that each of the nested lists corresponds to a row of the Cayley table, and the product $x * y$ is given by `X[x][y]`. Indeed, we observe that

```python
# The product of 0 and 1, using the binary operation X
X[0][1]
# Output: 1

# The product of 1 and 2 under X
X[1][2]
# Output: 0
```

This way of getting the product only works because the Cayley table is defined on the set $\{0, 1, \dots, n-1\}$ for some positive integer n. For the purposes of this project, we may assume that all binary operations will be in this form. Moreover, we assume that a binary operation is a list `X` such that:

1. The length of `X` is a positive integer n.
2. Each entry of `X` is a list of length n, called a *row*.
3. Each row of `X` contains only `int` values $x \in \{0, 1, \dots, n-1\}$.

Now that we explained our setup, we wish to define functions that verify some of the properties of a binary operation.

We start by defining the function `validate_binary_operation(X)` that returns `True` if `X` is a valid binary operation, and raises an appropriate `TypeError` or `ValueError` otherwise.

```python
def validate_binary_operation(X):
    # Binary operations must be lists:
    if not isinstance(X, list):
        raise TypeError(f"binary operations should be lists, but got a {type(X).__name__}")

    # Binary operations must have positive length:
    if not len(X) > 0:
        raise ValueError("binary operations should be positive-length, but got a list of length 0")

    for row in X:
        # Each entry of a binary operation must be a list:
        if not isinstance(row, list):
            raise TypeError("binary operations should be lists of lists, "
                            + f"but got a list containing a {type(X).__name__}")

    # Each row must be of length n
    for row in X:
        if not len(row) == len(X):
            raise ValueError(f"binary operations should have the length of its rows equal to the number of rows, {len(X)}, "
                            + f'but got the row {row} of length {len(row)}')

    # Creates the set {0,1,...,n-1}
    validset=[]
    for i in range(len(X)):
        validset.append(i)

    # Binary operations must be defined on {0,1,...,n-1}
    for row in X:
        for x in row:
            # Each element in each row must be an integer
            if not isinstance(x, int):
                raise TypeError('each element in each row should be an integer, '
                                + f'but got a row containing a {type(x).__name__}')

        for x in row:
            # Each element in each row must be from the set {0,1,...,n-1}
            if x not in validset:
                raise ValueError('each element in each row should be an integer from the set {0,...,' + f'{len(X)-1}' + '} '
                                + f'but row {row} a contains a {x}')
    return True
```

Recall that a binary operation $*$ on a set A is *associative* if $a*(b*c) = (a*b)*c$ for all $a,b,c \in A$. If we were asked whether an operation is associative, we should either prove that it is, or provide a counterexample to show that it is not. A very direct way of proving it would be to check every choice of three values $a,b,c \in A$, but this quickly becomes too much work to do by hand. For example, if $|A| = 50$, we would need to check more than one hundred thousand triples - and each one requires us to compute 4 products! This sort of direct verification is best done by computer.

We write a function `is_associative(X)` which takes a binary operation (as defined above) `X` and returns `True` if the operation is associative, and `False` if not.

```python
def is_associative(X):
    validate_binary_operation(X)
    for i in range(len(X)):
        for j in range(len(X)):
            for k in range(len(X)):
                if not X[i][X[j][k]] == X[X[i][j]][k]:
                    return False
    return True
```

Recall that in general, a binary operation $*$ on a set A has *identity* e if $e*a = a*e = a$ for all $a \in A$. In the context of this project, $e \in \{0,1,\ldots,n-1\}$ is an identity of a list-of-lists binary operation `X` if and only if the row and column with index e are both equal to `[0, 1, ..., n - 1]`.

We write a function `get_identity(X)` which takes a binary operation `X` and returns the identity of the binary operation, or `None` if there is no identity. We also write a function `has_identity(X)` which takes a binary operation `X` and returns whether it has an identity using the function `get_identity(X)`.

```python
def get_identity(X):
    validate_binary_operation(X)
    for i in range(len(X)):
        row = X[i]
        column = []
        for j in range(len(X)):
            column.append(X[j][i])
        if row == column == list(range(len(X))):
            return i
    return None

def has_identity(X):
    validate_binary_operation(X)
    if get_identity(X) == None:
        return False
    else:
        return True
```

Recall that in general, for a set A under the binary operation $*$, the element $a \in A$ has an inverse if there exists $b \in A$ such that $a * b = b * a = e$. In such case, the inverse of a is b.

We write a function `get_inverses(X, i)` which takes a binary operation `X` on $\{0, 1, \ldots, n-1\}$ and an element `i` in $\{0, 1, \ldots, n-1\}$ and returns the inverse of `i` under `X` if `i` has an inverse, and `None` otherwise. We also write a function `has_inverses(X)` which takes a binary operation `X` (on $\{0, 1, \ldots, n-1\}$) and returns whether every $x \in \{0, \ldots, n-1\}$ has an inverse under `X`, using `get_inverses(X)`.

```python
def get_inverse(X, i):
    validate_binary_operation(X)
    e = get_identity(X)
    for j in range(len(X)):
        if X[i][j] == X[j][i] == e:
            return j

def has_inverses(X):
    validate_binary_operation(X)
    for i in range(len(X)):
        if get_inverse(X,i) == None:
            return False
    return True
```

Recall that in general, a binary operation $*$ on a set A defines a group if it is associative, has an identity element, and has an inverse element for every element of the set. We may now verify, using the functions above, whether `X` defines a group. We define `validate_group(X)` that returns `True` if `X` defines a group, and raises an appropriate `TypeError` otherwise.

```python
def validate_group(X):
    validate_binary_operation(X)
    if is_associative(X) == False:
        raise TypeError('The binary operation is not associative')
    if has_identity(X) == False:
        raise TypeError('The binary operation has no identity')
    if has_inverses(X) == False:
        raise TypeError('At least one element has no inverses')
    return True
```

3 Equivalence relations

A relation $\sim$ on a set A can be defined by a subset $R \subseteq A \times A$, where for any $a, b \in A$, we have $(a, b) \in R$ if and only if $a \sim b$. This set is an easy way of defining relations to a computer, though there are other options. Note that as with the binary operations, this representation relies on the underlying set being $\{0, 1, \ldots, m\}$, which will always be the case in this project. Below, we show an example.

```python
my_relation = {(1, 2), (2, 1), (3, 3)}
```

As done in the previous section, we define functions that verify some properties of a relation.

Recall that a relation `rel` on $\{0, 1, \ldots, m\}$ is reflexive if (a, a) is in `rel` for all a in $\{0, 1, \ldots, m\}$. We define `is_reflexive_relation(rel,m)` which takes a relation `rel` and `m`, and returns `True` if it is reflexive and `False` otherwise.

```python
def is_reflexive_relation(rel, m):
    # Generates the set {0,1,...,m}
    A = []
    for i in range(m+1):
        A.append(i)

    # We'll assume for the moment that rel is a binary relation, but let's check that it
     is over A.
    # It's not clear whether we should raise an exception or return False if it isn't.
    for (x, y) in rel:
        if not (x in A and y in A):
            return False

    for a in A:
        if not (a, a) in rel:
            return False
    # If we reach this point in the function, we have checked each element of A
    # and found that they all have (a, a) in rel, so rel is reflexive.
    return True
```

Recall that a relation `rel` on $\{0, 1, \ldots, m\}$ is symmetric if whenever (a, b) is in `rel`, so is (b, a), for all a, b in $\{0, 1, \ldots, m\}$. We define `is_symmetric_relation(rel)` which takes a relation `rel` and returns `True` if it is symmetric and `False` otherwise.

```python
def is_symmetric_relation(rel):
    for (x, y) in rel:
        if not (y, x) in rel:
            return False
    return True
```

Recall that a relation `rel` on $\{0, 1, \ldots, m\}$ is transitive if whenever (a, b) and (b, c) are in `rel`, so is (a, c), for all a, b, c in $\{0, 1, \ldots, m\}$. We define `is_transitive_relation(rel)` which takes a relation `rel` and returns `True` if it is transitive and `False` otherwise.

```python
def is_transitive_relation(rel):
    for (x, y) in rel:
        for (w, v) in rel:
            # We test that if (w, v) is actually (y, v), then
            # (x, v) should be in rel by transitivity.
            if y == w and not (x, v) in rel:
                return False
    return True
```

Finally, we recall that an equivalence relation is a relation that is reflexive, symmetric, and transitive. We write a function `validate_equivalence_relation(rel,m)` which takes a relation `rel` defined on $\{0, 1, \ldots, m\}$ and `m`, and returns `True` if `rel` is an equivalence relation, and raises an appropriate `ValueError` otherwise.

```python
def validate_equivalence_relation(rel,m):
    if is_symmetric_relation(rel) == False:
        raise ValueError('The equivalence relation is not symmetric')
    if is_reflexive_relation(rel,m) == False:
        raise ValueError('The equivalence relation is not reflexive')
    if is_transitive_relation(rel) == False:
        raise ValueError('The equivalence relation is not transitive')
```

So far, we have used a straightforward explicit way of representing the relation. It is often more convenient (and faster) to convert it to another representation, which we call *adjacency set representation*. The idea is to give a *list-of-sets* where the item of the list with index i is a set that contains all of the elements j such that (i, j) is in the relation. For example, the relation

```
{(0, 1), (0, 0), (2, 3), (3, 1), (3, 3)}
```

would have adjacency set representation

```
[{0, 1},     # (0, 0) and (0, 1) are in the relation, so index 0 contains {0, 1}
 set(),      # There are no pairs (1, x), so index 1 contains the empty set, set()
 {3},        # The pair (2, 3) is the only one starting with a 2, so index 2 contains {3}
 {1, 3}]     # (3, 1), and (3, 3) are both in the relation
```

Mathematically speaking, this is equivalent to giving a list of equivalence classes where the item with index i is the equivalence class $[i] = \{j \in \{0, 1, \dots, n-1\} : (i, j) \in R\}\}$.

We write the function `adjacency_sets(R)` which has parameters a binary relation `R` on $\{0, 1, \dots, m\}$ and the integer `m` and returns the adjacency set representation of `R`.

```python
def adjacency_sets(R, m):
    adjacencylist = []
    for i in range(m+1):
        adj_ith_entry = []
        for j in range(len(R)):
            if list(R)[j][0] == i:
                adj_ith_entry.append(list(R)[j][1])
        adjacencylist.append(set(adj_ith_entry))
    return adjacencylist
```

Given an equivalence relation R on a set A, we often want to find a *set of representatives* B of the equivalence classes of R. This is a set B which contains precisely one element from each equivalence class of R.

For example, if R is "congruence modulo 5" and A is $\mathbb{Z}$, then one set of representatives could be $\{0, 1, 2, 3, 4\}$. Another perfectly good set of representatives could be $\{0, -4, 7, 23, -1\}$.

Below is some pseudocode for efficiently finding an (ordered) list of representatives of $\{0, 1, \dots, m\}$ under an equivalence relation R:

Algorithm 1 Computation of list of representatives

```
1: Start with a set $A = \{0, 1, \dots, m\}$, and a list $B$ which is initially empty.
2: Get the adjacency set representation of $R$; call this $L$.
3: for $i$ from 0 to $m$ do
4:     if $i$ is still in $A$ then
5:         Append $i$ to $B$.
6:         Remove all of the elements in $L[i]$ from $A$.
7:     end if
8: end for
```

Not only does this produce a set of representatives B, but B will have the additional property that each representative is the *minimal* element in its equivalence class. We will call this the (ordered) list of minimal representatives.

We write a function `minimal_representatives(R,m)` which takes a binary relation `R` (as a set of tuples) and a positive integer `m`, and returns the list of minimal representatives as described above.

```python
def minimal_representatives(R, m):
    A = []
    for i in range(m+1):
        A.append(i)

    B = []

    L = adjacency_sets(R,m)

    for i in range(m+1):
        if i in A:
            B.append(i)
            for j in range(len(L[i])):
                if list(L[i])[j] in A:
                    A.remove(list(L[i])[j])
    return B
```

Finally, it will be useful to write a function `minimal_rep(R,i)` which takes an equivalence relation `R` (as a set of tuples) and a number `i`, and returns the minimal element in the equivalence class of `i`.

```python
def minimal_rep(R, i):
    L = adjacency_sets(R,i)
    return min(L[i])
```

4 Coset representatives construction

Throughout this section, `X` will denote a binary operation on $\{0,1,\ldots,n-1\}$ which defines a group, and `X` will denote a subset of $\{0,1,\ldots,n-1\}$ which forms a subgroup under this binary operation. We will represent `X` as in Section 2, and we will represent H by a Python `set`.

Given any group G with subgroup $M \leq G$, we can always define an equivalence relation $R_M = \{(a,b) \in G \times G \mid ab^{-1} \in M\}$.

We write a function `coset_equivalence` which takes two parameters `X` and `H` as above and returns the equivalence relation R_H (represented as a set of tuples).

```python
def coset_equivalence(X, H):
    validate_group(X)
    R = []
    # ab^-1 can be represented by X[a][get_inverse(X,b)]
    # We create a list of all the elements in X = [[row1],[row2],...,[row(n)]]
    elements = []
    for row in X:
        for i in range(len(row)):
            if not row[i] in elements:
                elements.append(row[i])
    #print(elements)

    for i in range(len(elements)):
        for j in range(len(elements)):
            if X[i][get_inverse(X,j)] in H:
                R.append((i,j))

    return set(R)
```

Suppose that `X`, `H`, and R_H are as above.

We will define a new group on the set $\left\{0,1,\ldots,\frac{n}{|H|}-1\right\}$ as follows:

Algorithm 2 Coset representatives construction

```
1: Let reps be the ordered list of minimal representatives of $\{0,1,\ldots,n-1\}$ under the relation $R_H$;
   this will always have length $\frac{n}{|H|}$.
2: for all $i,j \in \left\{0,1,\ldots,\frac{n}{|H|}-1\right\}$ do
3:     Let a = reps[i] and b = reps[j].
4:     Let c = X[a][b].
5:     Let d be the minimal representative in the equivalence class of c.
6:     Let k be the index of d in reps.
7:     Set $i * j =$ k.
8: end for
```

In this project, we will call this the *coset representatives construction*. It is a particular way of representing *quotient groups*. In order for this construction to work, `H` needs to be a normal subgroup of `X`.

It bears repeating: this is one way of representing a quotient group on a computer. It is not the definition of the quotient group.

Below, we write a function `coset_rep_group` which takes `X` and `H` as above, and returns the binary operation `Y` on $\left\{0, 1, \ldots, \frac{n-1}{|H|}\right\}$ given by the coset representative construction.

```python
def coset_rep_group(X, H):
    validate_binary_operation(X)
    R_H = coset_equivalence(X,H)
    reps = minimal_representatives(R_H,len(X)-1)

    M = []

    for i in range(len(reps)):
        a = reps[i]
        temp_M = []
        for j in range(len(reps)):
            b = reps[j]
            c = X[a][b]
            d = minimal_rep(R_H,c)
            temp_M.append(d)
        M.append(temp_M)
    return M
```

We add below some examples that illustrate how the function works. But first, we define the function `modular_addition(m)` which takes one parameter `m` and returns the Cayley table of $\mathbb{Z}/m\mathbb{Z}$ under addition.

```python
def modular_addition(m):
    cayley = []
    for i in range(m):
        row = []
        for j in range(m):
            row.append((i+j)%m)
        cayley.append(row)
    validate_binary_operation(cayley)
    validate_group(cayley)
    return cayley


modular_addition(4)
[[0, 1, 2, 3], [1, 2, 3, 0], [2, 3, 0, 1], [3, 0, 1, 2]]
```

Example 1 Given $\mathbb{Z}/4\mathbb{Z}$ and the subset $\{0, 2\}$, we should get $\mathbb{Z}/2\mathbb{Z}$ back.

```python
Z_4Z = modular_addition(4)
H = {0,2}
G_quotient_H = coset_rep_group(Z_4Z,H)
Z_2Z = modular_addition(2)

print(f"Z_4Z = {Z_4Z}")
print(f"H = {H}")
print(f"G/H = {G_quotient_H}")
print(f"Z_2Z = {Z_2Z}")
print(G_quotient_H == Z_2Z)



Z_4Z = [[0, 1, 2, 3], [1, 2, 3, 0], [2, 3, 0, 1], [3, 0, 1, 2]]
H = {0, 2}
G/H = [[0, 1], [1, 0]]
Z_2Z = [[0, 1], [1, 0]]
True
```

Example 2 Given $\mathbb{Z}/40\mathbb{Z}$ and the subset $\{0, 10, 20, 30\}$, we should get $\mathbb{Z}/10\mathbb{Z}$.

```
Z_40Z = modular_addition(40)
H = {0,10,20,30}
G_quotient_H = coset_rep_group(Z_40Z,H)
Z_10Z = modular_addition(10)

print(G_quotient_H == Z_10Z)


True
```

5 Conclusion

In this project we explored a particular way of constructing quotient groups using the coset representatives construction. In section 2 we showed how we may use computers to represent binary operations. In particular, we wrote functions that check whether the properties of a group hold: associativity, identity, inverses. In section 3 we presented a way of defining equivalence relations using computers. We showed how we can obtain the equivalence classes and then a minimal set of representatives. Finally, in section 4 we wrote a function that constructs quotient groups using the coset representatives construction.